Characterizing the Security Facets of IoT Device Setup

Han Yang
Dalhousie University
Halifax, Canada
hn252486@dal.ca

Carson Kuzniar
Dalhousie University
Halifax, Canada
carson.kuzniar@dal.ca

Chengyan Jiang
Dalhousie University
Halifax, Canada
ch614775@dal.ca

Ioanis Nikolaidis
University of Alberta
Edmonton, Canada
nikolaidis@ualberta.ca

Israat Haque
Dalhousie University
Halifax, Canada
israat@dal.ca

Abstract

In this work, we characterize the potential information leakage from IoT platforms during their setup phase. Setup involves an IoT device, its "app", and a cloud-based service. We assume that the on-device firmware is inaccessible, e.g., read-protected. We focus on the combination of information that can be extracted from analyzing the app and the local communication between the app and the IoT device. An attacker can trivially obtain the app, analyze its operation, and potentially eavesdrop on the wireless communication occurring during the setup phase. We develop a semi-automated general methodology involving off-the-shelf tools to examine information disclosure during the setup phase. We tested our methodology on twenty commodity-grade IoT devices. The outcome reveals a wide range of device-dependent choices for encryption at various layers and the potential for exposure of, among other things, device-identifying information and local networking (WiFi) credentials. Our methodology contributes towards a means to assess and "certify" IoT devices.

CCS Concepts

• **Networks** → **Mobile and wireless security**; **Home networks**;
• **Security and privacy**;

Keywords

Smart Home; IoT; Setup Security; Information Leakage

ACM Reference Format:
Han Yang, Carson Kuzniar, Chengyan Jiang, Ioanis Nikolaidis, and Israat Haque. 2024. Characterizing the Security Facets of IoT Device Setup. In *Proceedings of the 2024 ACM Internet Measurement Conference (IMC '24), November 4–6, 2024, Madrid, Spain.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3646547.3688433

1 Introduction

The deployment of consumer-grade Internet of Things (IoT) devices is growing as a result of the trend for "smart homes" because of the ease of installation and convenience of operation they usually exhibit. There are a plethora of commonplace applications automated by IoT devices in a smart home, such as adjusting the thermostat, scheduling the brewing of coffee, detecting movement using cameras, (un)locking doors, etc., and they often involve also a smartphone application for convenience of use and configuration. Smart home IoT devices are expected to reach 100 million in the US alone by 2028 [23]. Such growth also brings security, privacy, and safety risks as these devices, if compromised, can be pivot points for intruders to interfere with the operation of a smart home [30, 31]. A typical smart home IoT platform comprises of (a) IoT devices, (b) their controlling applications (apps), and (c) the back-end cloud-based applications [2]. The applications on the app and the cloud can control and manage the device, adding also value, e.g., in the form of analytics, derived from the device data. The life cycle of a newly acquired IoT platform starts by turning on the IoT device, setting it up for operation, and using it.

The three platform components (device, app, and cloud) often exchange sensitive information (e.g., identity information) to establish preliminary trust before a device can be fully used. For example, consider the case where a user clicks the "power on" button of a smart plug app. It must already have established trust with the command initiator (app), command handler (cloud), and the target (smart plug) to respond accordingly and to keep the applications informed about the state of the plug. The pre-established trust requires exchanging sensitive information from a device to an app and cloud during the *platform setup phase*. However, as we detail in this paper, sensitive information (e.g., Device_ID or login credentials) could be revealed, potentially allowing attackers to fully control a device [7, 32]. Thus, assessing vulnerabilities of the setup phase and their consequences is crucial for the secure operations of smart home IoT devices. For the most part, existing works have taken an occasional interest in the *setup* phase, e.g., unexpected pairing [1, 6], protocol-oriented flaws leading to potential information leakage [19, 34], etc. We posit that the information leakage can have significant downstream effects on users and services, and a systematic assessment is necessary.

Compared to previous research [1, 6, 19, 34] related to the setup phase, we attempt to answer a somewhat more pragmatic question: *if we were to purchase today a handful of IoT devices for different smart home tasks, what fraction of those involve a setup phase that leaks sensitive information (SI)?* We study a sample of twenty IoT devices and find that two-thirds involve some degree of leaking SI. The extent to which we found exposed SI is directly influenced

IMC '24, November 4–6, 2024, Madrid, Spain

ACM ISBN 979-8-4007-0592-2/24/11
https://doi.org/10.1145/3646547.3688433

by the methodology we follow. Our methodology is explained in the next sections and involves steps to identify the SI and its use across the platform. It is conceivable that an improved methodology would find a larger fraction of devices with exposed SI. A fraction of two-thirds can already be considered alarming. The lack of stronger safeguards during setup may be explained by an attitude of treating setup as an unlikely target for an attacker because (i) in most cases, it takes place only once, (ii) at an unpredictable point in time, and (iii) is brief in overall duration. As we later explain, reasonable threat models exist despite the eccentricities of the setup phase.

In summary, we identify a new possible SI exposure channel during the *setup* phase, where users' apps and devices establish *local communication* with each other. Moreover, we develop a semi-automated generalized assessment methodology based on the following research questions:
(RQ1) Which devices are "beaconing" sensitive information? We define *beaconing* as public over-the-air (OTA) information transmission which can be captured by (nearby) packet sniffers and are interested in the separation between beaconing and non-beaconing data.
(RQ2) Can app analysis be used to break channel encryption? Many vendors encrypt sensitive information during the *local communication*; however, it is possible, including for any attacker, to obtain and analyze the corresponding app to determine the used encryption/decryption logic.
(RQ3) Which exposed information is sensitive? RQ1 and RQ2 allow us to generate a list of potentially exposed information during *local communication*. We target assessing the list to discover information that qualifies as sensitive based on its potential downstream effects.

We begin by capturing the OTA traffic during the device setup process. Then, we apply binary analysis to the traces to recover human-readable information. For devices where this process fails, we recruit the help of decompilation of the corresponding app. Finally, we intercept the traffic between the app and the cloud to verify that the information recovered from the traces is used throughout the platform. By focusing our approach on setup traces and accompanying apps, we eschew cumbersome analysis techniques, e.g., device firmware reverse engineering, and more importantly, obtain a better understanding of what information is readily available in the wild.

Twenty devices, from a total of 16 different vendors, for various representative (e.g., plug, camera) smart home tasks were acquired. The summary of findings is that eleven were found to leak Device_ID, eight exposed users' home access point WiFi credentials, and two revealed app credentials. While some platforms are secure, many expose sensitive information even if they apply some form of encryption or other security measures.

2 Background

Sensitive Information. We define two types of sensitive information with respect to authentication or confidentiality.
Device_ID is the unique device identifier that the app, cloud, and device agree upon using a specific device instance authentication. Device_ID may be hard-coded or dynamically assigned/bound by the cloud, e.g., the app submits the device model to the cloud, which assigns a Device_ID to the device [7, 32]. Similarly, a Construct_ID is the information used to derive the Device_ID. For example. Construct_ID could be a serial number submitted to the cloud, with the cloud subsequently generating Device_ID and sharing it with both device and app.
Credentials are any information required to access a system, service, or resource, e.g., home-AP WiFi credentials, app/user account, and password. Potential consequences of credential information exposure can be severed, including unhindered access by an attacker to a user's home-AP, network, data, app online accounts, etc.
Platform States. There are three cloud-based IoT platform architectures insofar the interaction of the three components (device, app, cloud) is concerned: cloud-in-the-middle (CITM), trigger-action (TA), and app-in-the-middle (AITM), with CITM being the most common one [20]. We focus on CITM-based smart home IoT platforms. In CITM, the device and the app depend on the cloud service to function correctly after any initial setup. This paradigm allows the device to operate and communicate with the cloud without the app's ever-present involvement. We also distinguish the following macro states:
Local configuration: The three components communicate necessary information in this initial setup state, e.g., the device shares Device_ID or Construct_ID with the app, which responds with the home-AP WiFi credentials to enable Internet access for operations in the subsequent states. We subsequently call *local communication* the app–device communication, which usually uses Bluetooth Low Energy (BLE) or WiFi. Standard pairing (central-peripheral mode) is used in BLE during local communication. WiFi communication is further split into AP-mode and EZ-mode (e.g., SmartConfig [15]). EZ-mode accounts for only a small fraction (8% of IoT apps support this mode [19]) and involves special app-generated 802.11 frames. Most WiFi-based setups involve the device starting in AP-mode with a DHCP server to assign an IP address to a connected smartphone, establishing communications between the app and device.
Remote binding: In this state, both the Device_ID and the user's unique account information must be submitted to the cloud for registration to ensure that each device instance is correctly linked with the user account, and the cloud maintains state to reflect this mapping relationship.
Operation: The *setup* process transitions from the local configuration to the operation state, where the device is fully configured and in use. Suppose a user clicks a smart camera app's "ON" button to start recording, and the command is issued via the cloud server. The corresponding commands include the Device_ID to specify which device to act on. We call such a request/response cycle *remote command*. Often, a remote command to reset the device (*remote reset*) is also supported to force a transition back to the *local configuration* state. Based on our observation, many devices share Device_ID (even if hard-coded) with apps during *local communication* to ensure the apps have access to the Device_ID.
Threat Model. Despite the unpredictable timing, brief process, etc, in the setup phase, particular threat models are still within the realm of possibility. For example, an opportunistic attacker passively and continuously sniffs OTA WiFi and BLE traffic across all channels in a high IoT device density (e.g., apartment complex) environment. Given the popularity of IoT smart home devices,

they can, after some time, witness the setup of a new device. Non-opportunistic attacks are also possible, assuming access to side-information informing them of the purchases/acquisition of a new IoT device by a user which is reasonably expected to be set up soon afterwards. The attackers may not have prior knowledge of the users' home environments (e.g., device models). Therefore, they cannot conduct real-time attacks like man-in-the-middle (MITM). However, they can infer information about the introduction of a new platform and device (e.g., based on MAC address, AP-Mode SSID name, etc.) using the sniffed data. If their aim is to harvest sensitive information like IDs, users' home WiFi credentials, etc. from the sniffed *local communication*, then it aligns with the SI we have shown to be exposed. Subsequently, with the SI extracted, after the devices' setup finishes, attackers take advantage of the platform state design flaws remotely, enabling at least three categories of attacks, *remote device hijacking, command injection and phantom devices* [7, 32] discussed in Section 5.

3 Methodology

Our methodology is built on answering the research questions outlined in Section 1. Our approach applies to IoT devices which users set up using smartphone apps without the aid of input interfaces like keyboards or touchscreens. We assume the accompanying app can be identified using the device's MAC address and downloaded from the marketplace[1]. Fig. 1 shows the high-level overview of our workflow, beginning with the raw data sniffed OTA during setup and producing exposed and verified sensitive information.

Data Collection. We assess the setup phase of 20 devices setup using 14 IoT apps. We follow the instructions provided with each IoT device for the standard setup procedure on the Android platform. Simultaneously, we set a sniffer to capture the OTA traffic data of the app device *local communication*, producing the *monitor dataset*. The sniffer selection depends on the chosen device and its setup method. For WiFi-based setups, we use a BCM4387 network adapter in monitor mode, and for BLE, we run a Nordic nrf52840 board [27]. The *monitor dataset* contains all the WiFi or Bluetooth frames captured over the air, at Layer 2 and all layers on top encapsulated in the frames, during the local configuration. We capture a wide range of 802.11 frames for WiFi, including Beacon, RTS, CTS, ACK, and Probe. For devices not applying link layer encryption, like WPA2, higher layer headers, protocols (e.g., TCP, UDP, HTTP, etc) and application payload information are also available. For BLE capture, the frame data includes advertisements, connection requests, etc. When no link layer encryption is applied, we check the Attribute Protocol (ATT) of the application-layer payload exchange.

Once the device has completed local configuration, we begin to collect data at the home-AP, a Raspberry Pi 4 Model B running `hostapd` [21] and `netsniff-ng` [22] for packet capture. This AP also enables app-cloud communication and produces the *AP dataset* traffic capture. The *AP dataset* captures Layer 3 and higher (TCP, TLS, etc.) app-cloud communications. All traffic datasets are stored in `.pcap` format.

Binary analysis (RQ1). At step ① (see Figure 1), we begin by inspecting local configuration traffic, as recorded in the monitor dataset, to identify exposed information, i.e., parse the raw data to human-readable format (e.g., ASCII, JSON) using standard tools like `Wireshark` and `Tshark`. For serialized data not directly readable (e.g., compressed by `gzip`), we apply `Binwalk` [26] to find potential headers in the payload. `Binwalk`'s limitation of not recognizing chunks that lack magic value headers is handled using protocol-specific tools like `protoscope` [24]. Once the encoding format is identified, we use Python scripts to decode the raw data using the identified headers. We refer to binary data that can be parsed/decoded using binary analysis alone as *beaconing* data and proceed to sensitivity analysis. For traffic where we cannot create complete human-readable information, we move to Android Package Kit (APK) analysis.

APK Analysis (RQ2). We extract additional information from the setup traces by performing APK analysis on the app through the following steps.

APK Decompilation: We use `jadx` [17] to decompile the `Dalvik` byte-code stored in the APK back into a high-level Java/Kotlin code. Many Android apps utilize a compiled native code library (.so) by invoking Java Native Interface (JNI) to prevent decompilation. We check for such safeguards using `IDA-pro` [14] to disassemble the native code.

Locating Functions: In many consumer IoT devices' apps, it is challenging to isolate the app code critical to *local communication*. Also, over half of the tested vendors performed APK obfuscation to hinder understanding the decompiled code (e.g., ProGuard obfuscation [13]). Thankfully, most of the tested IoT devices send APK hard-coded imprints (e.g., UUIDs, device models) to the app during *local communication*. We use imprints identified in step ① and static taint analysis to guide us to the location of critical code snippets. We use `FlowDroid` [3] to mark the hard-coded imprints as the *taint source* and record function calls it processes. We consider the corresponding function calls as highly involved in network traffic exchanges or cryptographic usage as *taint sink*. Step ② shows an exposed model name being used to find the relevant cryptographic function in the APK. We focus on this final set of pertinent code snippets, which define the code logic (e.g., libraries, crypto). Fig. 2 shows an example of pseudocode for a relevant function, where we uncover the logic used by the app for decrypting messages using `device_model`.

Key reproduction: When investigating relevant code snippets, we uncover two situations: the data is encoded by customized protocols (e.g., SmartConfig [15]) or protected by encryption. In the former case, parsing non-beaconing data back to human-readable text is less challenging since the decompiled APK exposes the encoding/decoding structure. Allowing the authoring of a script, albeit a custom one, to decode the setup traffic. In the case of encrypted data, we attempt to extract the secret key. Using only the app and setup traces, we identify two situations where it is possible to decrypt the traffic. Either the key is hard-coded in the APK or derived from known information. Fig. 2 presents an example, where the `device_model` appended to an APK hard-coded magic string is used to generate a key to guard subsequent communications. Analyzing the decompiled code, we can discover this logic and easily recreate the same key, *without* access to the IoT device, essentially evaluating whether the implemented cryptographic scheme is robust. We combine the information successfully decrypted/decoded from both step ① and step ② to form the *exposed info*.

[1]We use Android as our app platform because it is open source, but the high-level methodology can be extended to other smartphone operating systems.

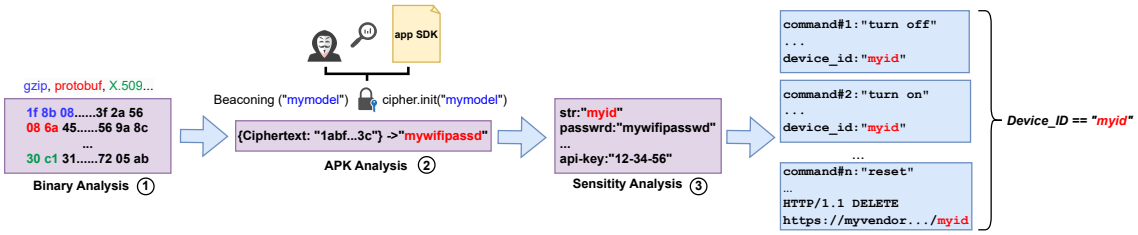

Figure 1: Assessment workflow; traces are combined with APK analysis to identify exposed information.

Sensitive information identification (RQ3). Finally, we explore the information identified and extracted from analysis of the monitor dataset at the previous steps, to determine how sensitive it is. Credential information, e.g., home-AP WiFi credentials, app's user name and password, is self-explanatory and provable based on the information entered during the setup. Additional sensitive information can be challenging to infer based on plaintext headers alone. For example, device information, model, and MAC address vary among vendors and may not be used throughout the platform. Yet, we have to establish if any of this information has any consequence on the operation of the platform. Because we cannot access device firmware, we follow a Device_ID verification method based on the observation that Device_ID is transmitted from the app to the cloud as part of *remote commands*. Thus, we can verify the Device_ID by finding common information between the transmitted *remote commands* and the exposed information collected in steps ① & ②. This process is highlighted in step ③, where the string "myid" occurs in the setup information and every device command.

When searching for the Device_ID, we look for a fixed persistent value. Consequently, no matter which *remote command* is transmitted (e.g., reset or turn on/off device), the Device_ID is assumed to be attached to each message. For our verification process, we want to compare the exposed info across different types of *remote commands* to see if there are repeated matches. Hence, once the local configuration has finished and the device entered normal operation, we will study the AP dataset and attempt to bypass the security protocol protection (e.g., TLS) of the traffic exchanges between the app and the cloud. We develop the following two approaches to intercept the *remote commands*. Note that since Construct_ID contributes to the formation of the Device_ID, it is possible to verify its presence only after the whole Device_ID generation exchanges have been observed.

Traffic Interception. We use `mitmproxy` [8] to set up a proxy to intercept traffic exchanges between an app and the cloud during the *operation state*. We use a rooted smartphone and move the proxy's certificate from the users' trust store to the system trust store. This process causes the app to trust the proxy's certificate; however, many tested IoT platforms have pre-trust certificates (i.e., they only accept public keys or hostnames from specific servers' certificates). Based on our observation, IoT apps implement such certificate pinning via standard libraries (e.g., *com.android.org.conscrypt.TrustManagerImpl*). To get around this, we use a dynamic code instrumentation tool `Frida` [10], which allows injecting binaries into an ongoing process to remove the certificate pinning functions at runtime [28]. Fig. 3 shows the Device_ID verification process using traffic interception for a sample device. The top half of the figure shows partial traffic traces when *remote command* is triggered, e.g., switching ON a camera. The bottom half shows part of beaconing data sniffed OTA during *local communication*. The string "CforkLLzBQeCIP==" is common to both and thus confirmed as the Device_ID.

```
func decrypt_message(http_response)
    byte[] cipher_bytes = http_response.payload.toByte();
    String device_model = "D8123456EC";
    String magicString = "magic";
    String keyStr = MD5(device_model + magicString);
    ...
    SecretKeySpec privateKey = generateSecret(keyStr);
    Cipher cipher = Cipher.getInstance("AES/CBC/...");
    cipher.init(Cipher.DECRYPT_MODE, privateKey, IV);
    String plaintext = cipher.doFinal(cipher_bytes);
    ...
```

Figure 2: Example identified "setup" crypto function.

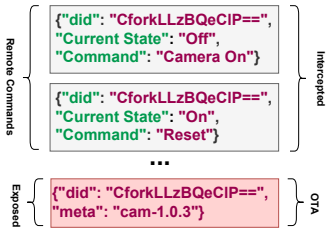

Figure 3: Sample device Device_ID verification.

Traffic Hooking. We additionally employ function-level network traffic inspection, i.e., *traffic hooking*, to uncover potential *remote commands* before they are encrypted. We leverage the knowledge that Android (since v7.0) usually handles HTTPS traffic data via the native functions, e.g., *java.net.SocketOutputStream.socketwrite0* and *com.android.org.conscrypt.ConscryptFileDescriptorSocket$SSLOutputStream*, then modify tool `r0capture` [25] to hook into these functions and inspect the data streams they handle. We use *traffic hooking* as a backup heuristic analysis when encountering limitations using traffic interception alone. The reader can check Table 2 in Appendix B for details of *remote commands* uncovered methods for each app.

4 Evaluation

We provide the evaluation results of the workflow introduced in the previous section using data collected from 20 devices and their

Table 1: Evaluation summary of studied devices.

● WiFi Setup only ● BLE Setup only ◐ WiFi and BLE Setup Support ● Out-of-band Setup (b) Beaconing (nb) Not Beaconing (via APK analysis)

(Device,App)	Architecture	Layer2 Encryption	Layer5 Encryption	ID Exposed	Credentials Exposed
● (D#1, A#1)	CITM	✗	✗	Device_ID (b)	WiFi(b), Account Credentials (b)
● (D#2, A#1)	CITM	✗	✗	Device_ID (b)	WiFi(b), Account Credentials (b)
● (D#3, A#2)	CITM	WPA2	○	○	○
● (D#4, A#2)	CITM	WPA2	○	○	○
● (D#5, A#3)	CITM	WPA2 (Break)	✗	Device_ID (b)	WiFi (b), LAN Remote Commands (b)
● (D#6, A#4)	CITM	WPA2 (Break)	✗	Device_ID (b)	WiFi (b)
● (D#7, A#5)	CITM	✗	AES (Break)	Device_ID (b)	WiFi(nb)
● (D#8, A#6)	CITM	✗	TLS	Device_ID (b)	-
● (D#9, A#7)	CITM	✗	AES	Partial Construct_ID (b)	-
● (D#10, A#8)	CITM	✗	AES (Break)	Device_ID (b)	WiFi (nb)
● (D#11, A#9)	CITM	✗	J-PAKE	-	-
● (D#12, A#9)	CITM	✗	SSL	-	-
◐ (D#13, A#9)	CITM	✗	SSL	Device_ID (b) (●), - (●)	- (◐)
◐ (D#14, A#9)	CITM	✗	SSL	Device_ID (b) (●), - (●)	- (◐)
● (D#15, A#10)	CITM	✗	SSL	Construct_ID (b)	-
● (D#16, A#11)	AITM	✗	✗	N.A.	-
● (D#17, A#12)	AITM	LE Security	○	N.A.	○
● (D#18, A#13)	CITM	N.A.	N.A.	○	○
● (D#19, N.A.)	Not Cloud	✗	✗	N.A.	WiFi (b)
● (D#20, A#14)	CITM	✗	✗ (encoded)	Device_ID (nb)	WiFi (nb)

✗: No encryption ○: Not allowed / Need for further study (e.g., no suspect info exposed) -: Attempted / Not found N.A.: Not applicable

corresponding apps. Some devices use the same app. Table 1 summarizes the results, highlighting if encryption was used, at what layer (L2/L5), and the information leakage observed. Among the 20 tested devices, two ((D#16, D#17)) are AITM devices while one (D#19) is not cloud-enabled. The remaining 17 devices are CITM platforms. In summary, from the set of 20 devices, 13 expose at least one piece of defined *sensitive information*.

Safe Devices. We could not extract sensitive information from seven of the devices we tested (D#3, D#4, D#11, D#12, D#16, D#17 D#18). The information shared by both WiFi devices, D#3 and D#4, is protected using WPA2 as L2 encryption. Both passphrases are encoded into QR codes, which are pre-shared to the app before *local communication* is established. For each device, we verify that the passphrases are not APK hard-coded and are difficult to brute force. While D#11 and D#12 have no L2 encryption, they are instead protected by key exchange-based encryption. D#11 uses J-PAKE with the shared password exchanged through QR code scanning, and D#12 uses TLS/SSL. To confirm this, we observe handshakes and device certificates in decoded traffic. We also find an API call such as `java.security.KeyStore` by statically analyzing the app, with the private key stored in the private keyStore [12]. D#17 is the only BLE-based device with L2 encryption (through LE Secure Connections [4]). Key exchanges are introduced in BLE 4.2 to prevent long-term key exposure, which we will consider unbreakable via passive sniffing [4].

The final two safe devices do not fit our typical model, and their information is protected using alternate means. D#16 is not a CITM device and does not rely on an ID for coordination across the platform. It only communicates with its app and does not share any credentials. D#18 is unique because it has no explicit wireless network connection during the setup phase. The app encodes the home AP credentials into a QR code which the camera scans, i.e., it uses an out-of-band channel to transfer the credentials.

Sensitive Information Leaking Devices. The remaining 13 devices expose at least one type of sensitive information. Note that we focus only on IDs that are shared throughout the platform, i.e., meaningful for CITM devices. Because a Construct_ID may be formed by multiple information fields (e.g., MAC address and device serial number together), we only record the Construct_ID as exposed when all the fields are obtainable. Briefly, eleven devices expose IDs, eight devices expose credentials, and six devices expose both.

ID Exposure. We identify the ID (Device or Construct) for 11 of the devices. For all identified Device_IDs, we verify that they subsequently appear in *remote commands*.

Two devices hard-code their Device_ID (D#8) or partial Construct_ID (D#9) as a suffix of the SSID used at setup. The SSID is broadcast when the device presents as an AP, exposing the ID even when subsequent communication is encrypted. Five devices (D#1, D#2, D#10, D#13 (when setup via WiFi), and D#14 (when setup via WiFi)) expose their Device_ID during setup as human readable plaintext. Although D#13 and D#14 are marked as exposing their Device_ID, we could identify this to be the case only when setup via WiFi, although they both support setup via BLE as well.

Despite using encryption, after performing binary analysis, D#5 and D#15 expose their Device_ID. The L2 encryption (WPA) used by D#5 has a weak passphrase, which is easily broken by brute force or consulting online resources. D#15 exposes a Construct_ID before switching to encrypted traffic (TLS). Pre-handshake setup traffic is shared as gzip compressed data containing `cloud_device_id`, `certificates`, and `device_name`. Once decoded, these three fields are sufficient to produce the Device_ID.

With the APK inspection, we expose the Device_ID of two more devices (D#7 and D#20). Both use custom protocols, which we can decode after inspecting the APK. Specifically, D#20 utilizes the SmartConfig framework [15] along with a custom encoding.

Credentials Exposure. After binary analysis, five devices (D#1, D#2, D#5, D#6, D#19) were found to expose at least one piece of credential information during setup. D#1 and D#2 transmit both WiFi credentials and IoT app login credentials as plaintext. D#6 uses the same weak L2 encryption scheme as D#5, resulting in WiFi credentials being exposed for both. Additionally, during the *operation*

state, D#5 uses AES to protect commands issued while the app is on the same network as the device. However, an exposed field called `api-key` is hashed and used as the secret key. We verify this by recreating the same key and forcing D#5 to operate as commanded (power on, reset, etc.). D#19 also leaks WiFi credentials because it is configured as a captive portal using plaintext HTTP. D#7, D#10, and D#20 expose WiFi credentials with the knowledge obtained from their APK. For D#20, the WiFi credentials are available along with the Device_ID after discovering the encoding. Both D#7 and D#10 use symmetric encryption to protect the home-AP WiFi credentials; however, the encryption keys for both are derivable. D#7 uses an MD5 hash of the `token`, which is initially transmitted in plaintext as the private key. Comparably, D#10 creates its private key using `metainfo` (sends from device to app as plaintext initially) and an APK hard-coded string.

Other Information Exposure. In addition to the aforementioned information, we uncovered more identifiable information (presented in Appendix C) from 19 of the devices, currently not categorized as being sensitive but potentially useful for additional tasks, e.g., device fingerprinting.

Take Away and Mitigation. To protect *local communication*, L2 encryption is effective and practical approach to shield higher-layer payloads and protocols. For AP-based devices, a good practice is using an out-of-band channel (QR code, printed WPA2 passphrase, etc.) for , assuming the private key is unique for each device. The adoption of LE Secure Connections is another reliable protection for BLE devices. For application-level encryption, asymmetric cryptography key exchange (e.g., as in TLS) is preferred, and if symmetric encryption is used, the key must not be derivable from previously transmitted information.

Discussion. Our validation method for inferring the Device_ID is an empirical analysis by considering multiple different *remote commands*; however, the most straightforward verification method is a direct examination of the cloud source code. Without the knowledge of the cloud-based code and following our assumptions of inaccessible firmware, we are limited to app-side APK inspection. Further, we also observed some communication between devices and cloud after initial setup without using apps, which we suspect also contains identifying information. In future work, we will explore connections between this traffic and our analysis to fully automate a sequence of tools following our methodology to check the security of a large number of IoT platforms through the analysis of APK. The tool would automate time-consuming processes allowing "grading" of IoT platforms on the basis of properties and characteristics the apps reveal about those platforms.

5 Related Work

Platform-specific security: A body of work is not specific to the setup phase but specific to an IoT platform. Several research groups [11, 18] focus on the Amazon IoT platform, where they check device-level (using firmware/SDK reverse engineering) or application-level (e.g., *skills*) vulnerabilities. For the same platform, Iqbal *et al.* check the process of users' privacy data collection [16]. Samsung's SmartThings platform is analyzed in [5, 29], where the former develops an app *taint analysis* tool, and the latter extracts security policies from IoT apps. We assume no access to the app source code, leaving only the option to analyze the apps' operation. We follow a general approach across IoT platforms, assuming we do not have source code or SDKs provided by the IoT platform vendors. Equally, we assume no knowledge about the device firmware.

Vulnerabilities due to exposed identity: Three general types of vulnerabilities have been reported in the literature, which also apply to our study: *(a) Remote Device Hijacking* [7, 32] enables a (remote) attacker to assume control of a device, by establishing a binding ahead of the legitimate user during the *remote binding* state, to associate the victim's Device_ID with the attackers' account. *(b) Command Injection* [7, 32] allows an attacker to execute actions in a device by constructing and sending unauthorized commands using the victim's Device_ID in the *operation* state. A special case of this attack is to simulate an illegitimate remote reset. *(c) Phantom Device Construction* [32] allows an attacker to introduce a simulated device and inject fake sensor data and intercept commands, and depending on the specific type of vulnerabilities can register the device to the cloud, obtaining a corresponding Device_ID, and/or bind the Device_ID with the attacker's account.

Our work is nearest to [32], [7], and [9]. Zhou *et al.* [32] develop a state model that confirms all investigated platforms do not strictly guard the validity of the involved state transitions for a number of conjectured reasons. Chen *et al.* [7] provide an analysis of the downstream effects, assuming an attacker can access sensitive information. Instead, we focus on whether such sensitive information can be acquired in the first place, evaluating which devices are more vulnerable to sensitive information disclosure. Fezeu *et al.* [9] dissect a new setup design flaw, assuming that the setup phase can reveal sensitive information. This is consistent with our approach, and also methodologically related insofar as app side analysis is concerned. However, compared to three IoT vendors studied in [9], we provide results for a diverse set of devices using a general and widely applicable methodology.

6 Conclusion

We have examined the setup phase of 20 representative smart home IoT devices for potential information leakage. We developed a semi-automated general methodology to assess if, and what is the nature, of the information leaked during setup. The results show that two-thirds of the devices expose some form of sensitive information. The results are alarming given that the devices chosen are commonplace off-the-shelf devices available through online marketplaces – likely to be owned by thousands of customers. Future work will focus on fully automating the testing methodology we have introduced.

The proposed methodology will also benefit developers, vendors, and researchers by applying strict criteria for accepting new IoT devices as secure and for rectifying security issues of existing ones. The acquired information from the setup phase may be used for elaborate end-to-end attacks that are part of our future work. End-to-end proof-of-concept attacks may require the involvement of cloud services provided by vendors, possibly impacting many users, thus presenting interesting practical and ethical issues on how they can be studied. Further refinement of the methodology described in this paper may allow the identification of more sensitive information being leaked.

References

[1] Anand Agrawal and Rajib Ranjan Maiti. 2023. iTieProbe: Is Your IoT Setup Secure against (Modern) Evil Twin? https://doi.org/10.48550/ARXIV.2304.12041 Publisher: arXiv Version Number: 2.

[2] Omar Alrawi, Chaz Lever, Manos Antonakakis, and Fabian Monrose. 2019. SoK: Security Evaluation of Home-Based IoT Deployments. In *2019 IEEE Symposium on Security and Privacy (SP)*. 1362–1380. https://doi.org/10.1109/SP.2019.00013

[3] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. *ACM SIGPLAN Notices* 49, 6 (June 2014), 259–269. https://doi.org/10.1145/2666356.2594299

[4] Bluetooth Special Interest Group. 2014. Bluetooth Core Specification Version 4.2. https://www.bluetooth.com/specifications/specs/core-specification-4-2/

[5] Z. Berkay Celik, Leonardo Babun, Amit Kumar Sikder, Hidayet Aksu, Gang Tan, Patrick McDaniel, and A. Selcuk Uluagac. 2018. Sensitive Information Tracking in Commodity IoT. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 1687–1704. https://www.usenix.org/conference/usenixsecurity18/presentation/celik

[6] Jiongyi Chen, Menghan Sun, and Kehuan Zhang. 2019. Security Analysis of Device Binding for IP-based IoT Devices. In *2019 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*. 900–905. https://doi.org/10.1109/PERCOMW.2019.8730580

[7] Jiongyi Chen, Chaoshun Zuo, Wenrui Diao, Shuaike Dong, Qingchuan Zhao, Menghan Sun, Zhiqiang Lin, Yinqian Zhang, and Kehuan Zhang. 2019. 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)*. IEEE, Portland, OR, USA, 222–233. https://doi.org/10.1109/DSN.2019.00034

[8] Aldo Cortesi, Maximilian Hils, Thomas Kriechbaumer, and contributors. 2010–. mitmproxy: A free and open source interactive HTTPS proxy. https://mitmproxy.org/ [Version 10.3].

[9] Rostand A. K. Fezeu, Timothy J. Salo, Amy Zhang, and Zhi-Li Zhang. 2023. Dissecting IoT Device Provisioning Process. http://arxiv.org/abs/2310.14125 arXiv:2310.14125 [cs].

[10] Frida. 2012. Frida: A dynamic instrumentation toolkit for developers, reverse-engineers, and security researchers. https://github.com/frida.

[11] Dennis Giese and Guevara Noubir. 2021. Amazon echo dot or the reverberating secrets of IoT devices. In *Proceedings of the 14th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec '21)*. Association for Computing Machinery, New York, NY, USA, 13–24. https://doi.org/10.1145/3448300.3467820

[12] Google. 2023. KeyStore (Java Platform SE). https://developer.android.com/reference/java/security/KeyStore.

[13] Google. 2024. Shrink, obfuscate, and optimize your app. https://developer.android.com/build/shrink-code.

[14] Hex-Rays SA. 2008. IDA Pro. https://hex-rays.com/ida-pro/.

[15] Texas Instruments. 2019. provisioning_smartconfig README. https://software-dl.ti.com/ecs/CC3200SDK/1_5_0/exports/cc3200-sdk/example/provisioning_smartconfig/README.html.

[16] Umar Iqbal, Pouneh Nikkhah Bahrami, Rahmadi Trimananda, Hao Cui, Alexander Gamero-Garrido, Daniel J. Dubois, David Choffnes, Athina Markopoulou, Franziska Roesner, and Zubair Shafiq. 2023. Tracking, Profiling, and Ad Targeting in the Alexa Echo Smart Speaker Ecosystem. In *Proceedings of the 2023 ACM on Internet Measurement Conference*. ACM, Montreal QC Canada, 569–583. https://doi.org/10.1145/3618257.3624803

[17] Jadx. 2004. jadx: Dex to Java decompiler. https://github.com/skylot/jadx.

[18] Christopher Lentzsch, Sheel Jayesh Shah, Benjamin Andow, Martin Degeling, Anupam Das, and William Enck. 2021. Hey Alexa, is this Skill Safe?: Taking a Closer Look at the Alexa Skill Ecosystem. In *Proceedings 2021 Network and Distributed System Security Symposium*. Internet Society, Virtual. https://doi.org/10.14722/ndss.2021.23111

[19] Changyu Li, Quanpu Cai, Juanru Li, Hui Liu, Yuanyuan Zhang, Dawu Gu, and Yu Yu. 2018. Passwords in the Air: Harvesting Wi-Fi Credentials from SmartCfg Provisioning. In *Proceedings of the 11th ACM Conference on Security & Privacy in Wireless and Mobile Networks*. ACM, Stockholm Sweden, 1–11. https://doi.org/10.1145/3212480.3212496

[20] Hui Liu, Juanru Li, and Dawu Gu. 2020. Understanding the security of app-in-the-middle IoT. *Computers & Security* 97 (Oct. 2020), 102000. https://doi.org/10.1016/j.cose.2020.102000

[21] Jouni Malinen. 2013. hostapd: IEEE 802.11 AP, IEEE 802.1X/WPA/WPA2/WPA3/EAP/RADIUS Authenticator. https://w1.fi/hostapd/.

[22] The netsniff-ng team. 2013. netsniff-ng. https://github.com/netsniff-ng/netsniff-ng.

[23] Oberlo. 2024. US Smart Home Statistics (2019–2028). https://www.oberlo.com/statistics/smart-home-statistics.

[24] Protoscope. 2022. Protoscope: An interactive tool for analyzing protocol buffers. https://github.com/protocolbuffers/protoscope.

[25] R0capture. 2020. r0capture: A universal SSL/HTTPS interception for most Android applications. https://github.com/r0ysue/r0capture.

[26] ReFirmLabs. 2015. Binwalk: Firmware Analysis Tool. https://github.com/ReFirmLabs/binwalk.

[27] Nordic Semiconductor. 2019. nRF Sniffer for 802.15.4. https://github.com/NordicSemiconductor/nRF-Sniffer-for-802.15.4.

[28] Maurizio Siddu. 2020. Frida multiple unpinning. https://gist.github.com/akabe1/5632cbc1cd49f0237cbd0a93bc8e4452.

[29] Yuan Tian, Nan Zhang, Yueh-Hsun Lin, XiaoFeng Wang, Blase Ur, Xianzheng Guo, and Patrick Tague. 2017. SmartAuth: User-Centered Authorization for the Internet of Things. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 361–378. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/tian

[30] Mengmei Ye, Nan Jiang, Hao Yang, and Qiben Yan. 2017. Security analysis of Internet-of-Things: A case study of august smart lock. In *2017 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, Atlanta, GA, 499–504. https://doi.org/10.1109/INFCOMW.2017.8116427

[31] Yiwei Zhang, Siqi Ma, Tiancheng Chen, Juanru Li, Robert H. Deng, and Elisa Bertino. 2024. EvilScreen Attack: Smart TV Hijacking via Multi-Channel Remote Control Mimicry. *IEEE Transactions on Dependable and Secure Computing* 21, 4 (2024), 1544–1556. https://doi.org/10.1109/TDSC.2023.3286182

[32] Wei Zhou, Yan Jia, Yao Yao, Lipeng Zhu, Le Guan, Yuhang Mao, Peng Liu, and Yuqing Zhang. 2019. 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)*. USENIX Association, Santa Clara, CA, 1133–1150. https://www.usenix.org/conference/usenixsecurity19/presentation/zhou

[33] Qingsong Zou, Qing Li, Ruoyu Li, Yucheng Huang, Gareth Tyson, Jingyu Xiao, and Yong Jiang. 2022. IoTBeholder: A Privacy Snooping Attack on User Habitual Behaviors from Smart Home Wi-Fi Traffic. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 7, 1 (March 2022), 1–26. https://doi.org/10.1145/3580890

[34] Chaoshun Zuo, Haohuang Wen, Zhiqiang Lin, and Yinqian Zhang. 2019. 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*. ACM, London United Kingdom, 1469–1483. https://doi.org/10.1145/3319535.3354240

A Ethics

This study has three main ethical considerations. First, this study involves in-lab network traffic interception between popular smart home apps and IoT cloud services, which may expose sensitive API calls and give potential adversaries a chance to issue forged commands to the cloud, harm the interests of legitimate users, etc. Second, the developed scripts decrypt the collected traffic from *local communication*. Thus, network traces of specific vendors may allow adversaries to decrypt the sniffed traffic to access WiFi credentials, Device_ID, and more. Hence, we do not provide open access to either one of them but are willing to share privately with research groups. Because the traces contain sensitive information about our devices, we commit to keeping them in the lab until they are destroyed. Also, we do not actively forge messages to the cloud to avoid any potential impact on regular users. Finally, potential attackers may explore the security weakness reported here. We thus anonymize device and vendor information in a way that reduces the ease of reproducibility of our results and also does not single out particular vendors. In addition, we contacted each vendor whose devices were found to exhibit at least one vulnerability and reported our findings, allowing them to develop mitigation measures.

B Sensitive Information Verification Details

Table 2 summarizes devices and their apps along with the pinning APIs we identified. We test 11 apps out of 14 (the rest do not expose information during *local communication*). App A#4 does not utilize TLS/SSL-based communications, instead using a customized

encrypted UDP-based traffic for *app-cloud* communications. The remaining ten TLS/SSL-based apps apply certificate pinning. The most common pinning API, *com.android.org.conscrypt.TrustManagerImp*, is used for all tested apps. We bypass the certificate pinning for eight apps. We verify the bypass by the app's ability to connect to the Internet without certificate failure exceptions. Intercepting the traffic of two apps (A#1 and A#8) fails as the identified pinning APIs cannot fully cover the entire pinning technique used by these vendors. In these cases, the vendor obfuscates the pinning APIs to protect them from tests like ours, i.e., their apps do not trust our proxy's certificate.

Bypassing certificate pinning does not always allow a means to extract fully understandable plaintext, as we find vendors utilize additional levels of serialization, e.g., `protobuf`, encryption like RC4, or even unknown vendor-specific raw data that requires deeper protocol/crypto analysis. Still, some can be (partially) interpreted to a degree that yields to our sensitive information analysis. For example, D#20 has a high percentage of additional encryption/encoding on the intercepted application payload that we cannot decode/decrypt. In this case, we must utilize *traffic hooking* to verify sensitive information. Overall, we perform *traffic hooking* verification for four apps (A#1, A#4, A#8, and A#14). We use ◯ in Table 2 to refer to app-device pairs that have no suspected information exposure during *local communications*, preventing the need for further *remote command* analysis. Finally, devices with no meaningful cloud-based (or TLS/SSL-based) communication are referred to as N.A.

C Other Exposed Information

Table 3 summarizes additional exposed info during device setup. In most cases, there is other self-explainatory information or information that is easy to identify by accessing online resources like vendor documents. The second column of the table displays the potential fingerprint information each tested device exposes, with 14 devices revealing their device model and one device revealing its device type during setup. The device model (fingerprinting) has potential security implications since devices under the same model type have similar network traffic behaviors even if the traffic is encrypted, enabling inferences about a device owner's activities [33]. Specially, we find that all devices which support BLE setup (D#11 - D#17) report a device_model as `SCAN_RSP` on advertising channels before *local communication*, presumably for setup convenience. In addition, although D#3 and D#4 have relatively secure *local communication*, they nevertheless hard-code device_model in the SSID of the device-AP. The remaining five devices (D#1, D#2, D#6, D#7, D#10) report their device model during their *local communication* to the app.

Some devices send metadata to their apps, including firmware version, status, supported protocols, and certificates. Additionally, there's "side info" not directly related to the devices but may still be of interest, such as location data and lists of nearby access points. Among those, the most notable exposure we observed is the server hostname that the app transmits to the device. This suggests that the device may not have pre-configured trust anchors for cloud servers built into their firmware. Instead, they might depend on the app to specify which servers to trust during its setup. If confirmed, this could increase the risk of MITM attacks in device-cloud communications. To further validate this analysis, a study of the firmware would be necessary.

Finally, some devices leak information for which we have not ascertained its usage during setup. We mark it as ✓ on the corresponding rows of the table. This category of exposed information is also potentially sensitive after further analysis beyond our current methodology.

D GENERATIVE AI ACKNOWLEDGMENTS

We used ChatGPT to generate LaTeX formatted tables.

(Device,App)	Pinning APIs Addressed	Successful Bypass
(D#1, A#1)	javax.net.ssl.SSLContext com.android.org.conscrypt.TrustManagerImpl okhttp3.CertificatePinner	✗
(D#2, A#1)	javax.net.ssl.SSLContext com.android.org.conscrypt.TrustManagerImpl okhttp3.CertificatePinner	✗
(D#3, A#2)	◯	◯
(D#4, A#2)	◯	◯
(D#5, A#3)	com.android.org.conscrypt.TrustManagerImpl javax.net.ssl.X509TrustManager	✓
(D#6, A#4)	N.A.	N.A.
(D#7, A#5)	javax.net.ssl.X509TrustManager com.android.org.conscrypt.TrustManagerImpl okhttp3.CertificatePinner	✓
(D#8, A#6)	com.android.org.conscrypt.TrustManagerImpl sdk.pendo.io.m.e.a (obfuscated)	✓
(D#9, A#7)	com.android.org.conscrypt.TrustManagerImpl okhttp3.CertificatePinner	✗
(D#10, A#8)	com.android.org.conscrypt.TrustManagerImpl	✗
(D#11, A#9)	com.android.org.conscrypt.TrustManagerImpl okhttp3.CertificatePinner javax.net.ssl.X509TrustManager apache.http.conn.ssl.AbstractVerifier	✓
(D#12, A#9)	com.android.org.conscrypt.TrustManagerImpl okhttp3.CertificatePinner javax.net.ssl.X509TrustManager apache.http.conn.ssl.AbstractVerifier	✓
(D#13, A#9)	com.android.org.conscrypt.TrustManagerImpl okhttp3.CertificatePinner javax.net.ssl.X509TrustManager apache.http.conn.ssl.AbstractVerifier	✓
(D#14, A#9)	com.android.org.conscrypt.TrustManagerImpl okhttp3.CertificatePinner javax.net.ssl.X509TrustManager apache.http.conn.ssl.AbstractVerifier	✓
(D#15, A#10)	com.android.org.conscrypt.TrustManagerImpl	✓
(D#16, A#11)	com.android.org.conscrypt.TrustManagerImpl javax.net.ssl.X509TrustManager	✓
(D#17, A#12)	◯	◯
(D#18, A#13)	◯	◯
(D#19, N.A.)	N.A.	N.A.
(D#20, A#14)	com.android.org.conscrypt.TrustManagerImpl okhttp3.CertificatePinner	✓

✓: Successful Bypass
✗: Failed Bypass
◯: Not allowed / Not possible to identify (e.g., no suspect info exposed)
NA: Not Applicable

Table 2: Summary of IoT devices and their certificate pinning status.

(Device,App)	Fingerprinting Information	Device Meta	Side Info	Extra Info
(D#1, A#1)	device model;	-	nearby AP list, location, server hostname;	✓
(D#2, A#1)	device model;	-	nearby AP list, location, server hostname;	✓
(D#3, A#2)	device model;	-	-	-
(D#4, A#2)	device model;	-	-	-
(D#5, A#3)	-	chipid;	server hostname;	-
(D#6, A#4)	device model;	device parameters (e.g., supported voltage);	-	-
(D#7, A#5)	device model;	device firmware version, bootloader version, memory usage statistics;	nearby AP list, location;	✓
(D#8, A#6)	-	device certificates;	-	-
(D#9, A#7)	-	device firmware version;	encrypted session;	-
(D#10, A#8)	device model;	device firmware version;	nearby AP list, icon images;	-
(D#11, A#9)	device model;	device certificates;	-	-
(D#12, A#9)	device model;	device certificates;	-	-
(D#13, A#9)	device model;	device certificates;	nearby AP list;	✓
(D#14, A#9)	device model;	device certificates;	nearby AP list;	✓
(D#15, A#10)	device model;	supported languages, device firmware version, build version;	-	-
(D#16, A#11)	device model;	device firmware version;	-	✓
(D#17, A#12)	device model;	-	-	-
(D#18, A#13)	-	-	-	-
(D#19, N.A.)	device type;	-	captive portal meta info (HTML code);	-
(D#20, A#14)	-	-	-	✓

-: Attempted / Not found ✓: More info extracted / Not validated

Table 3: Other exposed information from the tested IoT platforms.