FIRMRES: Exposing Broken Device-Cloud Access Control in IoT Through Static Firmware Analysis

Yuting Xiao*, Jiongyi Chen†(✉), Yupeng Hu*(✉), Jing Huang*
*College of Computer Science and Electronic Engineering, Hunan University, Changsha, China
Email:{xyt990626, yphu, huangjjing}@hnu.edu.cn
†College of Electronic Science and Engineering, National University of Defense Technology, Changsha, China
Email:{chenjiongyi}@nudt.edu.cn

***Abstract*— Device-cloud interfaces are a critical component of IoT given their centrality of the cloud-side control over the connected devices, which has attracted an increasing number of attacks exploiting their access control. Regrettably, there is a lack of techniques to facilitate the examination of such a critical interface, primarily hindered by the challenges of dynamic firmware analysis to reconstruct device-cloud messages and generate testing cues.**

This paper presents FIRMRES, a principled static approach that automatically reconstructs device-cloud messages by modeling message construction semantics in IoT firmware. At the center of FIRMRES is a message field tree which is formed of the backward data flows from message delivery callsites to the potential sources of message fields. By walking through, transforming, and contextual learning from this tree, device-cloud messages are automatically reconstructed and a set of semantics during "message construction" such as the message format, the field semantics, and the order of the fields are inferred. Facilitated with the messages reconstructed by FIRMRES, we were able to manually examine the access control of device-cloud interfaces. FIRMRES reconstructed 246 effective messages from the firmware of 20 IoT devices, leading to the discovery of 13 previously-unknown vulnerabilities in their clouds.

I. INTRODUCTION

The device-cloud interface holds paramount significance within the IoT ecosystem, serving as the linchpin for device control. This interface conveys critical messages responsible for device control, enabling devices to access cloud resources. Regrettably, there has been a notable surge in attacks targeting these interfaces in recent years [1], [10].

Device-to-cloud functionality primarily includes device registration, remote management, and data transmission. Device registration typically refers to the process where a device initiates registration and binds with the cloud, associating legitimate devices with a user's cloud account for remote management. Remote management allows users to remotely manage and control IoT devices through the cloud, including making configurations, performing software upgrades, and remotely restarting devices. Data transmission enables devices to transmit various types of information like sensor data and device status to the cloud platform, facilitating real-time viewing, data collection, and profiling for users. Generally, device-to-cloud functionality is achieved by authenticating a device to the cloud and establishing a binding relationship between the device and the user. After that, the cloud still needs to perform stringent checks on the credentials provided by the device, ensuring that it is not vulnerable to impersonation attacks. However, many manufacturers do not strictly adhere to this process. Attackers can exploit such broken access control to communicate with the cloud, resulting in severe consequences such as privacy breaches, data tampering, disruption of device functionality, and device control.

(✉):Corresponding authors

However, uncovering potential threats in device-cloud interfaces of IoT still remains a labor-intensive and predominantly manual process. The challenge primarily stems from the inherent difficulty in reconstructing the messages responsible for cloud access from the device side (we call them "device-cloud messages" in this paper). This complexity arises because, traditionally, the construction of device-cloud messages has relied on dynamic analysis techniques. These techniques encompass dynamic program instrumentation/hooking or real-time execution with the interception of communication traffic. However, dynamic analysis for firmware images is hard. Creating a unified emulation environment for dynamic firmware analysis is a formidable hurdle in many cases. Diverse devices often employ distinct architectures and hardware configurations, making it challenging to build generic emulators to achieve effective device simulation. Furthermore, devices frequently interact with specific hardware peripherals and sensors, which significantly amplifies the simulation's complexity.

In this paper, we present FIRMRES, a new solution that automatically reconstructs device-cloud messages using a novel static analysis technique. FIRMRES enables analysts to efficiently assess the access control issues in device-cloud interfaces by forging device-cloud messages and impersonating real devices. Specifically, given the fact that a bunch of executables are presented in the firmware images, FIRMRES first leverages a statistic-based approach to identify the specific device-cloud executables by pinpointing request handlers that interact with the cloud, with statistics of the branches that check the validity of message fields. Then, it performs static taint analysis from message delivery functions to the sources of message fields and constructs a message field tree. This tree, composed of data flows from taint sources to taint sinks, not only helps identify message fields but also preserves the "message construction" logics. Subsequently, FIRMRES generates "message construction" code slices based on the paths of the tree and

2158-3927/24/$31.00 ©2024 IEEE
DOI 10.1109/DSN58291.2024.00054

uses a deep learning model to recover the field semantics. After that, the message field tree is simplified by removing the nodes that are irrelevant to field concatenation. The remaining nodes in the tree structure can be leveraged to recover the order of the message fields and reconstruct the messages. In the end, FIRMRES checks whether the forms of reconstructed messages align with pre-defined primitive messages.

We evaluated FIRMRES on the firmware of 22 devices from 18 vendors. The devices belong to 7 different device types. The evaluation shows that FIRMRES is effective in reconstructing device-cloud messages and is demonstrated useful in assessing the access control of device-cloud interfaces. FIRMRES successfully identified the device-cloud executables in 20 of the devices and reconstructed 246 valid messages. By probing the cloud-side interfaces with the reconstructed messages, we identified 14 vulnerabilities including 13 of them that were previously unknown. These identified vulnerabilities could lead to severe consequences such as uploading false logs, illegally viewing cloud storage footage, triggering alarms, and hijacking devices.

Contributions. The contributions of this paper are summarized as follows:

- **New Problem.** To the best of our knowledge, this work makes the first attempt to automatically reconstruct device-cloud messages through static firmware analysis. Facilitated with the reconstructed messages, systematic assessment of access control issues in device-cloud interfaces of IoT is made possible.
- **New Approach.** We present a principled static approach and develop FIRMRES to reconstruct device-cloud messages from firmware, without relying on dynamic firmware analysis. This involves a set of new techniques that utilize the "message construction" logics in the firmware to recover semantics and reconstruct messages.
- **Real-world Impact.** The evaluation demonstrates that FIRMRES is effective in reconstructing messages and facilitating the assessment of the access control in IoT device-cloud interfaces. The identified vulnerabilities could lead to serious consequences. This tool is open sourced[1].

II. BACKGROUND

A. Device-Cloud Communication

As illustrated in Fig. 1, an IoT system typically includes three parties: the cloud, the IoT device, and the user's management console (mobile apps or web). The cloud manages the remote communication between the device and the user, acting as the center of the system. In device-cloud communication, the cloud forwards the commands issued by the users and receives the sensor data uploaded by the device. The mainstream application-layer protocols are MQTT and HTTP. MQTT is a publish-subscribe messaging protocol that

[1]https://github.com/Nakuro1999/FirmRES

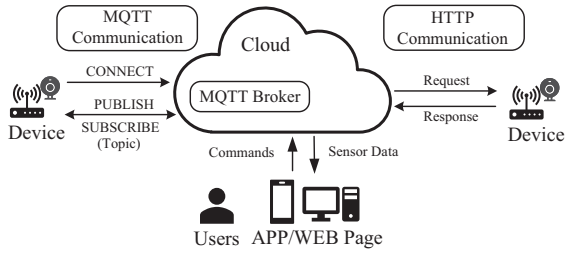

Fig. 1. Remote Communications in IoT Systems

allows the sender to deliver messages to a class [8], based upon the topic of the messages which can be subscribed by a group of receivers. The core component of the cloud is the MQTT broker, which hosts MQTT topics at the cloud server endpoint with each topic structured like a file path such as `/sys/properties/report`. Topics serve as the intermediary in the transmission between message publishers (Pubs) and message subscribers (Subs). Devices send and receive messages based on topics. On the other hand, HTTP is a request-response model protocol that is commonly used in device-cloud communication. The cloud based on HTTP receives the HTTP messages sent by the devices to collect the sensor data. HTTP is stateless, meaning that each message sent by the device is independent and not reliant on previous messages.

B. Device-Cloud Access Control

As shown in Fig. 2, the device-cloud interaction has two phases for the remote control of the devices: the binding phase where the validity and the authenticity of the device are proved to the cloud, and the business phase where the cloud confirms that the device is bound to a user and allows the device to access cloud resources. The message could contain a variety of fields, but only a few are used for access control. In this paper, we call the access control elements in the messages "primitives". They are *Dev-Identifier*, *Dev-Secret*, *User-Cred*, *Bind-Token*, and *Signature*. The primitives and arguments are derived from the reverse engineering of the existing designs in the real world.

Binding Phase. The main purpose of the access control in this phase is to verify the identity and authenticity of the device. The identity refers to the fact that the device is produced by the manufacturer. The authenticity concerns that the message is sent from a specific real device rather than an attacker. The device uses device identifiers (denoted by *Dev-Identifier*) such as MAC addresses, serial numbers, device IDs, or product IDs to prove its identity. The device secret (denoted by *Dev-Secret*) which is also known as the secret key, the device key, or the device certificate, is encoded by manufacturers to prove its authenticity. The user credential denoted by *User-Cred* represents the user's login credential. Once receiving and verifying *Dev-Identifier*, *Dev-Secret* and *User-Cred*, the cloud responds with a binding token (denoted by *Bind-Token*), which

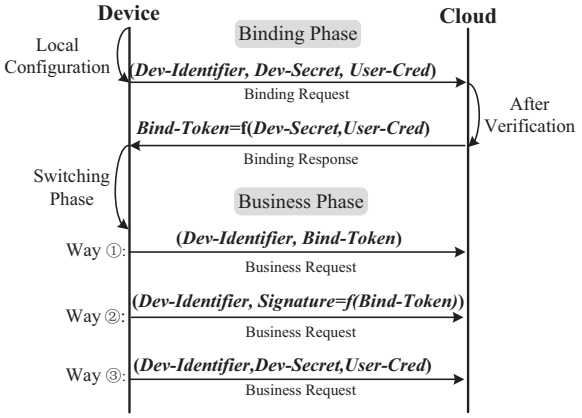

Fig. 2. Two Phases of Device-Cloud Access Control

is used for accessing resources in the subsequent business phase.

Business Phase. In this phase, the device accesses resources by sending a business request that proves the device is already bound with a specific user. This business request can be formed in three different ways: ① composition of *Dev-Identifier* and *Bind-Token*. The binding token is also known as the access token, the access key, or the session token; ② composition of *Dev-Identifier* and *Signature*. The *Signature* is also known as the temporary key or the temporary secret, which is derived from *Dev-Secret* (denoted by *Signature =f(Dev-Secret)*); ③ composition of *Dev-Identifier*, *Dev-Secret* and *User-Cred*. In this case, the device directly sends the original forms of primitives. The above are the forms of the business request that are made of primitives for access control. Meta data like the timestamp is not treated as primitives for access control.

III. Motivation

A. A Running Example

Conducting security assessments for the communication interface between the device and the cloud is often regarded as a challenging aspect of the IoT ecosystem due to the complexities involved in analyzing device firmware. This interface struggles with broken authentication and authorization. We show an example from CVE-2023-2586. Listing 1 is a code snippet of the vulnerable program `rms_connect` of the device model Teltonika RUT241. This program is in charge of device-cloud interactions including device authentication and requesting cloud resources.

The code snippet shows how a device-cloud authentication message is constructed in the device firmware. In function `FUN_00402098`, the program extracts the MAC address and writes it into the message `uVar1` for cloud communication at lines 2-5. The serial number is written into the message `uVar1` at lines 6-9. Finally, at line 18, the program uses the function `SSL_write()` to send the converted message `pcVar1` to the cloud. Listing 2 shows the message sent by the device to register to the cloud.

```
1  undefined4 FUN_00402098(void){...
2      if (DAT_00416508 != 0) {
3         uVar2 =  FUN_00404a20();
4         FUN_004047a4(uVar1,"mac",uVar2);
5      }// add MAC address to request message
6      if (DAT_00416504 != 0) {
7         uVar2 = FUN 00404a20();
8         FUN_004047a4(uVar1,"sn",uVar2);
9      }// add serial number to request message
10     ...
11     uVar2 = FUN_00404638(uVar1);                    //
        convert the format of the message
12     return uVar2;
13 }// Construct the message body of the binding request
14
15 int FUN_00402210(void){...
16     pcVar1 = (char *)FUN_FUN_00402098();          //get the
        message of the binding request
17     sVar8 = strlen(pcVar1);
18     local_2360 = SSL_write(s,pcVar1,sVar8 + 1);  //send the
         message to the cloud
19     ...
20 }
```

Listing 1. Code Snippet from CVE-2023-2586

```
1 {
2    "version":2,
3    "mac":"00:1e:42:**:**:**",    //Mac Address
4    "sn":"1123*",    //Serial Number
5    "certs":0,
6    "model":"RUT24101****",
7    "fw_version":"RUT2M_R_00.07.01.3\n",
8    "is_facelift":false
9 }
```

Listing 2. A Message Sent by Device to Identify Itself to Cloud

Unfortunately, the device only sends its serial number and the MAC address (i.e., Dev-Identifier) to the cloud to prove its identity and requests the private key and the certificate for subsequent MQTT communication. The message is shown in Listing 2. By leaking the serial number and the MAC address, attackers can register to the vendor's cloud and obtain the device certificate (i.e., *Dev-Secret*). Subsequently, they can use the acquired certificate to establish a connection with the vendor's MQTT broker, by forging device-cloud messages, enabling the attackers to communicate on behalf of the device and impersonate it. Such device authentication is not secure since the serial number and MAC address are simply weak identifiers. Once they are leaked, for example, through brute-forcing or device ownership transfer, the attackers can gain remote and complete control over the running devices through the cloud.

B. Threat Model, Research Goal, and Challenges

Threat Model. In this paper, the attackers aim to use device identifiers or hard-coded primitives to impersonate messages sent from victim's devices (acting as malicious registered devices). This could result in serious consequences to the victim's cloud resources such as uploading false logs, illegally viewing cloud storage footage, triggering false alarms, and even hijacking devices. We assume that the attacker can obtain the device identifiers, due to the weak protection in real life:

- Device discovery. When the devices are exposed on the Internet, attackers can scan their IPs and collect device information. For example, the attacker can discover the devices'

MAC address and serial numbers by using Shodan.io [9] to query SNMP services of Internet-exposed devices and querying the relevant OIDs [6].
- Inference of the device ID. Attackers may infer, bruteforce, or enumerate the device ID according to the regulation of ID sequence arrangement. For example, the ID randomness of some devices is not enough, which is easily guessed; the MAC addresses contain a vendor-specific field, which can be inferred by enumeration.
- Off-site physical interaction of the device. Device information can be leaked through device ownership transfer, including device reuse, reselling, stealing, and so on. For example, the attacker could purchase a device from Amazon, record its information, then sell it out or return it.

Research Goal. Given the stealthiness and severe consequences of IoT device-cloud interfaces using weak credentials for access control, it is imperative to discover them automatically and assess their impact in the real world. *As such, the goal of this research is to automatically expose the device-cloud messages, and systematically assess the device-cloud interfaces through a follow-up manual analysis.* Particularly, we focus on device-cloud access control issues that are caused by insufficient checks of the primitives in the cloud, including:

- Missing primitives in device-cloud messages. This allows attackers to impersonate the device to communicate with the cloud. Attackers could bind the device or access sensitive information of the victim.
- Hard-coded *Dev-Secret* in firmware. As long as the *Dev-Secret* is hardcoded and leaked, attackers can also impersonate the device to communicate with the cloud.

Challenges. Unfortunately, we found that there is no existing solution to generate device-cloud messages and test the device-cloud interfaces. Traditionally, the firmware needs to be executed or emulated to generate device-cloud messages. Then analysts can capture the message through dynamic instrumentation or in unencrypted network traffic. Regrettably, the dynamic firmware analysis still remains an open challenge [33]. It is difficult to perform full and transparent instrumentation of firmware on physical devices. On the other hand, firmware re-hosting requires the missing NVRAM parameters to be presented or systematic modeling of peripherals, in order to support the smooth execution of firmware images, which remains an unsolved problem yet.

IV. Approach

Given the difficulty and infeasibility in dynamic firmware analysis, we choose to reconstruct device-cloud messages based on static firmware analysis, for the purpose of facilitating the examination of device-cloud interfaces.

The workflow of FirmRES is shown in Fig. 3. The inputs are the firmware images of IoT devices. The firmware analysis module is automated and outputs testing cues and alarms of incorrect device-cloud messages. Particularly, FirmRES first identifies the program that interacts with the cloud by calculating the feature scores of all binary programs. Then, it performs a static taint analysis on the identified executable to identify the fields of device-cloud messages. After that, FirmRES generates code slices based on taint propagation paths. The semantics of the message fields are recovered using a deep learning model which is fed with those code slices with rich semantics. Next, FirmRES concatenates the fields into messages by grouping the fields and inferring the message format. Finally, an automated message form check is performed on the message with the recovered semantics to check the primitives issues. With FirmRES's help in preparing device-cloud messages, we are finally able to conduct a systematic assessment with manual efforts on the access control issues of device-cloud interfaces.

A. Pinpointing Device-Cloud Executables

The device-cloud executables are in charge of the interaction between the device and the cloud. Given the different types of executables presented in the firmware image, it is necessary to pinpoint the device-cloud executables. They share two significant features: first, device-cloud executables have request handlers that handle incoming requests and return responses; second, handling requests from the cloud is asynchronous, which means the cloud request handlers are often implemented with event-based implicit invocations. In general, the request handlers of other executables are explicitly invoked. As such, the device-cloud executables can be identified in the following two steps: (1). identifying the request handlers in all executables; (2). identifying the asynchronous request handlers among the identified request handlers.

- **Request Handler Identification.** A request handler consists of a set of functions that accept incoming requests, parse them, handle them, and send out responses. In request handles, a large portion of predicates are related to request parsing. The operands of the predicates are mostly request fields. As a result, the request handlers can be identified by calculating the statistics about how many operands of predicates are request fields.
  Specifically, We first define two types of anchor nodes, namely the callsites of the request incoming functions fun_{in}s (e.g., `recv/recvfrom/recvmsg` functions) and the callsites of the response outgoing functions fun_{out}s (e.g., `send/sendto/sendmsg` functions). Next, as shown in Fig. 4, we cluster them into pairs of incoming functions and outgoing functions by their closest distances on the call graph. Function call sequences between the anchor nodes can be regarded as handlers. Not all sequences are request handlers. For instance, IPC handlers are not request handlers. As such, we calculate a "string-parsing" factor for each sequence given below, to identify request handlers.

$$P_f = \frac{O_r}{O}, O \in f$$
$$score_S = \max_{f \in S}(P_f) \quad (1)$$

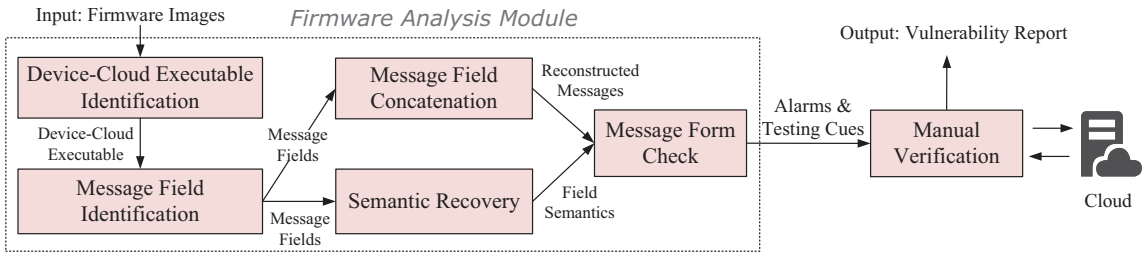

Fig. 3. Workflow of FIRMRES

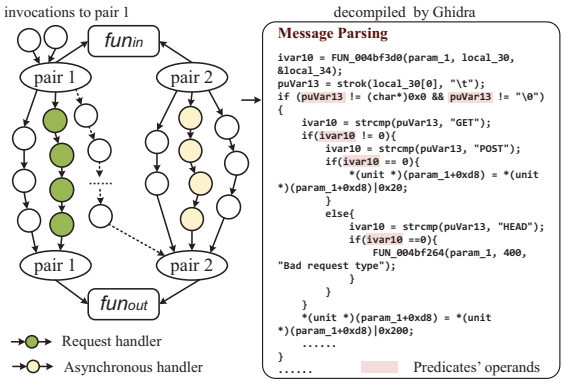

Fig. 4. Illustration of Asynchronous Handler Identification

O_r is the number of the predicates' operands that originate from the arguments of the callsite of the request incoming function (i.e., the incoming request) in function f. Function f belongs to sequence S. O is the total number of predicates' operands in f. The function with the maximum $\frac{O_r}{O}$ is regarded as the main parsing function. The maximum P_f is the feature score $score_S$ of the sequence S. The sequence with the highest score between the same anchor node pairs is identified as the request handler.

- **Asynchronous Handler Identification.** Asynchronous handlers have no direct control flow before the request incoming function invocation, as the handlers are often implemented with event-based implicit invocations. As shown in Fig. 4, the identified request handler in *pair 2* has no direct invocation to it, which can be regarded as an asynchronous request handler. To identify the asynchronous handler, we check in the identified request handlers whether there is any direct invocation of the callers of request incoming functions fun_{in}s or not. If not, it is considered an asynchronous handler.

As long as the asynchronous handler is identified in an executable, such an executable is regarded as the device-cloud executable.

B. Identifying Message Fields

To reconstruct the messages, the first step is to identify message fields. We observe that the message fields are concatenated in a certain order and formed as the arguments of network functions, which are eventually sent to the cloud. As a result, by defining data sources of the message fields, we can leverage backward taint analysis to capture the information flows from the arguments of network functions to the predefined sources for the aim of identifying message fields. In particular, the static backward taint analysis is given below.

- **Taint Sources.** Taint sources are the arguments of the callsites of the functions that send out the device-cloud messages. Those functions include SSL-message sending functions like `CyaSSL\_write()`, HTTP-message sending functions like `curl\_easy\_perform()`, and MQTT protocol functions like `mosquitto\_publish()`. The arguments of those functions are the variables that hold device-cloud messages.
- **Taint Sinks.** The taint sinks are the potential sources of the message fields, which could be constants from the data segment, values from the NVRAM or configuration files, and environment variables that originate from the front end. Examples of constants include request paths, presented in the form of function arguments (e.g., `sprintf(cloudpath, "?m=camera\&a=login")}`. The values read from the NVRAM or the configuration files are often related to networking (e.g., `"Host:www.linksyssmartwifi.com"`) or device information (e.g., MAC address and device model). Message fields from the front end are usually information provided by the user.

  Given the complexity of the program logic and semantics, it is impractical to enumerate all the sinks by defining their information sources. Fortunately, we observe that the program retrieves and stores the values of message fields piece by piece into variables or memory locations, then assembles the stored values of the fields into messages in the form of <key, value>. It means that for a single message field, the information source of the value stored in variables or memory locations cannot be further decomposed. Namely, a single-information-source variable cannot be decomposed into multiple other variables. Instead, during the assembly of message fields, some variables can hold composed fields that originate from multiple information sources. These variables storing the assembled fields can be further decomposed. As such, the single-information-source variables propagated from the taint sources are defined as taint sinks.

- **Propagation Rules.** Propagation rules are set to trace the complete process of message construction. We modified Ghidra's [5] intra-procedural dataflow analysis and implemented our own inter-procedural dataflow analysis to support taint propagation. Specifically, we trace backwards the callers of the taint sources on the call graph and analyze how the taint sources are defined. If the taint source is a parameter of its caller, all possible callsites of the caller would be analyzed. Backtracing dataflows in the callsites starts from the callsites' function's return. In addition, we write function summaries for commonly invoked system calls and library calls, to avoid time and memory costs during dataflow analysis.

C. Recovering Field Semantics

This step recovers the semantics of the message fields identified in Section IV-B. Traditionally, the semantics of the protocol fields are inferred from execution context using dynamic analysis [12], [17], which is not adaptable to our task. Our solution is to extract the code context of each field from the firmware code (by leveraging a tree structure), and train a deep learning model to learn the association between the code context and the field semantics. We observed that message fields like MAC addresses, serial numbers and usernames that are critical to access control are often presented in the form of key-value pairs. This feature can help the model to accurately recover the primitive semantics of the fields.

Message Fields Tree. To capture contextual information about the semantics of the field, FIRMRES generates a "message field tree" (MFT for short) based on the paths from taint sources to taint sinks. It takes the taint sources (e.g., the message arguments) as the root nodes and the taint sinks (e.g., the sources of message fields) as the leaf nodes. The paths from the leaf nodes to the root node represent message construction. The context on the paths can be used by the deep learning model to infer semantics. Thus we compute code slices for each path on the MFT, which represent the code context of the corresponding message fields. We observe that the message fields are primarily constructed in two ways: (1) using third-party libraries like `cJSON.so` to assemble the fields piece by piece. This naturally preserves the contextual semantics; (2) using formatted output functions like `sprintf` to assemble partial and highly-constructed messages. In this case, the `sprintf` function is invoked multiple times on the paths, meaning the format string with multiple fields will add noise to neural networks. It takes extra steps to handle the latter. Our solution is to separate the partial message into fields by delimiters before computing code slices and feeding them into the neural network. We identify the delimiters by splitting the formatted strings in the formatted output function and using clustering on the split substrings. The clustering similarity is calculated as $Similarity(a,b) = 2 * (L_{common}) / (L_a + L_b)$, where L_{common} is the length of the longest common subsequence between two substrings, L_a and L_b are the lengths of the two substrings. The message separation is illustrated in Listing 3.

```
sprintf(param_9, "%s %s HTTP/1.1\r\nUser-Agent: GooLink
    Terminal 0x%x\r\nHost: s%\r\nConnection: Keep-Alive\r\
    nContent-Type: application/x-www-form-urlencoded\r\
    nContent-Length: %d\r\n\r\n", &DAT 00074d60,"/
    storageweb/UpFileInfoReq.jsp",0x15010011,param_8,sVar1)
    ;

// after the split, new statements are created to replace
    the above
sprintf(param_9, "%s %s HTTP/1.1", &DAT 00074d60,"/
    storageweb/UpFileInfoReq.jsp");
sprintf(param_9, "User-Agent: GooLink Terminal 0x%x", 0
    x15010011);
.......
```

Listing 3. Separation of Partial Messages

Semantic Information Embedding. Given that the intermediate language P-Code provided by Ghidra [5] preserves qualified and accurate code semantics from binary code, we use it as the representation of the slices that contain code context. P-Code is a register transfer language designed for reverse engineering applications. It converts a single processor instruction into a series of P-Code operations that use portions of the processor state as input and output variables, which are also known as nodes. The basic form of P-Code is $< Address : Output\ OP\ Input_1, Input_2, ... >$, where $Address$ is the address of the P-code instruction. $Input_i$ nodes and $Output$ nodes are generalizations about registers or memory locations for the operands. OP is the operator, including a set of arithmetic and logical operations performed by a general-purpose processor. We extract semantic information from P-Code representation and embed them into the slices. Ghidra-extracted information includes the keywords about field semantics as well as the information in symbol tables, which enrich the context information of the slices and benefit the neural network training. Specifically, we embed semantic information into $Input_i$ nodes and $Output$ nodes according to their data types. The semantic information at the P-Code level includes:

- **Data Types.** Data types are the types of outputs and inputs of a P-Code instruction, including function, local variable, parameter, constant, and data pointer. It determines the information we embed.
- **Contents of Constants.** Constants includes numeric constants and string constants. In particular, the contents of string constants contain rich semantic information.
- **Names.** Names of local variables, parameters, data pointers, and functions are suitable sources of semantic information.
- **Node IDs.** There are variables, parameters, and data pointers with the same name in different functions. To avoid confusion, we randomly generate Node IDs for them to differentiate them.

The generic form of the semantic enriched representation of the $Output$ nodes and $Input_i$ nodes is $(Datatype, Name/Constant, NodeID)$. When the data type is a function, the semantic enriched representation is $(Fun, FunctionName)$. Similarly, when the data type is a local variable, parameter, or data pointer, the seman-

tic enriched representation is $(Datatype, Name, NodeID)$. When the data type is constant, the new representation is $(Constant, ConstantContent)$. For example, the original P-Code *CALL (ram,0x12bd4,8), (unique,0x1000024e,4), (register,0x2c,4)* is represented after processing as *CALL (Fun,printf), (Cons,"posting data of is %s"), (Local,finalBuf, v 1357)*, where data in the form of *(ram,0x12bd4,8)* are $Output$ nodes and $Input_i$ nodes.

Network Training. We select the BERT-TextCNN model [18] [23] for our semantics recovery task. The BERT model learns the field context as global features, and the TextCNN model learns the field semantic information as local features to improve recognition accuracy. We use code slices with embedded information as input to the model. The outputs of the network are *Dev-Identifier*, *Dev-Secret*, *User-Cred*, *Bind-Token*, *Signature*, *Address*, and *None*, which are mostly aligned with the primitives in Section II-B. In particular, *Address* is used to label the IP address of the communication (explained in Section IV-D), and *None* is used to label the type that does not belong to any primitives. The architecture of the model contains Input Layer, BERT Layer, TextCNN Layer and Fully Connected Layer. The parameter count of BERT is the default value. TextCNN uses a convolutional kernel of size (2,3,4,5) with a total of 268800 parameters. We leverage the Multi-Head-Self-Attention mechanism in the model training to make it focus on the features and accelerate the fitting of the network. The attention value can be calculated by:

$$\begin{aligned} &MultiHeadAttention(Q,K,V) \\ &= Concat\left(head_1, head_2, \ldots, head_k\right) W^0 \\ &head_i = Attention\left(QW_i^Q, KW_i^K, VW_i^V\right) \end{aligned} \tag{2}$$

where Q is the query matrices, K is the queried matrices, V is the actual feature matrices. They can be obtained through model training. W_i^Q, W_i^K and W_i^V represent the weight matrix of the i-th head for W^Q, W^K and W^V respectively. W^0 measures the additional weight matrix and $Concat()$ is the concatenation function. The model outputs probabilities of the defined types. The label of the output node with the highest probability is considered the message field semantics.

D. Concatenating Message Fields

The goal of this step is to reconstruct messages by concatenating identified message fields.

Field Grouping. FIRMRES first groups the code slices of the same message based on the MFT. This is achieved by matching the paths of the code slices with the MFT using depth-first search. FIRMRES numbers each path of the MFTs and assigns a hash value to each path for efficient matching. After grouping, we check the code slices whose recovered semantics are $Address$ in the group, extract all string constants, and check whether they are LAN IP addresses. If a LAN IP address is found, the corresponding MFT is discarded. The LAN IP addresses include `10.*.*.*`, `172.16-31.*`, `192.168.*.*`, IPv6 addresses beginning with `FE80`, common multicast addresses, and broadcast addresses.

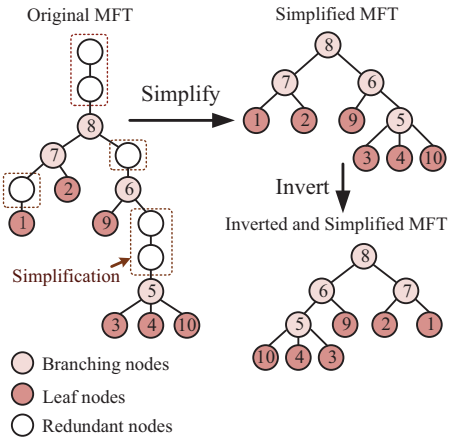

Fig. 5. MFT Transformation

Message Format Inference. Inferring the message format (with the correct order of the fields) is necessary as it is strictly checked by the cloud. Given that the content of the message body is often formed in a nested format (e.g., in `JSON/XML`), they can be easily inferred from the MFT representation. We process the MFTs by the following steps to recover the message format and the order of the fields:

- Simplifying the MFT. The format of the message can be reflected by the structure of the tree representation. Since MFT represents a message construction process, the nodes of MFT contain not only field concatenating operations but also field encoding and message formatting. In terms of message format inference, we need to remove redundant nodes in MFT. To this end, we only keep the branching nodes and the leaf nodes in the MFT, as shown in Fig. 5, because the branching nodes represent field concatenation and the leaf nodes are the fields.
- Inverting the simplified MFT. Since MFT is generated by backward taint analysis, early tagged fields are concatenated later into the message. Thus, we invert the simplified MFT for the aim of correctly ordering the fields in the messages.

According to field grouping, we add the annotation of the identified semantics of the field as a new leaf node to the corresponding path of the field. This could facilitate the check of message forms in Section IV-E.

E. Assessing Access Control Implementations

Assessing access control implementations is performed in two steps: automatic analysis and manual verification. The step of message form check identifies messages that lack primitives or hard-coded *Dev-Secret*. After that, to fill the fields with dynamically-generated values and verify that the vulnerabilities indeed exist, we perform manual verification.

Message Form Check. In this step, FIRMRES first checks the form of the messages to detect if there are missing primitives, based on the inverted and simplified MFT. Then it tracks the source of *Dev-Secret* and detects whether it is hard-coded.

- Checking the form of messages. To detect the lack of key primitives in the messages, We check whether the semantic annotations are aligned with correct primitive forms defined in Section II-B by examining the composition of the semantic annotations added in Section IV-D.
- Tracking the source of *Dev-Secret*. The Leak of *Dev-Secret* in firmware is typically caused by being hard-coded in the program context or through a readable configuration file. We identify the patterns when tracking the source of *Dev-Secret*: (1) `<Variable = Constant>`, meaning that a hard-coded constant exists; (2) `<Variable = Function(Constant)>`, representing that the constant is read from a file. In this case, we try to read the file from the firmware system (i.e., the file shown in `Constant`).

At the end of this step, FIRMRES outputs the organized contents of the messages and marks the messages that could potentially lead to security flaws.

Manual Verification. Manual verification is necessary, as some dynamically generated information is needed to send probing device-cloud messages. Corresponding to threat model mentioned in Section III-B, we developed three methods to obtain necessary device information. First, we use the Shodan [9] API to search for IP addresses of devices with open port 161 on the Internet, thus finding devices with exposed SNMP services. We obtain the OIDs [6] of device identifiers by querying device manufacturers' Management Information Base (MIB) files [6], then send an SNMP query request with OIDs to get the device identifiers. The plaintext transmission and overlooked default passwords of SNMP services make this approach viable. Second, device identifiers are easily guessable. For example, the first 3 bytes of a MAC address are fixed codes assigned by the registry to different manufacturers, and the last 3 bytes are assigned by the manufacturers themselves. So we can also enumerate them by brute force guessing in some cases. Third, when we have physical access to the device, we set up a device-cloud interaction environment in PCs to intercept the exchanged messages. When needed, we also import SSL certificates of firmware to bypass TLS authentication. Additionally, we perform a hardware read of the device's flash or NVRAM to obtain the device certificates not packaged in the firmware. Some vendors use the same device certificate for all users of the device model. In this case, certificates can be obtained by extracting certificates from any same model devices. Other vendors use unique device certificates for users. In this case, the device certificates can only be obtained from the victim's device during the device ownership transfer. With the results from automation, those procedures are manual operations about configuring and setting up testing environments, which cannot be solved by program analysis techniques.

In addition, not only are the responses needed to confirm that the vulnerabilities indeed exist, but also the responses themselves could include sensitive information. For example, some vendors return *Bind-Token* to the device for accessing cloud resources. We review all cloud responses to confirm whether there is any sensitive information leakage.

TABLE I
LIST OF EVALUATED DEVICES

Device ID	Device Model	Device Type	Firmware Version
1	InRouter: InRouter302	Industrial Router	V1.0.52
2	TP-Link: ***	Smart Camera	***
3	TP-Link: ***	Industrial Router	***
4	TP-Link: TL-TR960G	4G Router	0.1.0.5_Build_211202_Rel.47739n
5	Linksys: ***	Wi-Fi Router	***
6	Netgear: GC110	Smart Switch	V1.0.5.36
7	Netgear: R8500	Wi-Fi Router	V1.0.2.160_1.0.107
8	Netgear: WAC720	Wireless Access Point	V3.1.1.0
9	Araknis: AN-100FCC	Wireless Access Point	V1.3.02
10	TENDA: AC6	Wi-Fi Router	V02.03.01.114
11	Teltonika: RUT241	4G-LTE Wi-Fi router	RUT2M_R_00.07.0 1.3
12	360: C5S	Wi-Fi Router	V3.1.2.5552
13	Tenvis: 319W	Smart Camera	V3.7.25
14	Western Digital: My cloud	NAS	V5.25.124
15	Mindor: ZCZ001	Smart Plug	V1.0.7
16	Mank: WF-CT-10X	Smart Plug	V1.1.2
17	Cubetoou: T9	Smart Camera	a01.04.05.0020.5591a.190822
18	DF-iCam: QC061	Smart Camera	2.3.04.25.1
19	VStarcam: BMW1	Smart Camera	10.194.161.48
20	RUISION: S4D5620PHR	Smart Camera	1.4.0-20230705Z1s
21	MOFI: MOFI4500	4GXeLTE Router	2_3_5std
22	D-LINK: DAP1160L	Wireless Access Point	FW101WWb04

V. EVALUATION

This section presents the evaluation results. Particularly, Section V-A gives the implementation and experimental setup. In other sections, We present the effectiveness of the tool in device-cloud identification and message reconstruction, the identified vulnerabilities, the performance and a comparison with other works.

A. Implementation and Setup

FIRMRES is built upon Ghidra [5] and is implemented with around 6800 lines of Java code and 1600 lines of Python code. Notably, static taint analysis is implemented with Ghidra's representation Varnode [2] based on the P-Code [7], which leverages Ghidra's decompiler to recover variables from binaries and allows developers to implement customized dataflow analyses. The core of FIRMRES is run on an Intel Core i5 processor featuring 12 cores at 2.90GHz and 8 GB of RAM. The version of Ghidra is 10.1.3. The model is trained on the machine with a 13th Gen Intel Core i9-13900K processor and an NVIDIA GeForce RTX 4090 GPU.

We purchased 22 IoT devices from e-commerce websites. The device-cloud interaction of 20 devices (with ID 1 to 20 in Table I) is handled by binary executables, which effectively apply FIRMRES for evaluation. The devices are made by different vendors (ranging from top vendors to non-famous ones) and from various categories. The device types include industrial routers, home routers, smart cameras, smart plugs, wireless access points, smart switches and NAS devices. All these devices can be remotely operated through their cloud platforms. Table I presents the detailed information about the target devices.

TABLE II
OVERALL RESULTS OF MESSAGE RECONSTRUCTION

Device ID	Message Reconstruction		Field Identification		Semantics Recovery			
	#Identified	#Valid	#Identified	#Confirmed	*thd*=0.5	*thd*=0.6	*thd*=0.7	#Accurate
1	21	17	82	69	-	-	-	64
2	16	14	74	67	-	-	-	60
3	18	16	102	93	-	-	-	84
4	17	14	97	86	-	-	-	79
5	8	7	52	48	-	-	-	43
6	14	13	82	78	-	-	-	71
7	18	16	98	81	-	-	-	74
8	13	13	101	92	5	7	7	86
9	15	14	96	88	-	-	-	80
10	7	6	62	57	5	6	7	54
11	13	11	76	52	0	0	0	47
12	15	11	85	71	4	5	5	65
13	17	17	162	147	6	8	10	135
14	30	26	323	291	5	7	8	279
15	5	4	58	53	8	8	9	49
16	7	5	71	64	6	8	8	57
17	9	9	101	88	6	6	7	75
18	13	11	117	91	5	7	7	83
19	13	12	93	87	5	6	6	80
20	12	10	87	82	5	6	7	76
Average	14	12	100	89	5	6	7	82
Total	281	246	2019	1785	60	74	81	1641

B. Device-Cloud Executable Identification

The identification algorithm is run on the firmware from 22 purchased devices. FIRMRES successfully identified the device-cloud executables in the firmware images of the first 20 devices (devices with ID 1 to 20). However, the device-cloud interaction for the remaining two devices (device ID 21 and device ID 22) is handled by `shell` scripts and `php` files. At the current stage, FIRMRES can only deal with binary executables but not scripts.

C. Effectiveness of Message Reconstruction

FIRMRES recovered a total of 281 device-cloud messages from 20 devices, with detailed statistics shown in Table II. To assess whether the reconstructed messages are valid, we forged device-cloud messages sent by a PC and checked the responses of the cloud. The responses such as `"Request OK"`, `"No Permission"` and `"Access Denied"` indicate that the reconstructed message is valid. The responses like `"Bad Request"`, `"Request Not Supported"`, and `"Path Not Exits"` mean the device-cloud messages are invalid. 246 out of 281 messages were verified as valid. The field *Address* for device-cloud message reception is not directly evident in some firmware images, which requires capturing communication traffic and examining it.

Message Field Identification. As detailed in Table II, FIRMRES identified a total of 2019 message fields in 246 identified device-cloud messages. We manually verified the reconstructed messages and confirmed that 1785 of these message fields are required to construct the messages. Thus, the accuracy of field identification is 88.41%. The irrelevant items identified as message fields are usually numeric constants that have no specific meanings. For example, the statement `*(undefined4 *)buf = 0x5353414d` was identified as a taint sink by FIRMRES, making `0x5353414d` to be identified as a message field. In fact, `buf` is an intermediate taint data during analysis. This statement is a register shift operation caused by disassembly errors. The real operation on `buf` is `strcpy(buf, username)`. FIRMRES is unable to identify the device-cloud message fields in the PHP files and the shell script files, which leads to some false negatives. Notably, no message fields were missed in device-cloud executables given that our strategy is to overtaint during dataflow analysis.

Field Semantic Recovery. we collected about 147,414 firmware images from different vendors' official websites and open firmware repositories [3] to build the dataset of the deep learning model. We select both device-cloud executables (about 73%) and non-device-cloud executables (about 27%) to form the dataset. Given that the purpose of training is to recover field semantics, feeding the neural network model with code slices from non-device-cloud executables could improve the accuracy of primitive recognition. We selected 547 executables from those firmware images as samples to generate code slices. Finally, we obtained 30,941 code slices as the dataset to train the deep learning model. We put the obtained code slice into a file. Each code slice is a row of data. We developed a script to add labels by searching for manually-defined keywords about field semantics in each line through regular matching. We define a simple dictionary for each primitive for regular matching of keywords. For instance, Dev-Identifier's keywords include "MAC", "deviceId", "modelId", and so on. Labels generated using only this approach also have errors. We then used the visualization tool Doccano [4] to effectively examine the labels, and correct the label when there is an error. We split the dataset into training, validation, and test sets with a ratio of 7:2:1. Our model underwent 100 epochs of training, consuming nearly 5 hours. The accuracy on the validation set and the test set is 92.23% and 91.74%, respectively.

In building the MFT, not all executables use the `sprintf` function to assemble the partial messages (denoted by `"-"` in Table II). We empirically set the similarity thresholds of the clustering to 0.5, 0.6, and 0.7, and the substrings of the deconstructed message are grouped into 5, 6, and 7 clusters, respectively. In total, we performed semantic recovery on 1785 code slices for the 20 firmware images and used the primitive with maximum probability outputted by the model as its recovered semantics. After a manual examination, we found that there are 1641 message fields whose semantics were correctly predicted. FIRMRES achieves semantic recovery with an accuracy of 91.93%.

D. Identified Vulnerabilities

By manually probing the cloud and verifying the reconstructed messages, we found a total of 14 vulnerabilities in 8 devices, including 1 known vulnerability (by checking the device model and the firmware version) and 13 previously-unknown vulnerabilities. We present partial details of the vulnerable device-cloud interfaces and their impacts in Table III. 10 of these interfaces rely on *Dev-Identifiers* such as MAC addresses, serial numbers and uid for authentication and authorization checks. 2 of them lack *Dev-Secret* and 1 lacks *User-Cred*. It could lead to severe consequences such as privacy breaches, data tampering, or disruption of normal user operations.

TABLE III
SUMMARY OF DISCOVERED VULNERABILITIES

Device ID	Functionality	Partial Detail	Consequence
5	Registrating device to the cloud.	Path: */***/***/registrations* Params: *serialNumber/macAddress/modelNumber/uuid /hardwareVersion/firmwareVersion/manufacturingDate*	It returns a fixed device token, which can be used to upload tampered system information and crash logs to the cloud.
5	Uploading crash logs.	Path: */***/***?uploadType=***** Params: *uploadSubType/firmwareVersion/serialNo/ macAddress/hardwareVersion/uploadType/deviceToken*	Attackers upload fake crash logs to trick users.
2&3	Binding the device to the cloud user.	Method: *** Params: *deviceID/cloudusername/cloudpassword*	Attackers can bind the device to the accounts by sending a fake binding request.
3	Acquiring the shareID list of the device.	Method: *** Params: *deviceID*	ShareID list can be used to obtain the shared information about the device.
17	Checking the availability of the cloud storage service.	Path: *?m=cloud&a=queryServices* Params: *uid*	Privacy information leakage.
17	Uploading crash logs.	Path: *?m=camera&a=crash_report* Params: *uid/version*	After a successful upload, the device crashes and loses its connection.
17	Pushing monitor alert.	Path: *?m=camera_alarm&a=camera_pic_alarm* Params: *uid/alarm_time/lang/img*	Attackers push false alerts to victim users.
18	Obtaining binding information.	Path: */auth/get_bind_params* Params: *userid/mac/sdkver*	Privacy information leakage.
18	Retrieving stored video records.	Path: *app/device/save_video/report* Params: *start time/code/userid/mac/sdkver*	Privacy information leakage.
19	Changing the device ID	Path: */change* Params: *vuid/code/cluster*	Information tampering.
20	Querying the cloud storage services of the device.	Path: */store-server/api/v1/storages/status* Params: *deviceId/channel*	Privacy information leakage.
20	Authenticating the device to the cloud storage server.	Path: */store-server/api/v1/storages/auth* Params: *deviceId*	The cloud returns *access-key* and *secret-key* used to upload videos to the cloud.
20	Querying the videos stored on the cloud.	Path: */store-server/api/v1/storages/files* Params: *deviceId/channel/stream/type/date/begin/end*	The cloud returns video information and download paths for the queried time period.

***: To prevent malicious exploitation, we hide some details of the vulnerabilities.

FIRMRES reported a total of 26 flawed messages in 20 devices, and we confirmed 15 of them. The false positives are mainly caused by (1) customized primitives defined by vendors. For example, a user issues a command to the cloud through the web service front end of a device. This command requires the submission of a verification code previously received from the cloud by the user. The verification code is collected by the device front end and utilized as a field by the device back end to construct the corresponding command message sent to the cloud. In this case, the verification code is *User-Cred*. However, the FIRMRES cannot recover its semantics accurately because very few vendors use this field. This results in the misdetection of the message as lacking *User-Cred*, leading to a false positive alarm. (2) the messages lacking the primitives but containing vendor-specific fields used for message events, such as `eventType` and `pluginId`. They do not belong to the primitives.

Ethics and Responsible Disclosure. We have taken ethical considerations seriously in our research. We only validated the vulnerabilities on our own accounts and our own devices, and we never tried to compromise other users' accounts and devices. We responsibly disclosed the details to the device vendors. For those vulnerabilities that have not yet been patched at the time of this writing, we redacted their vendor names as well as their device models with the symbol "***", in order to avoid negative impacts. We will continue to engage with the vendors to offer help with our best efforts.

E. Performance of FIRMRES

Overall, FIRMRES is fast and efficient in reconstructing device-cloud messages. The shortest time to reconstruct device-cloud messages for a target firmware is only 154 seconds, while the longest is 1472 seconds (about 25 minutes). The time cost primarily depends on the number of executables in the firmware, the number of device-cloud messages, and the number of message fields. On average, the steps of pinpointing device-cloud executables, identifying message fields, recovering field semantics, concatenating message fields, and detecting incorrect forms occupy 37.67%, 43.83%, 3.71%, 9.96%, and 4.81% of the total time cost, respectively. For message field identification, the time is mostly spent on performing the taint analysis because the strategy is to overtaint.

F. Comparison with Other Tools

To the best of our knowledge, FIRMRES is the first tool that automatically reconstructs device-cloud messages from IoT firmware. There are no other solutions that analyze IoT firmware and target IoT cloud backends. As a result, we opt to compare our approach with LEAKSCOPE [40] and IOT-APISCANNER [25]. LEAKSCOPE deals with mobile apps and exposes access control issues in the mobile app's cloud

TABLE IV
COMPARISON OF EXISTING WORKS

	FIRMRES	LEAKSCOPE [40]	IOT-APISCANNER [25]
Inputs	IoT firmware	Mobile App	Mobile IoT App
Target Cloud Platforms	IoT vendor's clouds	AWS, Azure, and FireBase's cloud	IoT platforms
# of Cloud Interfaces	246	32	157
Accuracy of Recovery	87.5%	100%	100%

services. IOT-APISCANNER deals with mobile IoT apps and exposes access control issues in the clouds of IoT platforms.

As shown in Table IV, FIRMRES, LEAKSCOPE, and IOT-APISCANNER take different forms of inputs and target different cloud services. FIRMRES deals with IoT firmware images. Its targets are IoT device vendors' backends. LEAKSCOPE targets the backends of mobile apps, and IOT-APISCANNER targets the backends of the IoT platforms that provide IoT solutions to device manufacturers. LEAKSCOPE and IOT-APISCANNER target cloud platforms with API documentations for vendors. In contrast, FIRMRES focuses on vendors' private cloud services that often lack open API documentations. Among them, FIRMRES's targeted cloud backends are more diverse. As such, FIRMRES can test more cloud interfaces than the other tools. In terms of message reconstruction, IOT-APISCANNER directly inserts complete messages into send functions through dynamic analysis without reconstructing messages. Given that FIRMRES is based on static analysis while the other tools take dynamic approaches, the accuracy of FIRMRES in reconstructing APIs or device-cloud messages is lower than the others.

VI. RELATED WORK

Broken Access Control. In recent years, there has been a growing concern regarding the security of IoT systems [11], [13]. Several related studies have focused on detecting broken access control in cloud services. Zhou et al. [38] and Chen et al. [15], [16] dissected the IoT device binding process, analyzing the security risks associated with this procedure. Some solutions systematically investigated access control security risks in multi-party communication scenarios within the IoT and conducted evaluations using real devices [20], [21], [38], [39]. Research on broken access authorization detection solutions for IoT cloud-mobile applications has also seen significant developments [19], [25]–[27], [37]. KingFisher [37] detects security issues with shared credentials in IoT-mobile communication by identifying shared credentials, tracking their usage, and checking for violations of nine security attributes that the credentials should adhere to. ATester [26] is an access token execution security testing tool based on Android Hooking to detect access token security vulnerabilities in IoT smart home platforms. Yuan et al. [35] discussed authorization security issues in emerging cross-cloud delegation services and implemented a semi-automated tool for modeling cross-cloud delegations on different platforms. It conducts security attribute verification to detect security issues between cross-cloud delegations. Meanwhile, some research [24], [31], [34], [36] conducted an analysis of authorization flaws in the interaction process and proposed enhancement solutions. Although the above works aim to analyze broken access control, they are applicable to particular targets like mobile and IoT cloud backends. However, to the best of our knowledge, there is no existing solution that constructs messages from firmware and tests device-cloud interfaces.

Mobile Application Analysis. Since mobile applications play an important role in IoT systems, many solutions focus on them to study the security issues in IoT systems. Some solutions [14], [22], [28] perform security detections on physical devices by analyzing their companion apps. These solutions need physical devices to perform detection. Wang et al. [32] revealed the sharing of vulnerable components across smart home IoT devices by cross-analysis of the mobile companion apps. It requires prior domain knowledge about other devices and existing vulnerabilities. Schmidt et al. [29] combined value set analysis with data flow analysis to analyze mobile apps, revealing how the applications communicate with IoT devices and remote cloud-based backends. The other works [30], [41] focus on BLE issues by analyzing the IoT companion apps. However, when device-based interactions are not involved, app-cloud issues are not specific to IoT. Mobile app analysis is more concerned about securing communication on the application side, making it less intuitive to detect communication between devices and the cloud. Without the challenges in firmware analysis, performing static analysis or dynamic testing based on mobile apps is relatively easy.

VII. CONCLUSION

In this paper, we have presented the tool FIRMRES that reconstructs device-cloud messages through static firmware analysis, and have demonstrated that it is constructive in facilitating the assessment of access control in IoT device-cloud interfaces. We have also presented a set of new techniques that make FIRMRES highly practical. By conducting evaluations in 22 real-world devices, FIRMRES successfully identified 13 unknown vulnerabilities that could lead to serious consequences, bringing such hidden and critical device-cloud interfaces to the spotlight.

ACKNOWLEDGMENTS

We are grateful to our shepherd Le Guan and the anonymous reviewers for their insightful comments. This work was partially supported by Distinguished Youth Fund Project of Hunan Province, China (Grant No. 2022JJ10018). Jiongyi Chen was supported by the Natural Science Foundation of China (Grant No. 62302508) and the Natural Science Foundation of Hunan Province, China (Grant No. 2022JJ40553).

REFERENCES

[1] "otorio: Rce on inhand networks cloud-managed routers,". https://www.otorio.com/blog/cloud-to-firmware-rce-on\ -inhand-networks-cloud-managed-routers/,. Accessed on 5/22/2023.

[2] "class varnodeast". https://ghidra.re/ghidra_docs/api/ghidra/program/model/pcode/Varnode.html,. Accessed on 12/21/2021.

[3] "firmware-dataset.". https://github.com/WUSTL-CSPL/Firmware-Dataset,. Accessed on 11/23/2023.

[4] "github - doccano/doccano: Open source annotation tool for machine learning practitioners,". https://github.com/doccano/doccano,. Accessed on 8/10/2023.

[5] "github - nationalsecurityagency/ghidra: Ghidra is a software reverse engineering (sre) framework,". https://github.com/NationalSecurityAgency/ghidra,. Accessed on 10/17/2022.

[6] "introduction to snmp". https://datadoghq.dev/integrations-core/tutorials/snmp/introduction/,. Accessed on 12/11/2023.

[7] "p-code reference manual". https://spinsel.dev/assets/2020-06-17-ghidra-brainfuck-processor-1/ghidra_docs/language_spec/html/pcoderef.html,. Accessed on 9/27/2017.

[8] "publish-subscribe pattem.". https://en.wikipedia.org/wiki/Publish-subscribe_pattem,. Accessed on 12/11/2023.

[9] "shadon: Shodan search engine,". https://www.shodan.io/,. Accessed on 10/16/2023.

[10] "sierra wireless vulnerabilities "on air": Airvantage cloud and airlink routers". https://www.otorio.com/blog/sierra-\wireless-airvantage-and-\airlink-router-vulnerabilities/,. Accessed on 6/5/2023.

[11] ALRAWI, O., LEVER, C., ANTONAKAKIS, M., AND MONROSE, F. Sok: Security evaluation of home-based iot deployments. In *2019 IEEE symposium on security and privacy (sp)* (2019), IEEE, pp. 1362–1380.

[12] CABALLERO, J., POOSANKAM, P., KREIBICH, C., AND SONG, D. Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering. In *Proceedings of the 16th ACM conference on Computer and communications security* (2009), pp. 621–634.

[13] CHEN, F., LUO, D., XIANG, T., CHEN, P., FAN, J., AND TRUONG, H.-L. Iot cloud security review: A case study approach using emerging consumer-oriented applications. *ACM Computing Surveys 54* (04 2021), 1–36.

[14] CHEN, J., DIAO, W., ZHAO, Q., ZUO, C., LIN, Z., WANG, X., LAU, W. C., SUN, M., YANG, R., AND ZHANG, K. Iotfuzzer: Discovering memory corruptions in iot through app-based fuzzing. In *NDSS* (2018).

[15] CHEN, J., SUN, M., AND ZHANG, K. Security analysis of device binding for ip-based iot devices. In *2019 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)* (2019), IEEE, pp. 900–905.

[16] CHEN, J., ZUO, C., DIAO, W., DONG, S., ZHAO, Q., SUN, M., LIN, Z., ZHANG, Y., AND ZHANG, K. Your iots are (not) mine: On the remote binding between iot devices and users. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (2019), IEEE, pp. 222–233.

[17] CUI, W., PEINADO, M., CHEN, K., WANG, H. J., AND IRUN-BRIZ, L. Tupni: Automatic reverse engineering of input formats. In *Proceedings of the 15th ACM conference on Computer and communications security* (2008), pp. 391–402.

[18] DEVLIN, J., CHANG, M.-W., LEE, K., AND TOUTANOVA, K. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[19] FERNANDES, E., RAHMATI, A., JUNG, J., AND PRAKASH, A. Decoupled-ifttt: Constraining privilege in trigger-action platforms for the internet of things. *arXiv preprint arXiv:1707.00405* (2017).

[20] JIA, Y., XING, L., MAO, Y., ZHAO, D., WANG, X., ZHAO, S., AND ZHANG, Y. Burglars' iot paradise: Understanding and mitigating security risks of general messaging protocols on iot clouds. In *2020 IEEE Symposium on Security and Privacy (SP)* (2020), IEEE, pp. 465–481.

[21] JIA, Y., YUAN, B., XING, L., ZHAO, D., ZHANG, Y., WANG, X., LIU, Y., ZHENG, K., CRNJAK, P., ZHANG, Y., ET AL. Who's in control? on security risks of disjointed iot device management channels. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (2021), pp. 1289–1305.

[22] JUNIOR, D. M., MELO, L., LU, H., D'AMORIM, M., AND PRAKASH, A. A study of vulnerability analysis of popular smart devices through their companion apps. In *2019 IEEE Security and Privacy Workshops (SPW)* (2019), IEEE, pp. 181–186.

[23] KIM, Y. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882* (2014).

[24] LI, X., CHEN, Y., LIN, Z., WANG, X., AND CHEN, J. H. Automatic policy generation for {Inter-Service} access control of microservices. In *30th USENIX Security Symposium (USENIX Security 21)* (2021), pp. 3971–3988.

[25] LI, Y., YANG, Y., YU, X., YANG, T., DONG, L., AND WANG, W. Iot-apiscanner: Detecting api unauthorized access vulnerabilities of iot platform. In *2020 29th International Conference on Computer Communications and Networks (ICCCN)* (2020), IEEE, pp. 1–5.

[26] LIU, C., YANG, Y., ZHANG, Y., AND ZHANG, Y. Unfettered access tokens: Discovering security flaws of the access token in smart home platforms. In *ICC 2022-IEEE International Conference on Communications* (2022), IEEE, pp. 5391–5396.

[27] MAHADEWA, K., ZHANG, Y., BAI, G., BU, L., ZUO, Z., FERNANDO, D., LIANG, Z., AND DONG, J. S. Identifying privacy weaknesses from multi-party trigger-action integration platforms. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (2021), pp. 2–15.

[28] REDINI, N., CONTINELLA, A., DAS, D., DE PASQUALE, G., SPAHN, N., MACHIRY, A., BIANCHI, A., KRUEGEL, C., AND VIGNA, G. Diane: Identifying fuzzing triggers in apps to generate under-constrained inputs for iot devices. In *2021 IEEE Symposium on Security and Privacy (SP)* (2021), IEEE, pp. 484–500.

[29] SCHMIDT, D., TAGLIARO, C., BORGOLTE, K., AND LINDORFER, M. Iotflow: Inferring iot device behavior at scale through static mobile companion app analysis. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security* (2023), pp. 681–695.

[30] SIVAKUMARAN, P., ZUO, C., LIN, Z., AND BLASCO, J. Uncovering vulnerabilities of bluetooth low energy iot from companion mobile apps with ble-guuide. In *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security* (2023), pp. 1004–1015.

[31] TIAN, Y., ZHANG, N., LIN, Y.-H., WANG, X., UR, B., GUO, X., AND TAGUE, P. {SmartAuth}:{User-Centered} authorization for the internet of things. In *26th USENIX Security Symposium (USENIX Security 17)* (2017), pp. 361–378.

[32] WANG, X., SUN, Y., NANDA, S., AND WANG, X. Looking from the mirror: Evaluating {IoT} device security through mobile companion apps. In *28th USENIX security symposium (USENIX security 19)* (2019), pp. 1151–1167.

[33] WRIGHT, C., MOEGLEIN, W. A., BAGCHI, S., KULKARNI, M., AND CLEMENTS, A. A. Challenges in firmware re-hosting, emulation, and analysis. *ACM Computing Surveys (CSUR) 54*, 1 (2021), 1–36.

[34] YAHYAZADEH, M., PODDER, P., HOQUE, E., AND CHOWDHURY, O. Expat: Expectation-based policy analysis and enforcement for appified smart-home platforms. In *Proceedings of the 24th ACM symposium on access control models and technologies* (2019), pp. 61–72.

[35] YUAN, B., JIA, Y., XING, L., ZHAO, D., WANG, X., AND ZHANG, Y. Shattered chain of trust: Understanding security risks in {Cross-Cloud}{IoT} access delegation. In *29th USENIX security symposium (USENIX security 20)* (2020), pp. 1183–1200.

[36] YUAN, B., WU, Y., YANG, M., XING, L., WANG, X., ZOU, D., AND JIN, H. Smartpatch: Verifying the authenticity of the trigger-event in the iot platform. *IEEE Transactions on Dependable and Secure Computing 20*, 2 (2022), 1656–1674.

[37] ZHANG, Y., MA, S., LI, J., GU, D., AND BERTINO, E. Kingfisher: Unveiling insecurely used credentials in iot-to-mobile communications. In *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)* (2022), IEEE, pp. 488–500.

[38] ZHOU, W., JIA, Y., YAO, Y., ZHU, L., GUAN, L., MAO, Y., LIU, P., AND ZHANG, Y. Discovering and understanding the security hazards in the interactions between {IoT} devices, mobile apps, and clouds on smart home platforms. In *28th USENIX security symposium (USENIX security 19)* (2019), pp. 1133–1150.

[39] ZHOU, X., GUAN, J., XING, L., AND QIAN, Z. Perils and mitigation of security risks of cooperation in mobile-as-a-gateway iot. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (2022), pp. 3285–3299.

[40] ZUO, C., LIN, Z., AND ZHANG, Y. Why does your data leak? uncovering the data leakage in cloud from mobile apps. In *2019 IEEE Symposium on Security and Privacy (SP)* (2019), IEEE, pp. 1296–1310.

[41] ZUO, C., WEN, H., LIN, Z., AND ZHANG, Y. Automatic fingerprinting of vulnerable ble iot devices with static uuids from mobile apps. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (2019), pp. 1469–1483.