On Manually Reverse Engineering Communication Protocols of Linux-Based IoT Systems

Kaizheng Liu, Ming Yang, Zhen Ling, *Member, IEEE*, Huaiyu Yan, Yue Zhang, Xinwen Fu, *Senior Member, IEEE*, and Wei Zhao, *Fellow, IEEE*

***Abstract*—IoT security and privacy has raised grave concerns. Efforts have been made to design tools to identify and understand vulnerabilities of IoT systems. Most of the existing protocol security analysis techniques rely on a well understanding of the underlying communication protocols. In this article, we systematically present the first manual reverse engineering framework for discovering communication protocols of embedded Linux-based IoT systems. We have successfully applied our framework to reverse engineer a number of IoT systems. As an example, we present a detailed use of the framework reverse engineering the WeMo smart plug communication protocol by extracting the firmware from the flash, performing static and dynamic analysis of the firmware, and analyzing network traffic. The discovered protocol exposes severe design flaws that allow attackers to control or deny the service of victim plugs. Our manual reverse engineering framework is generic and can be applied to both read-only and writable embedded Linux filesystems.**

***Index Terms*—Communication protocols, firmware, IoT system, reverse engineering.**

I. Introduction

SECURITY of IoT products has received increasing scrutiny as IoT is being pervasively deployed [1]–[5]. For example, smart plugs and routers may be fully controlled by buffer overflow or command injection attacks [6]–[8]. Security vulnerabilities also exist in popular IoT platforms, such as AWS IoT [3], [5].

Efforts have been made to design tools to identify and understand vulnerabilities of IoT systems. For example, Chen *et al.* [9] proposed an automatic fuzzing framework to find the memory corruption vulnerabilities caused by the software and firmware of IoT devices. Given a well-formed protocol, formal and heuristic methods could be used to study security and identify the vulnerabilities of the protocol [10]–[14]. For example, Kim *et al.* [10] used formal symbolic modeling to automatically analyze the frequently used IoT protocols, such as CoAP and MQTT. Only when these protocols have been formally verified (mathematically proved) could they be considered as secure. However, the challenge of automatic protocol verification relies on a well understood protocol.

In this article, we propose a framework of manually reverse engineering communication protocols of embedded Linux-based IoT systems so that automation techniques can be applied over the discovered protocols for vulnerability discovery and security analysis. We focused on the embedded Linux-based IoT system given its popularity. We find most IoT devices (more than 71%) are installed with Linux, according to the Eclipse IoT developer survey [15]. Our framework adopts network traffic analysis and static analysis and dynamic analysis of the app and device firmware to understand specific details, such as fields of the communication. Our manual reverse engineering framework works as follows.

1) Obtaining the app and firmware of the device.
2) Collecting network traffic generated by the device and app with testbeds.
3) Defeating traffic protection by using the man-in-the-middle (MITM) proxy, static analysis, and dynamic debugging to defeat traffic encryption and obfuscation.
4) Discovering the communication protocol through traffic analysis, static analysis, and dynamic analysis of the app and firmware.

We have applied our framework and reverse engineered a number of IoT systems, including smart plugs, IP cameras, and air quality monitoring sensors. As an example, this article presents a detailed case study of the popular WeMo smart plug from Belkin. The plug system involves three parts: 1) smart plugs; 2) smartphones; and 3) two cloud servers. A smartphone can communicate with a smart plug via the cloud servers. The cloud servers distribute keys to the smartphone and smart plug,

Manuscript received July 24, 2020; revised September 18, 2020; accepted October 21, 2020. Date of publication November 5, 2020; date of current version April 7, 2021. This work was supported in part by the National Key Research and Development Program of China under Grant 2018YFB0803400, Grant 2018YFB2100300, and Grant 2017YFB1003000; in part by the U.S. National Science Foundation under Award 1643835, Award 1931871, and Award 1915780; in part by the U.S. Department of Energy under Award DE-EE0009152; in part by the National Natural Science Foundation of China under Grant 62072103, Grant 62072102, Grant 62072098, Grant 62022024, Grant 61972088, Grant 61702097, Grant 61632008, and Grant 61532013; in part by the Jiangsu Provincial Natural Science Foundation for Excellent Young Scholars under Grant BK20190060; in part by the Jiangsu Provincial Key Laboratory of Network and Information Security under Grant BM2003201; in part by the Key Laboratory of Computer Network and Information Integration of Ministry of Education of China under Grant 93K-9; and in part by the Collaborative Innovation Center of Novel Software Technology and Industrialization. *(Corresponding authors: Zhen Ling; Ming Yang.)*

Kaizheng Liu, Ming Yang, Zhen Ling, and Huaiyu Yan are with the School of Computer Science and Engineering, Southeast University, Nanjing 211189, China (e-mail: kzliu@seu.edu.cn; yangming2002@seu.edu.cn; zhenling@seu.edu.cn; huaiyu_yan@seu.edu.cn).

Yue Zhang is with the College of Information Science and Technology, Jinan University, Guangzhou 510632, China (e-mail: zyueinfosec@gmail.com).

Xinwen Fu is with the Department of Computer Science, University of Massachusetts Lowell, Lowell, MA 01854 USA (e-mail: xinwenfu@cs.uml.edu).

Wei Zhao is with the American University of Sharjah, Sharjah, UAE (e-mail: weizhao@aus.edu).

Digital Object Identifier 10.1109/JIOT.2020.3036232

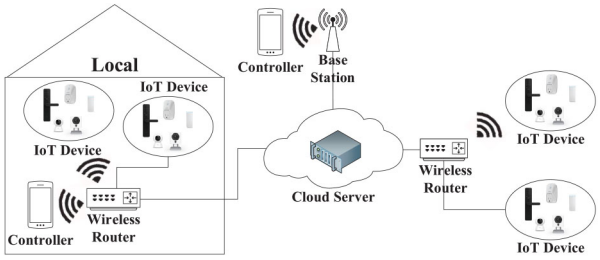


Fig. 1. Simplified architecture of an IoT system.

and authenticate them based on the distributed keys. Once the communication protocol of the smart plug is discovered, we are able to identify a serious design flaw that allows two attacks: 1) a malicious software smartphone bot could be used to control victim plugs and 2) a fake smart plug can pretend to be a real one and kick the real one offline. We also successfully demonstrated reverse engineering of a Xiongmai camera and a Haier camera and won us an Award of Excellence at GeekPwn 2020.

Contribution: Major contributions of this article can be summarized as follows.

1) We are the first to systematically propose a framework to manually reverse engineer communication protocols of IoT systems.
2) We have applied this framework to successfully reverse engineer a number of IoT systems. As an example, this article presents a complete protocol analysis of the WeMo smart plug and identifies severe design flaws that allow attackers to control victim plugs and deny the service of victim plugs. We also briefly discuss how we apply the framework to a few other IoT systems.
3) Our communication protocol, reverse engineering framework, is generic and can be applied to both read-only and writable Linux filesystems. We collected the firmware of 514 popular IoT devices on the market and showed that our framework is applicable to them.

Road Map: The remainder of this article is organized as follows. In Section II, we briefly introduce background knowledge. In Section III, we present our communication protocol reverse engineering framework. In Section IV, we present a case study of the WeMo smart plug using the proposed framework. In Section V, we discuss the generality and limitations of our framework. Related work is presented in Section VI and we conclude this article in Section VII.

II. Background

In this section, we present a brief introduction to the architecture of an IoT system and terms used in this article.

A. Architecture of IoT System

Fig. 1 shows a typical IoT communication system based on our experiments and previous researches [3], [13], [14], [16]. The system consists of three components: 1) an IoT device; 2) a controller; and 3) a cloud server. The IoT device implements specific functionalities, such as medical monitoring and electrical control. The controller, such as a smartphone app, is used to control the IoT device. The cloud server is used to relay messages between the controller and the IoT device. The cloud server may provide other services, including device management, data storage, and analysis. For a smart plug system, the smart plug is the IoT device while the smartphone app is the controller. When the controller and IoT device are located in the same network, the smart plug's official app could be used to directly communicate and control the plug through WiFi. If the controller and IoT device are in different networks, a cloud server could be adopted to transmit the message between the controller and the IoT device so as to traverse the network address translation (NAT).

B. Communication Protocols and Terms

An IoT communication system may realize complicated communication protocols and various functionalities. We have identified four common phases of an IoT communication protocol, including paring, binding, authentication, and controlling [13], [17], which are crucial for the overall system security.

1) *Pairing:* To bootstrap and configure an IoT device, a user often needs to connect a controller (e.g., an app on a smartphone) to the IoT device via various communication venues. For example, the IoT device can work as a WiFi access point (AP) so that the controller can connect to it. The controller can also connect to the IoT device via Bluetooth. We denote this connecting process as pairing. This is relevant to security since the pairing process may be under malicious sniffing and anyone may get access to the IoT device, particularly in the cases that the device is deployed in public.
2) *Binding:* When pairing is completed, a binding mechanism is often employed so that the cloud server can associate the controller and IoT device, and relay messages between them.
3) *Authentication:* The controller, device, and cloud server often need to authenticate each other to defeat various threats and abuses.
4) *Controlling:* After authentication, the controller can take control of the IoT device via a cloud server or a local network.

III. Framework of Manually Reverse Engineering IoT Communication Protocols

In this section, we will present the assumption about capabilities of security analysts, and our manual reverse engineering framework.

A. Capabilities of Security Analyst

We adopt the term "security analyst" to refer to those who would use our framework to reverse engineer third-party IoT products. We make the following assumptions about the capabilities of the security analyst.

1) To the best of our knowledge, most IoT vendors provide both Android and iOS apps. The communication protocol of both the Android app and iOS app is the same, for their functionality is similar. The analyst can

analyze either the Android app or the iOS app to extract the communication protocol between the controller and the cloud server. Since there are more existing reverse engineering tools for the Android apps than those for the iOS apps [18]–[20], we choose the Android app as an example of the controller in this article.

2) We focus on IoT devices that use the popular and open-source-embedded Linux-based operation system (OS).

B. Overview

Fig. 2 illustrates the workflow of our manual reverse engineering framework: obtaining the app and device firmware, collecting network traffic, defeating traffic protection, and discovering the communication protocol.

1) *Obtaining the App and Device Firmware:* The app is often free and can be downloaded from Google Play (or Apple App Store). The device firmware may be obtained from the manufacturer's website, over-the-air (OTA) update process [21] (i.e., firmware update process), or reading the flash chip as discussed later in this section. The first two approaches are straightforward. However, they may not be always available.
2) *Collecting Network Traffic:* In this step, we particularly want to collect network traffic and understand security related phases of the IoT communication protocols. During the pairing process, the IoT device may work as a WiFi AP and the controller connects to this AP. A sniffer is needed to dump the pairing traffic. After pairing, the device and controller will connect to the Internet through a router/switch/AP. For simplicity, we will use AP to refer to router/switch/AP. To intercept the traffic after pairing, we set up our own APs. The controller and IoT device connect to our APs and communicate with each other through either the local network or Internet. The traffic of interest can be collected from these APs.
3) *Defeating Traffic Protection:* Some vendors may adopt TLS/SSL encryption or obfuscation to protect the communication. The analyst can defeat the TLS/SSL encryption with an MITM proxy. Obfuscation algorithms can be disclosed through static analysis and dynamic debugging of the app and firmware.
4) *Discovering the Communication Protocol:* Through the combination of traffic analysis, static analysis, and dynamic analysis of the app and firmware, the communication protocol can be discovered. Based on the discovered communication protocol, the analyst may use either heuristic methods or formal methods to find vulnerabilities of the protocol. In this article, we use heuristic methods to demonstrate the feasibility of the reverse engineering approach.

C. Obtaining the App and Device Firmware

The app is often free and can be downloaded from Google Play. However, it can be a challenge to extract the firmware from the flash chip, which often involves the following steps. First, we take apart the physical device and identify the device's flash chip model (e.g., NOR flash and NAND flash) and packaging type [e.g., small-outline package (SOP), quad flat package (QFP), and ball grid array (BGA)]. The information can be found on the surface of the chip or the case of the IoT device. With such information, we can determine which type of surface-mount packaging is applied to the device's flash chip accordingly. For example, if the flash uses SOP that often exposes the flash pins, we can connect Bus Pirate [22] to the corresponding pins via a test clip and an adapter in order to read the firmware image from the flash. However, with a particular packaging technology, for example, BGA, a flash chip may not expose its pins. In such a case, we may desolder the flash chip by using a surface mount technology (SMT) rework station [23]. After obtaining the flash chip, a flash engineering programmer such as StarProg-F [24] may be used to read the firmware image from the flash.

D. Collecting Network Traffic

In the pairing phase, some IoT devices may work as an AP so that the controller can connect to it and deliver pairing information. To collect the network traffic in the pairing phase, a wireless network card supporting the monitor mode can be used as a sniffer to dump the WiFi traffic. Besides, we find there are another five methods can be used to transmit pairing information, i.e., SmartConfig, QRcode, Bluetooth, voice, and Ethernet cable connection. In these cases, the pairing information can be analyzed by dynamically hooking the functions used for encoding the pairing information in the controller app. We will discuss this in Section III-E2.

In order to dump the network traffic during binding, authentication, and controlling phases, we build an AP equipped with wireless network cards and Ethernet cards. To build our own AP or a wireless router, we install Hostapd [25] on a computer with a wireless network card supporting the AP mode. The computer is also equipped with an Ethernet card connecting to the Internet. Some IoT devices only support Ethernet. In such a case, we equip the computer with a second Ethernet card connecting to such an IoT device. In this way, the passing traffic can be intercepted by the computer.

E. Defeating Traffic Protection

We now discuss how to defeat encryption and obfuscation that are used to protect traffic from the app and IoT device.

1) Encryption: Network traffic can be encrypted by TLS/SSL. To decrypt the traffic, a MITM transparent proxy is installed in front of the smartphone (i.e., controller) or IoT device. The proxy is used to relay or manipulate the traffic between the device and remote server, or the traffic between the smartphone and remote server. With proper configuration, the MITM proxy can decrypt the passing traffic. Specifically, we use an open-source tool "mitmproxy" [26] as our MITM proxy.

We now show how to replace the target root certificate issued by a trusted certificate authority (CA) or a self-signed private root certificate with the forged root certificate on a controller. Take Android as an example. From our empirical analysis, the certificate can be located in three places as follows.

i) The trusted CA certificate is stored in "/system/etc/security" as an individual file [27]. In this case, we can just add the forged root certificate to the Android system.

ii) The private root certificate can be packaged as a file in an app. In this case, we use APKTool [28] to unpack the APK package and replace the original certificate with the forged root certificate. We then recompile and sign the APK [29].

iii) The private root certificate can also be hard coded in the format of a string in the app code. In this case, we decompile the original app into smali code, identify and replace the original hard-coded root certificate, and finally generate a new app.

We now discuss how to replace the original root certificate with the forged root certificate on an IoT device. This case is more complicated.

i) We first search the root certificate in the filesystem of the obtained firmware. The original certificate can be a standalone file or hard coded in a binary file. The certificate often has a set of features. For example, if the certificate is encoded in privacy-enhanced mail (PEM) [30] format, it contains a header ("-----BEGIN CERTIFICATE-----"). Therefore, we can locate the certificate by searching the header.

ii) Once we locate the root certificate, we need to identify which type of filesystem is used by the firmware so that a specific replacement method can be applied. An open-source tool named Binwalk [31] is introduced to identify the filesystem type, either a writable filesystem, such as *JFFS2* and *UBIFS* or a read-only filesystem, such as *SquashFS* and *CramFS*.

For a read-only filesystem, the replacement cannot be made directly since modification is not allowed. We can reflash a customized firmware with the forged root certificate into the device. We may need to generate the cyclical redundancy check (CRC) and append it to the customized firmware to pass the chip's integrity check. For a writable filesystem, there are two ways to replace the original certificate as follows.

i) If we can get into the console of the IoT device system, for example, by using universal asynchronous receiver–transmitter (UART), and locate file transfer tools like a ftp client, we can replace the original certificate directly via file transfer tools.

ii) We can replace the original certificate directly by mounting the writable filesystem segmented from the firmware onto a Linux computer. We then repackage a new firmware with the modified filesystem and flash the new firmware into the device.

There are two ways to flash a modified firmware with the forged root certificate into an IoT device.

i) We can flash the firmware back using Bus Pirate or a flash engineering programmer. If the flash chip is desoldered for reading the firmware [23], we need to resolder it back to the circuit board.

ii) We can also flash the firmware back to the chip via the firmware upgrading interface like the OTA interface.

2) Obfuscation: An IoT system may protect its traffic by obfuscation. Traffic obfuscation is used to make communications more complicated. Unlike encryption, obfuscation does not require a key to encrypt or decrypt the traffic [32]. Static analysis and dynamic analysis may be adopted to counter traffic obfuscation.

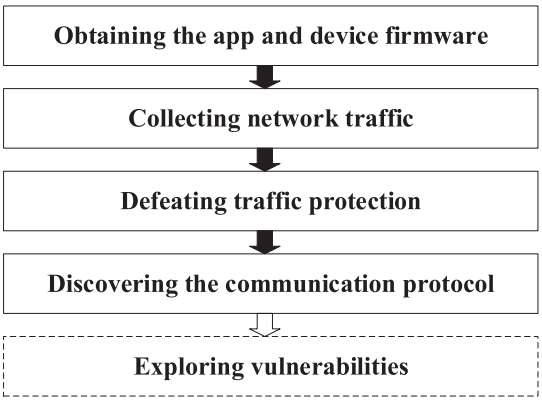


Fig. 2. Workflow of communication protocol reverse engineering framework

```
a=0;// # virtual methods
a=0;// .method protected onCreate(Landroid/os/Bundle;)V
a=0;//     invoke-static {}, Landroid/os/Debug;->
    waitForDebugger()V
```

Listing 1. Add waitForDebugger function to entry activity of app.

To deobfuscate traffic from a controller, for example, an Android app, we first need to understand how the obfuscated traffic is generated and then write a deobfuscation algorithm. To this end, we first need to check if the app is packed. For a packed app, we can unpack it [33]–[35]. Then, we can extract the smali code using Apktool. We analyze the workflow of the traffic obfuscation algorithm by reading the extracted smali code. We may use Smali2Java [36] to decompile the smali code into the Java format for easy understanding. We can also dynamically debug the smali code by using Android SDK and Android Studio [37] as follows.

i) We add a new field "android:debuggable = true" in the tag of Android manifest file "application" to enable debugging.

ii) We locate the function of the entry activity, "onCreate," and add a line of smali code at the beginning of this function as shown in List 1 to make the app wait for the debug signal after being started.

iii) We repackage the modified APK and install it in the smartphone.

iv) Now, once we start the app, we can use Android Studio to add break points and monitor the functions of interest.

However, the method above will fail when Java Native Interface (JNI) is applied. To address this issue, we introduce IDA pro [38], a multiplatform tool that offers both static and dynamic analysis functionalities.

i) We first enable USB debugging on the tested smartphone.

ii) We copy the binary file of IDA pro, "android_server," to the smartphone and run it via the Android Debug Bridge (adb) [39].

iii) We map a port on the computer to a port on the smartphone so that they can communicate with each other.

iv) We run the app in debug mode, and start the IDA pro client on the computer. The smartphone then forwards the debug log to the computer via the configured port.

Dynamic hooking tools (e.g., "Xposed" [40] and "Frida" [41]) can also be used to dynamically analyze the obfuscation algorithm. We can first generate the function call graph with FlowDroid [42] and IDA pro. Then, we statically analyze the function call graph to locate the potential functions related to network communication APIs that may be used to obfuscate the traffic. Next, we use these dynamic analyzing tools to hook the functions to record the arguments and return values. In this way, we can heuristically locate the obfuscation function by comparing the obfuscation traffic and the recorded log. Finally, static analysis can be used to extract the obfuscation algorithm in the obfuscation function. The hooking methods are also used to analyze the pairing information and discover the communication protocol demonstrated in Section III-F.

We now discuss *how to deobfuscate traffic from an IoT device*. We need to identify the algorithm that obfuscates the messages and write a deobfuscation algorithm. The obfuscation algorithm is usually stored in a particular binary file. Therefore, the first step is to identify this file in the firmware. We compare the information from the analysis of dumped network traffic with the IoT device's runtime system log. If a match is discovered, the file can then be identified. To obtain the log, we first need to obtain the console of the IoT device system. If we can locate the UART port on the board of IoT device, we can connect it to the debugging computer using a UART-to-USB bridge with a correct baud rate. User authentication may be required to login into an IoT device system via UART. The login passwords are often stored in "/etc/shadow" file or "/etc/passwd" file. In these cases, there are two solutions as follows.

i) We can try to extract the password hashes from the flash and crack them.
ii) If password racking fails because of long and complex passwords, we can use the repacking method introduced in Section III-E1 to modify the files and remove the login password so as to bypass the login authentication.

Otherwise, we can embed a backdoor, such as telnet into the IoT device firmware and update the device with the new firmware. A telnet app that is often hidden in an IoT device maybe for the purpose of debugging by the manufacturer and can be utilized too. We can then log in the IoT device system through the backdoor from the debugging computer. The log can then be shown in the console of the computer after the IoT device starts. For example, we are often interested in the design flaws in authentication of the controller and IoT device. Hence, we perform the authentication phase repeatedly and compare the ports used in each process in the runtime system log with the port of intercepted obfuscated traffic. If the ports match, we find the target binary file. Afterward, we extract the binary file with Binwalk from the firmware of the IoT device as discussed in Section III-C.

Once obtaining the binary file, in order to obtain the obfuscation algorithm, we can analyze it as follows.

i) We can perform static analysis to disassemble the binary file with IDA pro.
ii) We can also dynamically analyze it on the IoT device using binary instrumentation [43] by inserting additional code into the executable binary file to observe or modify the behavior of the binary file. Binary instrumentation allows us to trace functions of interest, and follow the workflow of the inputs and outputs. To use binary instrumentation, we need to modify the firmware with the method proposed in Section III-E1.
iii) We can also use the GDB client and GDBserver [44] to remotely debug the binary program of the IoT device from a computer. We first need to cross compile the GDBserver and embed the GDBserver into the firmware of the target IoT device and run the GDB client in our debugging computer. By configuring the IP address and port of the GDBserver, we can use the GDB client to dynamically debug the target binary file and identify the traffic obfuscation algorithm.

F. Discovering the Communication Protocols

Through traffic analysis, we may understand the basics of the communication protocols. However, there are some cryptographic fields and obfuscated fields that should be further analyzed. For cryptographic fields, we perform the following procedure to understand them.

1) We may measure the entropy of the bytes of the traffic to determine whether the command or data are created with cryptographic operations, such as encryption and hash. High entropy beyond a threshold indicates the data are encrypted or hashed.
2) We may also search cryptographic APIs within the firmware to determine if encryption is used and also identify cryptographic functions that are used. At the controller side, the developers may encrypt or hash the application layer data using cryptographic APIs of Android SDK or C/C++ libraries. We can use dynamic hooking tools introduced in Section III-E2 to analyze the frequently used cryptographic APIs [45]. Once a specific cryptographic function is called, the information of this function is recorded. Therefore, we know which function is used. At the device side, we can employ static data flow analysis to identify a cryptographic function [46].
3) Once we locate the target cryptographic function, we can obtain the original command or data and the key for the cryptographic function by dynamically debugging the binary file and analyzing the inputs of the target cryptographic function with the method introduced in Section III-E. Specifically, we can use the "Xposed" and "Frida" at the controller side and use the GDB debugging tool at the device side, respectively. For the obfuscated fields, we can use the deobfuscation methods for countering traffic obfuscation introduced in Section III-E2 to deobfuscate these fields.

G. Exploring Vulnerabilities

After obtaining a well-discovered protocol using the framework, we can employ heuristic methods or formal methods to perform security analysis of the discovered communication protocol. The security analyst may focus on the four

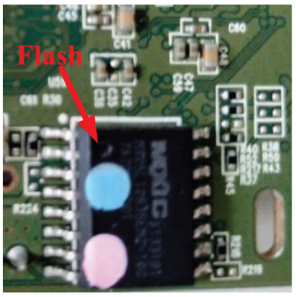


Fig. 3. Flash of WeMo plug.

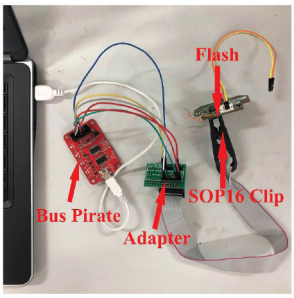


Fig. 4. Reading flash by bus pirate.

phases of the communication protocol (i.e., pairing, binding, authentication, and controlling) introduced in Section II while performing vulnerability assessment of IoT systems. According to our research [13], [14] and related work [3], we find that these four phases are often vulnerable and can cause severe privacy and security issues. For example, in [13], we find every online Edimax camera can be remotely controlled by attackers using the vulnerabilities in binding, authentication, and controlling phases of its protocol.

IV. Case Study: Smart Plugs From Belkin WeMo

The manual reverse engineering framework introduced in Section III-C is the result of our reverse engineering of a number IoT devices, including our previous research [13], [14]. In this section, we present a case study of reverse engineering the WeMo smart plug using the framework and the discovered communication protocols. We will also introduce novel attacks against the plug based on the discovered protocols.

A. Reverse Engineering WeMo Smart Plug

We present the workflow of reverse engineering the WeMo smart plug.

1) Obtaining the App and Device Firmware: The official app of the smart plug is free to download while the firmware is publicly unavailable. The flash chip of the smart plug is shown in Fig. 3 and it is packaged with SOP. As shown in Fig. 4, we can use Bus Pirate to read the firmware from the flash chip with an SOP16 clip and an adapter, which are shown in Fig. 5.

2) Collecting Network Traffic: A testbed is deployed to eavesdrop on the network traffic of interest. As shown in Fig. 6, during the pairing phase, the smart plug works as an AP and we collect the pairing traffic with a sniffer. We

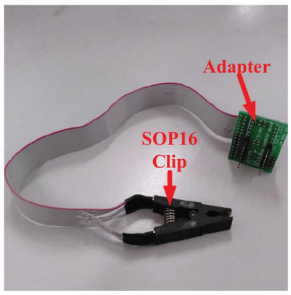


Fig. 5. Adapter and SOP16 clip.

Fig. 6. Pairing traffic collection.

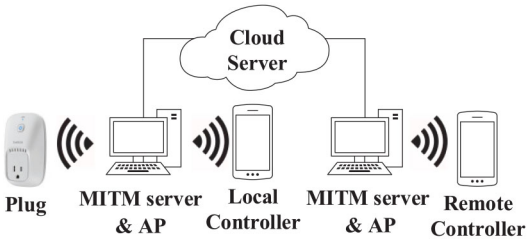


Fig. 7. NAT traffic collection.

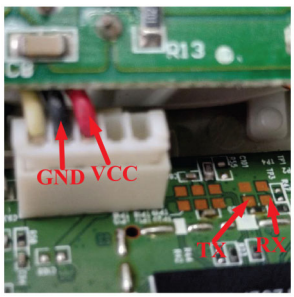


Fig. 8. UART of WeMo plug.

intercept the traffic between the smartphone, smart plug and cloud server by introducing two APs, as shown in Fig. 7.

3) Defeating Traffic Protection: The primary challenge of decrypting encrypted traffic is to replace the original certificate of the firmware and controller app with our forged one.

i) We first replace the certificate of the smartphone. We find that the original certificate is stored in "/system/etc/security." Therefore, on the smartphone, we can download the forged root certificate generated by the MITM proxy through a Web browser and Android will prompt us to install the certificate.

ii) We then replace the original CA certificate in the firmware of the smart plug. We find a UART port on

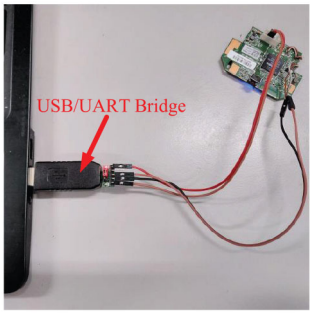


Fig. 9. Open a console of WeMo plug system by UART.

the chip as shown in Fig. 8, where the UART port has four pins, including TX, RX, GND, and VCC. We use a UART-to-USB bridge to open a console of the smart plug's embedded Linux system, as shown in Fig. 9, and find a ftp client in the system. We put the forged root certificate on a ftp server and download it to the plug system through the discovered ftp client, so as to replace the original CA certificate. The forged certificate will be preserved in the device even after the device reboots. This shows the plug's filesystem is writable. By using Binwalk, we find that the firmware actually contains a read-only *SquashFS* filesystem and a writable *JFFS2* filesystem. The plug system implements a virtual filesystem, "mini_fo," which merges the read-only *SquashFS* filesystem and the writable *JFFS2* filesystem. When a file is changed, the new file is written to the writable *JFFS2* filesystem while the read-only *SquashFS* filesystem still keeps the original file.

iii) After the certificate is successfully replaced, we can eavesdrop on connections with "mitmproxy."

4) Discovering the Communication Protocol: We now present how to reverse engineer the smart plug's application layer protocol. Based on traffic analysis, we are able to identify strings that start with "MESSAGE-INTEGRITY" or "Authorization," but other fields of such strings are unreadable. We find that these fields are generated with the HMAC-SHA1 algorithm [47] by using the methods in Section III-F. These fields actually contain authentication materials, which are crucial for our security analysis.

B. Communication Protocols of WeMo Smart Plug

We now present the discovered architecture of the WeMo smart plug system and its communication protocols.

1) Architecture of WeMo Smart Plug System: The WeMo smart plug system contains three components: 1) two cloud servers (a traversal using relays around NAT (TURN) server and a HTTPS server); 2) smart plugs; and 3) smartphones. A smart plug and a smartphone can communicate with each other via the cloud servers, as shown in Fig. 10. Since a smart plug is often behind a WiFi router using the NAT, the TURN [48] server is used to perform the NAT traversal for the plug so that a user on the Internet can send a command to the plug. The HTTPS server has three functionalities, including binding, authentication, and controlling (i.e., command relay and information update).

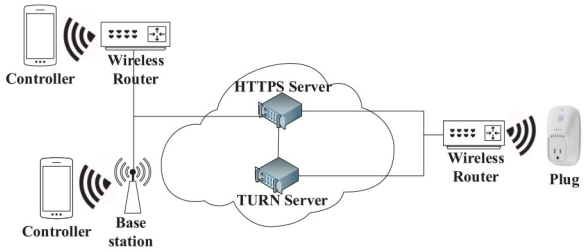


Fig. 10. Architecture of WeMo plug system.

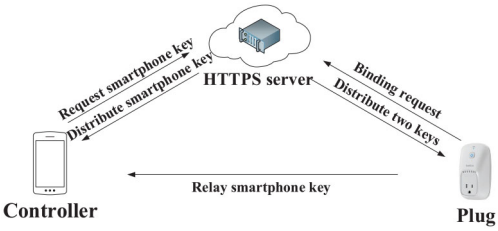


Fig. 11. Binding phase.

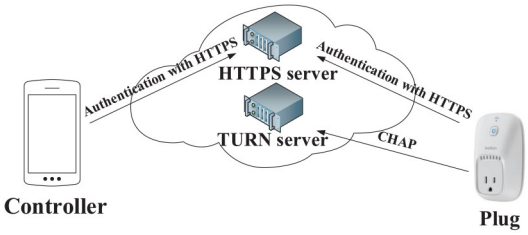


Fig. 12. Authentication phase.

2) Pairing: In the pairing phase, the plug works as an AP and the smartphone connects to it. The smartphone sends a request to the plug to obtain basic information of the plug, such as the MAC address and serial number. After receiving such information, the smartphone sends the plug its identification (ID) and description, a timestamp *TS*, and the home AP's WiFi credentials entered by the user. Then, the plug can access the Internet via the home AP.

3) Binding: The smartphone and smart plug are bound to the HTTPS sever as shown in Fig. 11. The smart plug first sends the binding request, including MAC address, smartphone's ID and description of the plug, SSID and MAC address of WiFi, and timestamp *TS* to the HTTPS server, which can now bind (associate) the particular plug and smartphone together on the basis of the received information. Based on materials contained in the binding request, the HTTPS server produces two keys: 1) the smart plug key and 2) the smartphone key. The HTTPS server then sends these two keys to the smart plug. After obtaining the two keys, the smart plug sends the smartphone key to the smartphone via the local WiFi network. If the smart plug and smartphone are not in the same local network, the smartphone can obtain the smartphone key by sending a request to the HTTPS server that knows the particular smartphone is bound to the particular plug. The request also contains a message authentication code, as introduced below.

4) Authentication: Fig. 12 summarizes the authentication phase. Within the local network, there is no authentication for a smartphone app to control the plug. When the smartphone

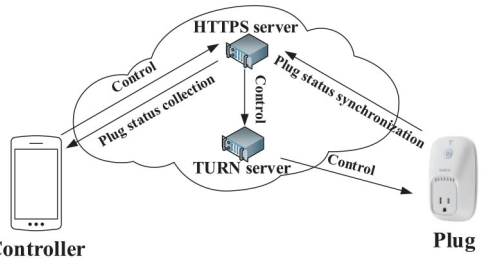


Fig. 13. Remote controlling phase.

and smart plug are not in the same local network, they need to communicate through the HTTPS server. In each message from the plug to the HTPPS server, the HTTP message header includes an "Authorization" field, which contains authentication data. The authentication data are generated by the HMAC-SHA1 algorithm over the plug key and other shared information with the HTTPS server. The HTTPS server authenticates the smartphone in a similar way. The TURN server obtains the smart plug key from the HTTPS server and authenticates the plug via the challenge handshake authentication protocol (CHAP) [49].

5) Remote Controlling: After authentication, the smartphone and smart plug can communicate with the cloud servers as illustrated in Fig. 13. The smart plug periodically synchronizes its status with the HTTPS server. To remotely control the plug, the smartphone first obtains the status of the smart plug by sending a request to the HTTPS server. The status can be either *switch_off* (integer "0") or *switch_on* (integer "1"). When the device is offline, the status is *unavailable* (integer "3"). Then, the smartphone can send control commands to switch on/off the smart plug via the HTTPS server. The HTTPS server actually forwards the commands to the TURN server, which uses the NAT traversal to send the command through the wireless router to the plug.

C. Attacks Against WeMo Plugs

Once the IoT communication protocols are discovered, we can now move forward with security analysis of pairing, binding, authentication, and controlling phases introduced in Section II-B. We discovered two novel attacks against the WeMo smart plug: 1) sharing attack and 2) connection hijacking attack. With the sharing attack, an attacker can remotely control a victim smart plug. The connection hijacking attack allows a DOS attack against a plug. It is worth noting that all the experiments are conducted on the plugs that we purchase.

1) Sharing Attack: We first introduce the details of the binding phase, which involves two binding requests from the plug. The authorization value in the first binding request is "dummy," as the plug key is not derived yet. After receiving the first binding request, the HTTPS server sends back a temporary key. The authorization value in the second binding request from the plug is generated using the temporary key. After receiving the second binding request, the HTTPS server sends the plug key and smartphone key to the plug.

To explore the smart plug resetting phase, we first press the reset button on a smart plug and then bind a new smartphone to the plug. We find now both the original and new smartphones can remotely control the plug. That is, the original and new smartphones now share the plug. Through traffic analysis, we find the plug sends only one binding request, which is regarded as rebinding request, to the HTTPS server. The rebinding request contains a new field, "reRegister." The authorization value is generated using the original plug key. It can be inferred that the original plug key is not erased after resetting.

We find that if we set the authorization value as "dummy" in the rebinding request to pretend that the smart plug loses its key, the HTTPS server will send the original plug key and a new smartphone key to the plug. Once the new smartphone obtains the new smartphone key, the smartphone can pass the authentication of the HTTPS server and access the plug.

Once we understand the plug sharing phase, we are able to bind a victim smart plug to a malicious smartphone. The details of the sharing attack are introduced as follows.

i) To deploy the attack, the attacker needs to obtain the victim plug's MAC address and serial number, as well as the home AP's SSID and MAC address. One limitation of this attack is that the attacker has to use wardriving or other means to get the victim plug's MAC address and home AP's SSID and MAC address. In wardriving, the attacker drives around and performs wireless sniffing. Blocks of MAC addresses are allocated to every manufacturer (Belkin in our case), which can be obtained from the Internet. Therefore, the attacker will be able to identify Belkin smart plugs through wardriving. We also find that a plug's serial number is predictable based on its MAC address. Therefore, the attacker can remotely attack the victim plug after obtaining the needed information.

ii) The attacker can now implement a fake software smart plug that pretends to be the real one. The fake plug sends a rebinding request with the authorization value "dummy" and fabricated smartphone information to the HTTPS server to get a temporary key. Once the plug receives the key, it resends a rebinding request with the authorization value that is generated by the temporary key, and then obtains the original plug key and a new smartphone key.

iii) The attacker now creates a fake software smartphone, which uses the new smartphone key and sends commands with correct authorization value to the HTTPS server. It is worth noting that the HTTPS server has already bound the victim plug and the fake smartphone together. In this way, the attacker can remotely control the target WeMo smart plug while the victim user cannot discover the attack for the sharing feature of the WeMo plug.

2) Connection Hijacking Attack: Once obtaining the plug key through the sharing attack, a fake smart plug can pretend to be the real device so as to hijack the connection between the victim user and the real plug. The details of the attack process are presented as follows.

i) The attacker first creates a fake smart plug that pretends to be the real one and uses it to deploy the sharing attack in Section IV-C1. In this way the attacker obtains the victim smart plug key.

TABLE I
FILE SYSTEM OF IOT DEVICES (CRAMFS, SQUASHFS, AND ROMFS ARE READ-ONLY FILE SYSTEMS AND JFFS2 IS A WRITABLE FILE SYSTEM)

File System \ Manufacturer	Axis	Asmnet	D-Link	TP-Link	Netgaer	Netis	Asus	Total
CramFS	45	0	0	0	0	0	0	45
JFFS2	33	1	0	0	0	0	0	34
SquashFS	0	9	35	13	34	29	57	177
CramFS&JFFS2	249	0	0	0	0	0	0	249
RomFS	6	3	0	0	0	0	0	9
Total	333	15	35	13	34	29	57	514

ii) Since the fake smart plug has the original smart plug key, the fake smart plug can perform the authentication process with the plug system's TURN server to request a relay port, which is shared with the HTTPS server. Therefore, the HTTPS server knows that the fake plug uses that specific TURN server port.

iii) Now, a control command from a victim smartphone is sent from the HTTPS server to the relay port of the fake plug on the TURN server. The command is relayed to the fake plug instead of the real one. The traffic from the smartphone is hijacked by the attacker, who denies the service of the victim smart plug as a matter of fact.

3) Discussion: At the time of writing this article, Belkin has added a security patch trying to defeat our sharing attack. With the patch, if the public source IP address of the rebinding request sent from a plug is changed, the HTTPS server will not send the original plug key, but generate a new smart plug key. Since the victim plug still keeps the old plug key, it will not be able to pass the authentication of the HTTPS server and the TURN server, and cannot be controlled by a controller anymore. Therefore, our sharing attack becomes a DoS attack under the security patch. If a user wants to reuse the victim plug, he/she has to reset the plug.

V. EVALUATION

In this section, we evaluate the generality of our communication protocol reverse engineering framework, present our reverse engineering of a number of real-world IoT system, and discuss the limitations of the proposed framework.

A. Generality of Our Manual Reverse Engineering Framework

The most challenging part of reverse engineering an IoT device is firmware analysis. The firmware may be from different vendors with high customization. Table I shows the mainstream manufacturers and the file systems used by their products. We collected 514 firmware from seven vendors by crawling the Internet. By analyzing these firmware with binwalk, we can identify the file systems used in these firmware. For example, out of the 333 firmware published by Axis, 6 of them use RomFS, 45 use CramFS file system, 33 use JFFS2 file system, and 249 use both CramFS and JFFS2 file systems. The file system can be read-only (e.g., CramFS, SquashFS, or RomFS) or writable (e.g., JFFS2). To reverse engineer these types of firmware, we often need to change the firmware, for example, embedding a fake CA certificate for mitmproxy or a GDBserver for debugging. We can perform such changes with approached introduced in Section III-E. Therefore, we will be able to reserve engineer all the devices listed in Table I while the actual manual reverse engineering tasks may last long given the complexity.

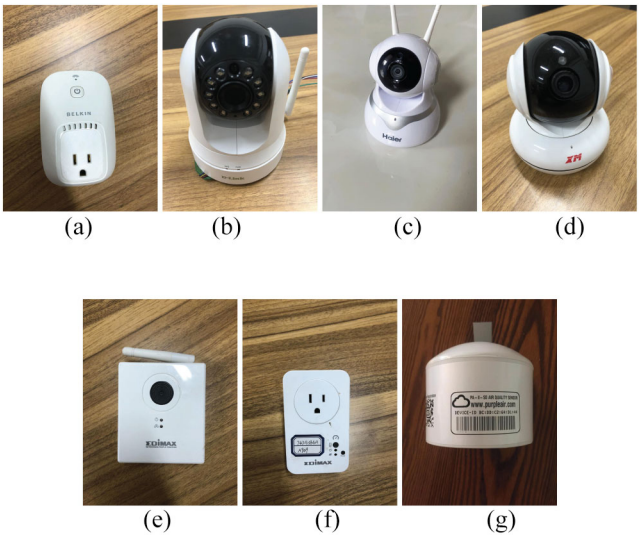


Fig. 14. IoT devices analyzed with our framework. (a) WeMo plug. (b) D-Link camera. (c) Haier camera. (d) Xiongmai camera. (e) Edimax camera. (f) Edimax plug. (g) PurpleAir sensor.

B. Reverse Engineering Real-World IoT Products

Fig. 14 shows all devices we have reverse engineered, including Edimax camera [13], Edimax smart plug [14], and PurpleAir air quality monitoring sensor [50], [51] in our previous work. The PurpleAir air quality monitoring sensors are actually bare metal systems based on microcontrollers (MCUs) without an OS like Linux. Now, our manual reverse engineering framework is still valid. Particularly, OpenOCD and GDB can be used to debug the MCU firmware through JTAG. We now briefly introduce how we used the framework to analyze the other devices that we are the first to have reverse engineered.

We reverse engineered the communication protocol of the D-Link cloud camera system. The camera uses a read-only filesystem and we are able to find the CA certificate. As proposed in Section III-C, we replace the certificate by generating a new firmware with a forged root certificate and flash the new firmware into the target camera through the device management interface. Therefore, we can decrypt the TLS/SSL encrypted traffic, and finally find that the camera is also under the risk of spoofing attacks.

We reverse engineered the communication protocol of the Haier IP camera and the Xiongmai IP camera and find they are vulnerable to the spoofing attack and the Xiongmai IP camera also under an unauthorized access attack.

1) For the Haier IP camera, we find the app is packed to hide the executable files, i.e., dex files. To extract the dex files from the packed app [33], we use Xposed and Fdex2 [52], which is a module of Xposed, to hook the *loadclass* function and extract the dex files. Then, we can hook the app with Xposed and Frida, and perform static data flow analysis and dynamic debugging to the binaries of IoT device using GDB to discover the communication protocol, as shown in Section III-F.
2) For the Xiongmai IP camera, we diassemble the camera app for static analysis and use code instrumentation techniques, such as hooking through Frida [41] to analyze the app side communication protocol. We also disassemble the firmware, embed gdbserver onto a flash and use GDB to dynamically debug the binary files of the firmware.

C. Limitations

Our communication protocol, reverse engineering framework, has the following limitations. If an IoT device employs secure boot and the firmware image verification key is in secure storage, such as e-fuse, we may not be able to change the firmware of the device, since secure boot will detect the change and refuse to start the device. Similarly, if flash encryption is enabled and the related keys are in secure storage, we cannot change the device firmware since we cannot obtain these keys. However, we find few IoT products use such secure boot and flash encryption.

VI. Related Work

In this section, we review the existing technologies for analyzing the security of IoT devices and Android apps. Particularly, we divided the state of the art into three categories, i.e., static analysis, dynamic analysis, and hybrid analysis approaches.

Static Analysis: Some static analysis approaches have been proposed to analyze the security of the IoT device firmware [13], [53]–[57] and Android apps [58]–[63]. For example, Costin *et al.* [53] preformed a large-scale static analysis of IoT device firmware with correlation engine that could evaluate the similarity between the target IoT device firmware and the vulnerable ones so as to determine whether the target firmware contains existing vulnerabilities. Nirumand *et al.* [61] proposed a model driven reverse engineering (MDRE)-based static analysis method to discover the security risks in the Android app communication. The static analysis approaches are fast and can reach comprehensive code coverage of the firmware or app [64], [65]. However, some IoT device firmware and Android apps are obfuscated or encrypted, which cannot be disassembled and statically analyzed [66], [67]. In addition, the runtime behavior, such as user input could not be statically determined and static analysis may cause false positives and false negatives [64], [65], [68].

Dynamic Analysis: Dynamic analysis approaches could observe the runtime behavior of the target app and IoT device firmware and could be used to verify the correctness of the results of static analysis approaches by running the app or IoT device firmware with test cases [64]. For Android apps, Zheng *et al.* [65] proposed a dynamic analysis framework based on ptrace (process trace), which is a system call that could be used by one process to control another. The framework uses ptrace to monitor selected system calls to dynamically analyze malicious behaviors of the binary. The frameworks of dynamic analysis methods for IoT device firmware can be divided into two categories, i.e., software emulator-based frameworks as well as the real IoT device hardware and the emulator-based frameworks. For the first category, the IoT device firmware is performed on a software emulator and applied the dynamic analysis methods [69]–[71]. For example, Chen *et al.* [69] presented FIRMADYNE, which is a dynamic debugging framework based on the emulator with an instrumented kernel. Fourteen previously unknown vulnerabilities were discovered by using FIRMADYNE with automated webpages analysis and manual analysis.

Since the IoT device hardware is fairly diverse, it is nontrivial to emulate various IoT device hardware with software emulators [72]. To address this problem, some frameworks have been proposed, which relay I/O accesses between real IoT device hardware and the emulator [72]–[74]. For instance, Zaddach *et al.* [72] presented Avatar, which is a framework that dynamically analyzes the IoT devices by combining the emulator and the real hardware. The framework forwards the I/O accesses from the emulator to the real IoT device. The framework was evaluated with KLEE symbolic execution engine and existing fuzzing tools. However, dynamic analysis is time consuming as it requires numerous test cases to ensure a certain degree of credibility for vulnerability detection. In addition, it is difficult to generate valid test cases [64], [68].

Hybrid Analysis: Hybrid analysis methods, which combines the static and dynamic analysis technologies, have been proposed [16], [64], [67], [68], [75]–[78] to improve the accuracy of vulnerability discovery. For example, Martinelli *et al.* [67] proposed a framework to detect malicious apps by performing both static and dynamic analyzing approaches. They evaluated the framework using 2794 malicious apps with high detection accuracy. Palavicini *et al.* [77] performed static analysis on IoT firmware to avoid path explosion when dynamically analyzing complex binaries with symbolic execution using a software emulator. Yao *et al.* [16] identified a previously unknown vulnerability, which is known as privilege separation vulnerability. They leveraged firmware loading information extraction, library function recognition, and symbolic execution methods to analyze the IoT device firmware and located 69 of 106 firmware containing privilege separation vulnerabilities.

Those existing technologies could not be used to probe into the various vulnerabilities located in the communication protocol of IoT systems [3], [56], [57], [79] and there is little systematically communication protocol reverse engineering approaches, since it is a great challenge to reverse engineer these protocols given the diversity of protocol implementation.

For example, Papp *et al.* [56] and Shwartz *et al.* [57] proposed the methods for reverse engineering IoT devices. They only focus on discovering the vulnerabilities in the firmware of IoT device instead of the security analysis of the communication protocol between the controller and device. To tackle this problem, we propose a framework to reverse engineering communication protocols of Linux-based IoT systems for further protocol security analysis in this article.

VII. Conclusion

In this article, we proposed a framework to manually reverse engineer the communication protocols of IoT devices so that the discovered protocol can be used for further security analysis. The framework works as follows: obtaining the app and firmware of an IoT device, collecting network traffic generated by the device and control app, defeating traffic protection, and discovering the communication protocol through traffic analysis, static analysis and dynamic analysis of the app and firmware. We presented a case study of using the framework to reverse engineer the communication protocols of the WeMo smart plug. Once the plug's communication protocols are discovered, we are able to identify a crucial authentication vulnerability that allows the plug sharing attack to control victim plugs and connection hijacking attack for DoS. We demonstrated our framework is generic and could be applied to a variety of embedded Linux-based IoT systems using either read-only or writable filesystems. We also briefly discussed how we applied the framework to a few other real-world IoT products and systems. We are the first to systematically propose such a manual communication protocol reverse engineering framework.

Acknowledgment

Any opinions, findings, conclusions, and recommendations in this article are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] E. Ronen and A. Shamir, "Extended functionality attacks on IoT devices: The case of smart lights," in *Proc. 1st Eur. Symp. Security Privacy (EuroS&P)*, Saarbrucken, Germany, Mar. 2016, pp. 3–12.

[2] Z. B. Celik *et al.*, "Sensitive information tracking in commodity IoT," in *Proc. 27th USENIX Security Symp. (USENIX Security)*, Baltimore, MD, USA, Aug. 2018, pp. 1687–1704.

[3] W. Zhou *et al.*, "Discovering and understanding the security hazards in the interactions between IoT devices, mobile apps, and clouds on smart home platforms," in *Proc. 28th USENIX Security Symp. (USENIX Security)*, Santa Clara, CA, USA, Aug. 2019, pp. 1133–1150.

[4] E. Bertino, "Data security privacy in the IoT," in *Proc. 19th Int. Conf. Extend. Database Technol. (EDBT)*, Bordeaux, France, Mar. 2016, pp. 1–3.

[5] Y. Jia *et al.*, "Burglars' IoT paradise: Understanding and mitigating security risks of general messaging protocols on IoT clouds," in *Proc. IEEE Symp. Security Privacy (SP)*. Los Alamitos, CA, USA, May 2020, pp. 465–481. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP40000.2020.00051

[6] Craig. (May 2014). *Hacking the D-Link DSP-W215 Smart Plug*. [Online]. Available: http://www.devttys0.com/2014/05/hacking-the-d-link-dsp-w215-smart-plug

[7] R. Chirgwin. (Nov. 2016). *Get PWNED: Web CCTV Cams Can Be Hijacked by Single HTTP Request*. [Online]. Available: https://www.theregister.co.uk/2016/11/30/iot_cameras_compromised_by_long_url/

[8] C. Zach. (Feb. 2013). *D-Link Dir-815 Upnp Command Injection*. [Online]. Available: http://shadow-file.blogspot.hu/2013/02/dlink-dir-815-upnp-command-injection.html

[9] J. Chen *et al.*, "IoTFuzzer: Discovering memory corruptions in IoT through app-based fuzzing," in *Proc. 25th Annu. Netw. Distrib. Syst. Security Symp. (NDSS)*, San Diego, CA, USA, Feb. 2018, pp. 1–15.

[10] J. Y. Kim, R. Holz, W. Hu, and S. Jha, "Automated analysis of secure Internet of Things protocols," in *Proc. 33rd Annu. Comput. Security Appl. Conf. (ACSAC)*, Orlando, FL, USA, Dec. 2017, pp. 238–249.

[11] B. Aziz, "A formal model and analysis of an IoT protocol," *Ad Hoc Netw.*, vol. 36, pp. 49–57, Jan. 2016.

[12] M. Mohsin, Z. Anwar, G. Husari, E. Al-Shaer, and M. A. Rahman, "IoTSAT: A formal framework for security analysis of the Internet of Things (IoT)," in *Proc. 4th IEEE Conf. Commun. Netw. Security (CNS)*, Philadelphia, PA, USA, Oct. 2016, pp. 180–188.

[13] Z. Ling, K. Liu, Y. Xu, Y. Jin, and X. Fu, "An end-to-end view of IoT security and privacy," in *Proc. 60th IEEE Global Commun. Conf. (GLOBECOM)*, Singapore, Dec. 2017, pp. 1–7.

[14] Z. Ling, J. Luo, Y. Xu, C. Gao, K. Wu, and X. Fu, "Security vulnerabilities of Internet of Things: A case study of the smart plug system," *IEEE Internet Things J.*, vol. 4, no. 6, pp. 1899–1909, Dec. 2017.

[15] Eclipse Foundation Inc. (Apr. 2018). *IoT Developer Survey Results*. [Online]. Available: https://iot.eclipse.org/resources/iot-developer-survey/iot-developer-survey-2018.pdf

[16] Y. Yao, W. Zhou, Y. Jia, L. Zhu, P. Liu, and Y. Zhang, "Identifying privilege separation vulnerabilities in IoT firmware with symbolic execution," in *Proc. 24th Eur. Symp. Res. Comput. Security (ESORICS)*, Luxembourg, Luxembourg City, Sep. 2019, pp. 638–657.

[17] C. Gao, Z. Ling, B. Chen, X. Fu, and W. Zhao, "SecT: A lightweight secure thing-centered IoT communication system," in *Proc. 15th Int. Conf. Mobile Ad Hoc Sensor Syst. (MASS)*, Chengdu, China, Oct. 2018, pp. 46–54.

[18] M. Egele, C. Kruegel, E. Kirda, and G. Vigna, "PiOS: Detecting privacy leaks in iOS applications," in *Proc. 18th Netw. Distrib. Syst. Security Symp. (NDSS)*, San Diego, CA, USA, Feb. 2011, pp. 177–183.

[19] D. Orikogbo, M. Büchler, and M. Egele, "CRiOS: Toward large-scale iOS application analysis," in *Proc. 6th Workshop Security Privacy Smartphones Mobile Devices (SPSM)*, Vienna, Austria, Oct. 2016, pp. 33–42.

[20] M. Erfani and A. Mesbah, "Reverse engineering iOS mobile applications," in *Proc. 19th Working Conf. Reverse Eng. (WCRE)*, Kingston, ON, Canada, Oct. 2012, pp. 177–186.

[21] M. Shavit, A. Gryc, and R. Miucic, "Firmware update over the air (FOTA) for automotive industry," SAE, Warrendale, PA, USA, Rep. 2007-01-3523, 2007.

[22] D. Prototypes. *Bus Pirate Homepage*. Accessed: Nov. 21, 2020. [Online]. Available: http://dangerousprototypes.com/docs/Bus_Pirate

[23] J. W. M. Oh, "Reverse engineering flash memory for fun and benefit," in *Proc. 17th Black Hat*, Aug. 2014, pp. 1–35.

[24] DediProg. (Nov. 2018). *StarProg-F Engineering Programmer*. [Online]. Available: https://www.dediprog.com/product/StarProg-F

[25] J. Malinen. *HostAPD: IEEE 802.11 AP, IEEE 802.1X/WPA/WPA2/EAP/RADIUS Authenticator*. Accessed: Nov. 21, 2020. [Online]. Available: https://w1.fi/hostapd/

[26] A. Cortesi, M. Hils, and T. Kriechbaumer. (2010). *Mitmproxy: A Free and Open Source Interactive HTTPS Proxy, Version 5.3*. Accessed: Nov. 21, 2020. [Online]. Available: https://mitmproxy.org/

[27] Google Inc. *How Can I Trust Cacert's Root Certificate?* Accessed: Nov. 21, 2020. [Online]. Available: https://sites.google.com/site/jamestu6166workingtips/how-can-i-trust-cacert-s-root-certificate

[28] T. Connor and W. Ryszard. *ApkTool*. Accessed: Nov. 21, 2020. [Online]. Available: https://ibotpeaches.github.io/Apktool

[29] Google Inc. (Nov. 2018). *Signapk—Onboard APK Signing Script for Android Devices*. Accessed: Nov. 21, 2020. [Online]. Available: https://code.google.com/archive/p/signapk/

[30] C. M. Ellison, B. Frantz, B. W. Lampson, R. Rivest, B. Thomas, and T. Ylönen, "SPKI certificate theory," IETF, RFC 2693, 1999. Accessed: Nov. 21, 2020.

[31] C. Heffner. (2010). *Binwalk: Firmware Analysis Tool*. Accessed: Nov. 21, 2020. [Online]. Available: https://code.google.com/p/binwalk/

[32] MITRE. *Obfuscation or Cryptography*. Accessed: Nov. 21, 2020. [Online]. Available: https://attack.mitre.org/versions/v7/techniques/T1313/

[33] Y. Zhang, X. Luo, and H. Yin, "DexHunter: Toward extracting hidden code from packed android applications," in *Proc. 20th Eur. Symp. Res. Comput. Security (ESORICS)*. Vienna, Austria, Sep. 2015, pp. 293–311.

[34] W. Yang *et al.*, "AppSpear: Bytecode decrypting and dex reassembling for packed android malware," in *Proc. 18th Int. Symp. Res. Attacks Intrusions Defenses (RAID)*, Kyoto, Japan, Nov. 2015, pp. 359–381.

[35] L. Xue, X. Luo, L. Yu, S. Wang, and D. Wu, "Adaptive unpacking of android apps," in *Proc. 39th IEEE/ACM Int. Conf. Softw. Eng. (ICSE)*, Buenos Aires, Argentina, May 2017, pp. 358–369.

[36] (2019). *Smali2java*. Accessed: Nov. 21, 2020. [Online]. Available: https://github.com/demitsuri/smali2java

[37] Google Inc. and JetBrains Inc. (2017). *Android Studio*. Accessed: Nov. 21, 2020. [Online]. Available: https://developer.android.google.cn/studio/intro

[38] Hex-Rays SA. (2008). *IDA Pro Disassembler*. Accessed: Nov. 21, 2020. [Online]. Available: https://www.hex-rays.com/products/ida/

[39] Google Inc. *Android Debug Bridge (ADB)*. Accessed: Nov. 21, 2020. [Online]. Available: https://developer.android.com/studio/command-line/adb

[40] ROVO89. (Mar. 2019). *Xposed Module Repository*. [Online]. Available: http://repo.xposed.info

[41] *Dynamic Instrumentation Toolkit for Developers, Reverse-Engineers, and Security Researchers*. Accessed: Jul. 22, 2020. [Online]. Available: https://frida.re/

[42] S. Arzt *et al.*, "FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *Proc. 35th ACM SIGPLAN Conf. Program. Lang. Design Implement. (PLDI)*, Edinburgh, U.K., Jun. 2014, pp. 259–269.

[43] M. Laurenzano, M. M. Tikir, L. Carrington, and A. Snavely, "PEBIL: Efficient static binary instrumentation for Linux," in *Proc. 10th Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, White Plains, NY, USA, Mar. 2010, pp. 175–183.

[44] R. Stallman, R. Pesch, and S. Shebs, "Debugging with GDB," Free Softw. Found., Boston, MA, USA, Rep. 2008-04-18.10, 2002.

[45] C. Zuo, W. Wang, Z. Lin, and R. Wang, "Automatic forgery of cryptographically consistent messages to identify security vulnerabilities in mobile services," in *Proc. 23rd Annu. Netw. Distrib. Syst. Security Symp. (NDSS)*, San Diego, CA, USA, Feb. 2016, pp. 1–17.

[46] S. Arzt, "Static data flow analysis for android applications," Ph.D. dissertation, zur Erlangung des akademischen Grades Doktor-Ingenieur, Technische Universität, Munich, Germany, 2017.

[47] H. Krawczyk, M. Bellare, and R. Canetti, "HMAC: Keyed-hashing for message authentication," IETF, RFC 2104, 1997.

[48] S. Perreault and J. Rosenberg, "Traversal using relays around NAT (TURN) extensions for tcp allocations," IETF, RFC 6062, 2010.

[49] W. Simpson, "PPP challenge handshake authentication protocol (CHAP)," IETF, RFC 1994, 1996.

[50] L. Luo, Y. Zhang, B. Pearson, Z. Ling, H. Yu, and X. Fu, "On the security and data integrity of low-cost sensor networks for air quality monitoring," *Sensors*, vol. 18, no. 19, p. 4451, 2018.

[51] C. Gao, L. Luo, Y. Zhang, B. Pearson, and X. Fu, "Microcontroller based IoT system firmware security: Case studies," in *Proc. IEEE Int. Conf. Ind. Internet (ICII)*, Orlando, FL, USA, Nov. 2019, pp. 200–209.

[52] C. Tumbleson and R. Wisniewski. (2019). *FDex2*. [Online]. Available: https://github.com/HangZhouCat/ReaverAPKTools

[53] A. Costin, J. Zaddach, A. Francillon, and D. Balzarotti, "A large-scale analysis of the security of embedded firmwares," in *Proc. 23rd USENIX Security Symp. (USENIX Security)*, San Diego, CA, USA, Aug. 2014, pp. 95–110.

[54] Y. David, N. Partush, and E. Yahav, "FirmUp: Precise static detection of common vulnerabilities in firmware," in *Proc. 23rd Int. Conf. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, Williamsburg, VA, USA, Mar. 2018, pp. 392–404.

[55] J. Kinder and H. Veith, "Jakstab: A static analysis platform for binaries," in *Proc. 20th Comput.-Aided Verification (CAV)*, Princeton, NJ, USA, Jul. 2008, pp. 423–427.

[56] D. Papp, K. Tamás, and L. Buttyán, "IoT hacking—A primer," *Infocommun. J.*, vol. 11, no. 2, pp. 1–12, 2019.

[57] O. Shwartz, Y. Mathov, M. Bohadana, Y. Elovici, and Y. Oren, "Reverse engineering iot devices: Effective techniques and methods," *IEEE Internet Things J.*, vol. 5, no. 6, pp. 4965–4976, Dec. 2018.

[58] D. Gallingani, R. Gjomemo, V. N. Venkatakrishnan, and S. Zanero, "Practical exploit generation for intent message vulnerabilities in android," in *Proc. 5th ACM Conf. Data Appl. Security Privacy (CODASPY)*, San Antonio, TX, USA, Mar. 2015, pp. 155–157.

[59] K. Lu *et al.*, "Checking more and alerting less: Detecting privacy leakages via enhanced data-flow analysis and peer voting," in *Proc. 22nd Annu. Netw. Distrib. Syst. Security Symp. (NDSS)*, San Diego, CA, USA, Feb. 2015, pp. 1–15.

[60] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard, "Information flow analysis of android applications in droidsafe," in *Proc. 22nd Annu. Netw. Distrib. Syst. Security Symp. (NDSS)*, San Diego, CA, USA, Feb. 2015, pp. 1–16.

[61] A. Nirumand, B. Zamani, and B. T. Ladani, "VAnDroid: A framework for vulnerability analysis of android applications using a model—Driven reverse engineering technique," *J. Softw. Pract. Exp.*, vol. 49, no. 1, pp. 70–99, 2019.

[62] S. Biswas, K. Sharif, F. Li, and Y. Liu, "3P framework: Customizable permission architecture for mobile applications," in *Proc. 12nd Int. Conf. Wireless Algorithms Syst. Appl. (WASA)*, Guilin, China, Jun. 2017, pp. 445–456.

[63] S. Seo, A. Gupta, A. M. Sallam, E. Bertino, and K. Yim, "Detecting mobile malware threats to homeland security through static analysis," *J. Netw. Comput. Appl.*, vol. 38, pp. 43–53, Feb. 2014.

[64] A. Aggarwal and P. Jalote, "Integrating static and dynamic analysis for detecting vulnerabilities," in *Proc. 30th Annu. Int. Comput. Softw. Appl. Conf. (COMPSAC)*, Chicago, IL, USA, Sep. 2006, pp. 343–350.

[65] M. Zheng, M. Sun, and J. C. S. Lui, "DroidTrace: A ptrace based android dynamic analysis system with forward execution capability," in *Proc. 10th Int. Wireless Commun. Mobile Comput. Conf. (IWCMC)*, Nicosia, Cyprus, Aug. 2014, pp. 128–133.

[66] S. Hao, B. Liu, S. Nath, W. G. J. Halfond, and R. Govindan, "PUMA: Programmable UI-automation for large-scale dynamic analysis of mobile apps," in *Proc. 12th Annu. Int. Conf. Mobile Syst. Appl. Services (MobiSys)*, Bretton Woods, NH, USA, Jun. 2014, pp. 204–217.

[67] F. Martinelli, F. Mercaldo, A. Saracino, and C. A. Visaggio, "I find your behavior disturbing: Static and dynamic app behavioral analysis for detection of android malware," in *Proc. 14th Annu. Conf. Privacy Security Trust (PST)*, Auckland, New Zealand, Dec. 2016, pp. 129–136.

[68] D. Sounthiraraj, J. Sahs, G. Greenwood, Z. Lin, and L. Khan, "SMV-Hunter: Large scale, automated detection of SSL/TLS man-in-the-middle vulnerabilities in Android apps," in *Proc. 21st Annu. Netw. Distrib. Syst. Security Symp. (NDSS)*, San Diego, CA, USA, Feb. 2014, pp. 1–14.

[69] D. D. Chen, M. Woo, D. Brumley, and M. Egele, "Towards automated dynamic analysis for Linux-based embedded firmware," in *Proc. 23rd Annu. Netw. Distrib. Syst. Security Symp. (NDSS)*, San Diego, CA, USA, Feb. 2016, pp. 1–16.

[70] E. Gustafson *et al.*, "Toward the analysis of embedded firmware through automated re-hosting," in *Proc. 22nd Int. Symp. Res. Attacks Intrusions Defenses (RAID)*, Beijing, China, Sep. 2019, pp. 135–150.

[71] B. Feng, A. Mera, and L. Lu, "P^{2}IM: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling," in *Proc. 29th USENIX Security Symp. (USENIX Security)*, Boston, MA, USA, Aug. 2020, pp. 1237–1254.

[72] J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti, "Avatar: A framework to support dynamic security analysis of embedded systems' firmwares," in *Proc. 21st Annu. Netw. Distrib. Syst. Security Symp. (NDSS)*, San Diego, CA, USA, Feb. 2014, pp. 1–16.

[73] M. Muench, D. Nisi, A. Francillon, and D. Balzarotti, "Avatar 2: A multi-target orchestration platform," in *Proc. 1st Workshop Binary Anal. Res. NDSS Symp. (BAR)*, San Diego, CA, USA, Feb. 2018, p. 18.

[74] N. Corteggiani, G. Camurati, and A. Francillon, "Inception: System-wide security testing of real-world embedded systems software," in *Proc. 27th USENIX Security Symp. (USENIX Security)*, Baltimore, MD, USA, Aug. 2018, pp. 309–326.

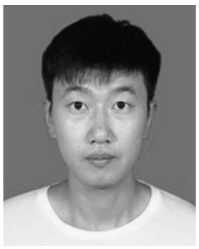
[75] M. Spreitzenbarth, T. Schreck, F. Echtler, D. Arp, and J. Hoffmann, "Mobile-sandbox: Combining static and dynamic analysis with machine-learning techniques," *Int. J. Inf. Security*, vol. 14, no. 2, pp. 141–153, 2015.

[76] H. Wang, Y. Guo, Z. Tang, G. Bai, and X. Chen, "Reevaluating Android permission gaps with static and dynamic analysis," in *Proc. 58th IEEE Global Commun. Conf. (GLOBECOM)*, San Diego, CA, USA, Dec. 2015, pp. 1–6.

[77] G. Palavicini, Jr., J. Bryan, E. Sheets, M. Kline, and J. San Miguel, "Towards firmware analysis of industrial Internet of Things (IIoT)—Applying symbolic analysis to IIoT firmware vetting," in *Proc. 2nd Int. Conf. Internet Things Big Data Security*, Porto, Portugal, Apr. 2017, pp. 470–477.

[78] V. Visoottiviseth, P. Jutadhammakorn, N. Pongchanchai, and P. Kosolyudhthasarn, "FirMaster: Analysis tool for home router firmware," in *Proc. 15th Int. Joint Conf. Comput. Sci. Softw. Eng. (JCSSE)*, Nakhonpathom, Thailand, Jul. 2018, pp. 1–6.

[79] J. Chen *et al.*, "Your IoTs are (NOT) mine: On the remote binding between IoT devices and users," in *Proc. 49th Annu. IEEE/IFIP Int. Conf. Depend. Syst. Netw. (DSN)*, Portland, OR, USA, Jun. 2019, pp. 222–233.



Kaizheng Liu received the B.S. degree in computer science and engineering from Southeast University, Nanjing, China, in 2017, where he is currently pursuing the Ph.D. degree in computer science and engineering.

His current research interests include Internet of Things, privacy, and security.

Ming Yang received the Ph.D. degree in computer science from Southeast University, Nanjing, China, in 2007.

He is currently a Professor and the Ph.D. supervisor with the School of Computer Science and Engineering, Southeast University. His main research interests include network security and privacy and Internet of Things.

Zhen Ling (Member, IEEE) received the B.S. degree in computer science from Nanjing Institute of Technology, Nanjing, China, in 2005, and the Ph.D. degree in computer science from Southeast University, Nanjing, 2014.

He is an Associate Professor with the School of Computer Science and Engineering, Southeast University. His research interests include network security, privacy, and Internet of Things.

Dr. Ling won ACM China Doctoral Dissertation Award in 2014 and the China Computer Federation Doctoral Dissertation Award in 2015.

Huaiyu Yan received the B.S. degree in software engineering from Southeast University, Nanjing, China, in 2019, where he is currently pursuing the Ph.D. degree in computer science and engineering.

His current research interests include Internet of Things, privacy, and security.

Yue Zhang is currently pursuing the Ph.D. degree with the College of Information Science and Technology, College of Cyber-Security, Jinan University, Guangzhou, China, under the supervision of J. Weng.

He also studied and worked with the University of Central Florida, Orlando, FL, USA, and the University of Massachusetts Lowell, Lowell, MA, USA, under the supervision of X. Fu. He has published papers in international conferences and journals, such as USENIX Security, IEEE INFOCOM, IEEE TDSC, IEEE TPDS, IEEE TVT, and RAID. His research focuses on system security and especially IoT security.

Xinwen Fu (Senior Member, IEEE) received the B.S. degree from Xi'an Jiaotong University, Xi'an, China, in 1995, the M.S. degree in electrical engineering from the University of Science and Technology of China, Hefei, China, in 1998, and the Ph.D. degree in computer engineering from Texas A&M University, College Station, TX, USA, in 2005.

He is a Professor with the Department of Computer Science, University of Massachusetts Lowell, Lowell, MA, USA. His current research interests include computer security and privacy, and digital forensics. His research was reported by various media, such as Wired and aired on CNN and CCTV 10.

Wei Zhao (Fellow, IEEE) received the Ph.D. degree in computer and information sciences from the University of Massachusetts Amherst, Amherst, MA, USA, in 1986.

Since 1986, he has been serving as a Faculty Member with Amherst College, Amherst; the University of Adelaide, Adelaide, SA, Australia; and Texas A&M University, College Station, TX, USA. From 2005 to 2006, he served as the Director of the Division of Computer and Network Systems, National Science Foundation, Alexandria, VA, USA, when he was on leave from Texas A&M University, where he served as a Senior Associate Vice President for Research and a Professor of Computer Science. He served as the Dean of the School of Science, Rensselaer Polytechnic Institute, Troy, NY, USA, from 2007 to 2008. He was the Founding Director of the Texas A&M Center for Information Security and Assurance, which has been recognized as the Center of Academic Excellence in Information Assurance Education by the National Security Agency. He was the Rector of the University of Macau, Macau, China. He is currently a Chief Research Officer with the American University of Sharjah, Sharjah, UAE. His current research interests include distributed computing, real-time systems, computer networks, and cyberspace security.