KINGFISHER: Unveiling Insecurely Used Credentials in IoT-to-Mobile Communications

Yiwei Zhang
Shanghai Jiao Tong University
yyyyyyw@sjtu.edu.cn

Siqi Ma
The University of New South Wales
siqi.ma@adfa.edu.au

Juanru Li
Shanghai Jiao Tong University
mail@lijuanru.com

Dawu Gu
Shanghai Jiao Tong University
dwgu@sjtu.edu.cn

Elisa Bertino
Purdue University
bertino@purdue.edu

***Abstract*—Today users can access and/or control their IoT devices using mobile apps. Such interactions often rely on IoT-to-Mobile communication that supports direct data exchanges between IoT devices and smartphones. To guarantee mutual authentication and encrypted data transmission in IoT-to-Mobile communications while keeping lightweight implementation, IoT devices and smartphones often share credentials in advance with the help of a cloud server. Since these credentials impact communication security, in this paper we seek to understand how such sensitive materials are implemented. We design a set of analysis techniques and implement them in KINGFISHER, an analysis framework. KINGFISHER identifies shared credentials, tracks their uses, and examines violations against nine security properties that the implementation of credentials should satisfy. With an evaluation of eight real-world IoT solutions with more than 35 million deployed devices, KINGFISHER revealed that all these solutions involve insecurely used credentials, and are subject to privacy leakage or device hijacking.**

***Keywords*-IoT-to-Mobile communication, Value-based Analysis, Shared Credential, Companion App**

I. INTRODUCTION

Today many Internet-of-Things (IoT) devices support multiple communication models. In addition to the connection between an IoT device and its cloud backend (IoT cloud, a cloud server maintained by either a vendor or a third-party public cloud provider), many IoT devices are able to directly communicate with smartphones via *peer-to-peer, local network based IoT-to-Mobile communication*. Rather than utilizing the IoT cloud as a portal to send data to the smartphone, an IoT device can leverage the local transmission capabilities to directly communicate with the smartphone when they are connected to the same LAN or PAN (Personal Area Network) [1]. Such a local communication model not only allows end-users to manage IoT devices via a companion app on smartphones with minimal delays. It also allows IoT devices and their associated smartphones to directly exchange privacy-sensitive data, without having to transmit the data via unnecessary third parties (e.g., the cloud server). Therefore, such a communication model reduces security risks arising from cloud-based communications [2], [3], and enhances privacy and compliance with privacy regulations [4]–[6].

Although an IoT-to-Mobile communication is only established between two nearby devices, it still needs to guarantee a secure mutual authentication between these devices and apply strong cryptographic protection to secure the data transmission as attacks are also possible within local networks (e.g., by compromising devices in a Wi-Fi network via a proxy of the remote attacker) [7], [8]. Despite those risks, security and protection schemes for IoT-to-Mobile communication have not been much investigated, as previous research has mostly focused on the security of cloud-centric communication [2], [3], [9]. To address the critical issue of establishing a secure channel for IoT-to-Mobile communication, current commercial solutions typically adopt a credential based security mechanism. Such a mechanism requires to first distribute some credentials to both the IoT device and the smartphone (often with the help of the cloud), and then utilizes these shared credentials for authentication and for establishing a secure channel. There are two types of shared credential (SC) that differ with respect to their use: authentication SC (ASC) used for identity verification, and cryptographic SC (CSC) used for communication encryption. Unlike pre-shared credentials installed into devices by manufacturers, SCs are often dynamically generated and distributed only when an IoT device needs to bind to a smartphone.

SCs are critical security elements for IoT-to-Mobile communication, but our observation is that the use of SCs in IoT-to-Mobile communication is very much ad hoc. They are usually poorly protected and insecurely used. In addition to directly attacking the communications between the IoT device and the smartphone, an attacker may also aim at obtaining the used SCs by either compromising the parties using and storing the SCs, or exploiting the weaknesses of the IoT-cloud assisted SC generation and distribution. However, to the best of our knowledge, no security standard exists to guide the secure use and implementation of SCs. Also, tools designed to analyze the use and implementation of credentials in mobile apps, such as CredMiner [10] and iCredFinder [11], mainly focus on hard-coded credentials, whereas tools such as AuthScope [12] and LeakScope [13] focus on to app-to-cloud communications and do not cover IoT-to-Mobile communication.

To handle the specific scenario of IoT-to-Mobile communication and inspect the security of SCs, we apply a two-fold approach: (i) We establish nine security properties that securely used SCs must/should satisfy; (ii) We design and implement KINGFISHER, an analysis framework to automatically detect violations against these properties.

KINGFISHER analyzes security across the multi-party interactions (among IoT device, smartphone, and cloud) required for the management of SCs to check whether they are securely generated, distributed, used, protected and revoked. KINGFISHER integrates several analysis techniques to identify SCs in use and examine their security posture. It adopts a keywords-guided function instrumentation and network clustering to first collect SC-related messages in IoT-to-Mobile communication, and employs a value-based analysis to identify SCs and track the functions that operate on them in (Android) companion apps. More importantly, KINGFISHER extends traditional analysis techniques that check credentials only in smartphone-to-IoT traffic or only in smartphone-to-cloud network traffic. By simultaneously considering both those two types of traffic and tracking credentials across them, KINGFISHER can better understand the SC distribution process across the IoT device, the smartphone, and the cloud.

To evaluate our approach and investigate how insecurely used SCs affect the IoT security, we used KINGFISHER to analyze eight popular IoT solutions with more than 35 million deployed devices. KINGFISHER successfully identified SC-related functions and messages for all eight solutions despite their implementation diversity. After pinpointing the SCs, KINGFISHER inspected their use and found that none of those solutions securely uses SCs; each of them violated at least four security properties, with the worst case solution only satisfying two out of nine properties. We provide the KINGFISHER details, instructions of our experiments, and the feedback of IoT vendors at https://kingfisher.code-analysis.org.

II. THREATS AGAINST SHARED CREDENTIALS

To systematically study the security threats of SCs, we need to first understand how SCs are managed in IoT-to-Mobile communication. Figure 1 shows the lifetime of SCs. Typically, the lifetime of SC involves three phases (device binding, data transmission, and unbinding) of IoT-to-Mobile communication, and includes two types of SCs based on the two main purposes for using secrets, authentication SC (ASC) and cryptographic SC (CSC). As first step, the smartphone and the IoT device need to authenticate the identity of each other. In this phase an IoT cloud (maintained by the IoT device vendor) often acts as a reliable authority. With the help of the IoT cloud, the smartphone and the IoT device are bound to each other and both obtain SCs, to be used in the subsequent data transmissions. Note that the SCs are either generated by the IoT cloud and separately sent to both the IoT device and the smartphone, or directly negotiated between the IoT device and the smartphone. Next, each time data is transferred, the two communicating parties utilize the ASC to prove their identities and the CSC to protect the data transmission. Note

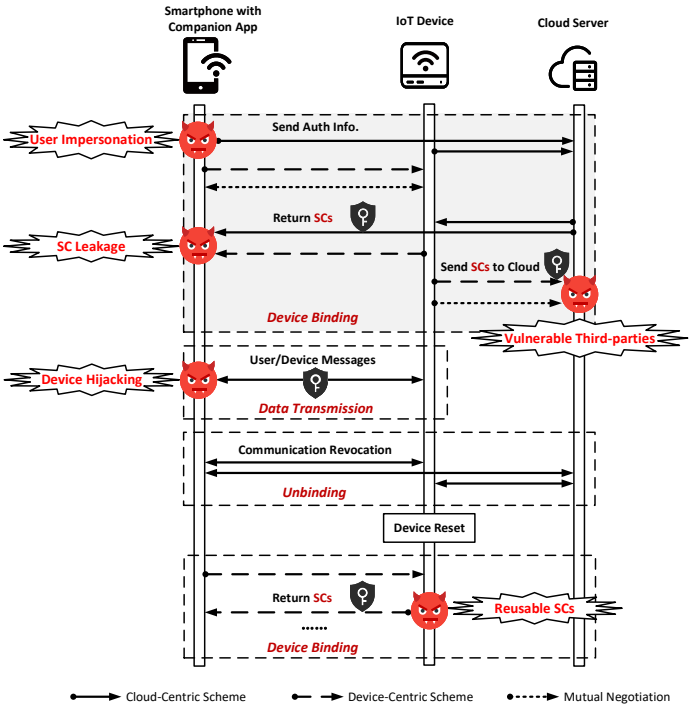

Fig. 1. Typical IoT-to-Mobile Communication Process and the Threat Model

that actually the ASC could be used to derive a session token used for authentication and the CSC used to derive session keys to encrypt and sign the data. Finally, if the smartphone and the IoT device need to revoke the binding, an unbinding request, usually initiated by the smartphone, is sent to the IoT device and the IoT cloud, so that they delete all SCs (and thus invalidate the binding).

SCs play a major role in securing IoT-to-Mobile communication. The ASC proves that only the bound IoT device and smartphone are authorized to communicate; the CSC helps implement cryptographic protection for transmitted data. However, the use of SCs in IoT-to-Mobile communication is error-prone. Unlike other types of credentials (e.g., web tokens) that are regulated by security standards [14]–[18], the implementation of SCs in IoT-to-Mobile communication lacks guidelines. Also existing security standards for credentials of the client-server model are not directly applicable to implement SCs. Accordingly, in this paper we adopt the threat model shown in Figure 1. We assume that the IoT device and the smartphone OS are both benign. However, attackers may install malicious apps on the smartphone, and these apps can access the data storage containing SCs [19]–[21]. We focus the threat model on the IoT-to-Mobile communication under Wi-Fi networks only. There are more powerful adversaries that can attack other local networks, such as Bluetooth [22], [23] and Zigbee [24], but they require specialized resources not commonly available. Also, we assume that the Wi-Fi network is an untrusted network, that is, the attacker can connect to the Wi-Fi network and perform passive eavesdropping or active message forgery attacks.

Under the assumed threat model, the attacker can conduct several attacks. He can forge user information as a legitimate

smartphone to cheat the IoT device (*User Impersonation*), or attack the (insecure) SC distribution (e.g., by utilizing the hard-coded key embedded in the companion app and device firmware) and obtain the SCs (*SC Leakage*). Moreover, the attacker can impersonate the cloud server (*Vulnerable Third-parties*) to distribute fake SCs or obtain the reported SCs. Once an attacker obtains or controls the SC, he can directly hijack the network communication of the smartphone once he is able to decrypt the traffic, and forge user commands to control the IoT device (*Device Hijacking*). Last, the attacker can replay the SCs not revoked after the unbinding operations to compromise the IoT-to-Mobile communication (*Reusable SCs*).

III. Security Properties of Shared Credentials

We found NO existing guidelines for regulating SC implementations. In response, we propose a number of critical security properties that a SC **MUST**[1] or **SHOULD**[2] comply with. To make sure that we identify a set of comprehensive security properties, we have adopted a two-fold process:

1) We divide the life cycle of SCs into five stages: **generation**, **distribution**, **validation**, **protection**, and **revocation**. Then we identify the relevant security properties for each stage.
2) We refer to existing credential design principles developed for Client-Server model and OAuth model, as well as key management standards, according to the guidelines given by related official documents [14]–[18], [25], [26].

Based on our design process, we have identified nine security properties that we believe are sufficient for a secure implementation and use of SCs.

Property 1 – Randomness. A SC MUST prevent brute force and guessing attacks. To achieve this property, a best practice is to generate SCs with a part constructed from a strong cryptographic pseudo-random number generator (PRNG). According to security considerations for OAuth 2.0 in RFC6749 [15] and encryption key in IPsec [26], the probability of an attacker guessing the generated SCs must be less than or equal to 2^{-128} and should be less than or equal to 2^{-160}. In view of both usability and security considerations, we conclude that a SC should contain at least a 128-bit random number.

Property 2 – Secure Distribution Channel. The distribution of SCs MUST rely on a secure channel, which fulfils strong authentication, and guarantees confidentiality and integrity. Authentication requires mutual identity verification between the SC distributor and the SC receiver to prevent man-in-the-middle (MITM) attacks or potentially malicious clients. Confidentiality means that the SCs must not be distributed in the clear, and integrity ensures that SCs cannot be modified during transmission. To satisfy those three security requirements, the RFC6749 [15], RFC6750 [27] and RFC7519 [16] as well as NIST key management [25] documents, which regulate OAuth 2.0 authorization framework, JSON web token and encryption key implementations, all recommend to deploy the Transport Layer Security protocol [28] (TLSv1.2) with multi-authentication to protect SC transmission.

Property 3 – End-to-end SC Sharing. A SC MUST be shared only among the authorized communicating parties. Referring to the security considerations of OAuth 2.0 in RFC6749 and HTTP State Management Mechanism in RFC6265 [17], this property requires that a SC should be only shared between the IoT device, its companion app and the trusted cloud server.

Property 4 – Different ASC and CSC. An IoT-to-Mobile communication MUST implement both ASC for authentication and CSC for communication protection and a given SC SHOULD be used only for identity authentication between the communicating parties or only for data encryption. Specifically, both ASC and CSC are necessary and the ASC and CSC used in an IoT-to-Mobile communication should be different, according to RFC6749 and POLP (Principle of Least Privilege) [29]. For example, if the ASC and CSC are same, an attacker would be able to not only decrypt the encrypted traffic, but also impersonate the smartphone (installing a companion app with login user accounts) to send fake messages once the attacker obtains one of the SCs (i.e., ASC or CSC).

Property 5 – Oblivious Validation. Feedback from the SC validation SHOULD not leak any information about the SC correctness. This property means that the responses should not contain any information that can reflect the correctness of the SC (see the OWASP cheat sheet [18]). Like to the padding oracle attacks [30], incorrectly implemented response messages can reveal meaningful information about the target and can be used for SC enumeration or guessing. Thus, when an invalid SC is detected, a generic response, rather than a message that contains error details, should be returned.

Property 6 – Brute Force Attack Resistance. The validation SHOULD only allow limited attempts with incorrect SCs. This property requires that the validation step maintains a counter for invalid SCs, which limits the attempts to a reasonable range to prevent the brute-force attack. The OWASP cheatsheet recommends that the counter be associated with the SC itself, rather than the source IP address. A best practice is to use a threshold of no more than 20 SC attempts from one source.

Property 7 – Encrypted-then-stored SC. A SC MUST be first encrypted, then stored in non-volatile storage medium such as flash memory (especially on smartphones), to prevent an attacker from obtaining the SC even when the devices are compromised. Past research [31] has shown that once the data is written to mediums such as solid state drives, it is not easy to erase it securely. Therefore if SCs are to be stored, they should be stored in ciphertext form. OWASP Cryptographic Storage cheatsheet suggests to protect SCs using either AES with at least 128 bits key and a secure mode, or ECC with Curve25519 or RSA with at least 2048 bits key.

Property 8 – Short-term SC. A SC SHOULD not be used for a long time. This property requires that the communication sets a SC expiration timeout to reduce the attack window. Referring

[1]**MUST** means that the property is an absolute requirement to implement a secure SC.

[2]**SHOULD** means that some properties may be ignored in specific circumstances. But it is still necessary to understand the full implications and carefully weight them.

to the RFC5280 [32], the X.509 PKI Certificate standard document and key management guide [33], used SCs should be immediately discarded after a session is terminated; OWASP cheatsheet [18] even suggests that for long time sessions, the SCs should be set to expire and renewed in eight hours to balance usability and security.

Property 9 – Revocable SC. A SC SHOULD be revoked actively when it is leaked or expired. This property requires that a secure SC revocation mechanism should be provided. According to OWASP JSON Web Token cheatsheet [18] and NIST key management [25], before using or validating a SC, the communication parties should check whether the SC is revoked. If a SC is revoked, it should fail the verification and not be used for later communications.

IV. Detecting Insecurely Used Shared Credentials

KingFisher is based on two analyses: i) **an analysis of both app code and network traffic to collect functions and packets that are related to SCs**; ii) **a value-based analysis to detect the SCs used for IoT-to-Mobile communication and label the corresponding functions containing these SCs**.

Concerning the first analysis, KingFisher explores all the potential information related to SCs. We observed that IoT vendors commonly customize their proprietary protocols to construct IoT-to-Mobile communication without disclosing specifications and protocol formats. Therefore identifying functions and network traffic related to SCs is challenging. Existing approaches for the analysis of protocol formats and types [34]–[36] are unable to analyze IoT-to-Mobile communications. Some of them can only handle protocols with plain-text messages, while others cannot pinpoint the SC-related fields since they only rely on network traffic analysis. In addition, a simple static code analysis cannot identify the dynamically generated SCs. Hence, KingFisher conducts a hybrid analysis of both functions and packets so to gather comprehensive information for the subsequent analysis.

The main challenge for the second analysis, aiming to detect the SCs used for IoT-to-Mobile communication, is that since the SCs are generated by the IoT device or the IoT cloud, it is difficult to track the data flow of each SC. Unlike credentials generated by the apps, the SCs are typically processed through multiprocessing (e.g., Binder IPC mechanism [37]) and multithreading, which involve both Java code and native code. Therefore, the existing analysis techniques [38], [39] cannot track the data flow of the SC precisely because they cannot analyze code with portions written in multiple languages simultaneously. KingFisher thus uses a value-based comparison, which is a code-independent method at the data flow level, to detect the SCs and the functions that process the SCs, without requiring information about the standard/format followed for encoding the SCs.

KingFisher executes four steps to assess the security of SCs (see the workflow in Figure 2). It first analyzes the app code to label functions correlated to SCs (*Function Interface Identification*). It then combines app dynamic instrumentation with network traffic analysis to collect function runtime values and network packets that are potentially correlated to SCs (*Message Collection*). Given the collected functions and network packets, KingFisher tracks the data flow of each SC in multiple modules/apps (*Value-based Analysis*) and pinpoints the flawed implementations (*Security Violation Detection*).

A. Function Interface Identification

To obtain the SC from a companion app, KingFisher identifies *SC-related candidates* that are potentially directly/indirectly data dependent on the SC. As the SC can be created by the cloud or negotiated locally between the IoT device and the smartphone, KingFisher explores the dynamically loaded functions and further identifies the candidates based on the usage of SCs.

In particular, KingFisher first extracts all functions that are loaded during the app execution through `ClassLoaders` [40]. Then it conducts a keyword-based search to retrieve SC-related candidates. For keyword matching, we manually built a *reference set* containing a list of function names that are commonly used to name SC-related function. In regard to the usage of SCs, they are generally utilized in user authentication and authorization, cryptographic algorithms, and data protection. Thus, we manually explored the SC-related functions from the top 100 IoT app projects and sample codes on Github and StackOverflow to extract the relevant keywords (e.g., "encrypt", "build", "token"). Given a reference set, KingFisher compares the dynamically loaded function with all the keywords in the set. A function is labeled as a SC-related candidate if any keyword is included as a subword of the function name. KingFisher includes the function prototypes of the SC-related candidates (i.e., function name, parameter types, return type) in a *SC Function Candidate List*.

B. Message Collection

KingFisher further dynamically collects the values passing through the corresponding SC-related candidate functions and conducts network traffic analysis.

Function Value Collection. To distinguish SCs, we construct an instrumentation component based on Frida [41] to track the function parameter values and return values for each candidate in the SC function candidate list.

Unlike common functions that are usually written in high-level Java code, SC transmission through network communication involves functions in both Java code and native code, that is, the SC-related candidates might exist in either Java code or native code. KingFisher processes the function candidates in different code levels separately because the programming logic and instrumentation interfaces of Java code and native code are inconsistent. To be more specific, KingFisher directly hooks each SC-related candidate in Java code to obtain all its parameter values and return values. For native code, KingFisher classifies the parameters and return variables of each SC-related function into "pointer" and "non-pointer" variables. For each pointer variable, KingFisher obtains the variable value by extracting the pointed address and further visits the

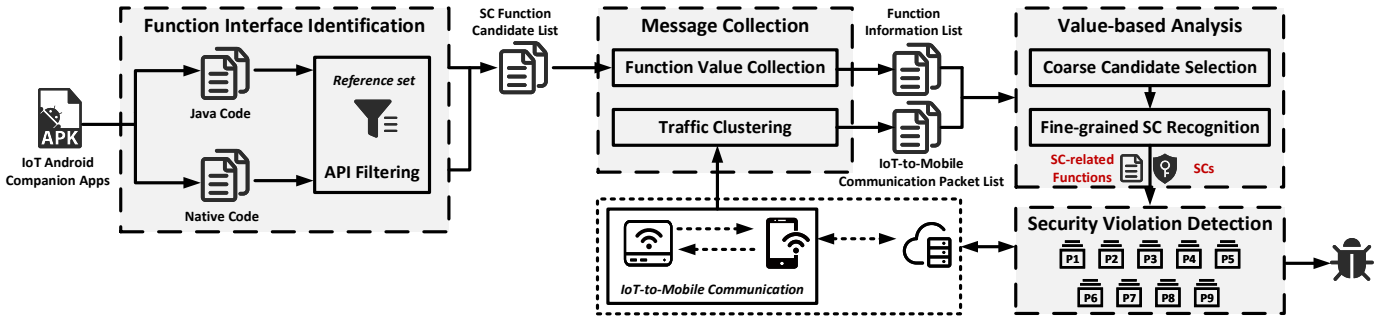

Fig. 2. The workflow of KINGFISHER analysis framework

corresponding memory block of the address, which ends with a sequence of '00' to collect the value stored in the memory block[3]. As the parameter variables might be handled within the function, their values might be modified. KINGFISHER thus records the initial and final values of each variable. Alternatively, KINGFISHER directly records the runtime values of non-pointer variables. All the information about the variable values is stored in a function information list, aligned with the SC-related candidates.

Traffic Clustering. KINGFISHER analyzes network traffic to identify IoT-to-Mobile communication packets. By executing tcpdump [42], KINGFISHER captures all network packets transmitted by the smartphone. It then relies on the IP addresses of both the IoT device and the smartphone to distinguish whether a packet is transmitted for IoT-to-Mobile communication, that is, a packet is considered as an IoT-to-Mobile communication packet if it contains the IP addresses of the IoT device and the smartphone. In addition to IoT-to-Mobile communication packets involving SCs, there are IoT-to-Mobile communication packets used for other purposes, such as heartbeat packets.

Accordingly, KINGFISHER clusters the similar packets into the same group. Given the IoT-to-Mobile communication packets, KINGFISHER utilizes a traffic clustering-based Sequence Alignment [43] to cluster the similar network packets into a group. Specifically, KINGFISHER pairwise compares all the IoT-to-Mobile communication packets and computes a similarity score of each packet pair by using a message similarity computation algorithm — Needleman-Wunsch algorithms [44]. According to the similarity score, it then merges the most similar packets by recursively selecting the pair with the highest similarity score and executes UPGMA clustering algorithm [45] to cluster the similar pairs into the same group[4]. Since the packets in the same group are similar, all packets in the same group will be regarded as containing SCs if any packet is identified as transmitting a SC. Thus, KINGFISHER randomly selects one packet from each group to construct an IoT-to-Mobile communication packet list for the subsequent value analysis.

[3]It is important to note that if there is another memory address stored in the memory block, KINGFISHER continues to visit the memory block of the new memory address. Such an operation is executed iteratively until a valid value is read from the memory address.

[4]We set the dissimilarity index to 0.54 to balance clustering accuracy and efficiency.

C. Value-based Analysis.

Taking as input the function information list and the IoT-to-Mobile communication packet list, KINGFISHER identifies the SCs through value comparison.

Coarse Candidate Selection. As a large amount of candidates and network packets are collected, KINGFISHER first filters out the irrelevant candidates. It compares the values stored in the function information list (i.e., parameter values and the return value) with the values of packets in the IoT-to-Mobile communication packet list. KINGFISHER considers a candidate as irrelevant if the function values do not include any of the packet values. The rest of the candidates are labeled as *initial functions* that are directly data dependent on the involved SC.

In addition, as the return value of each initial function might also be manipulated by the other functions (i.e., indirectly data dependent on the SC), KINGFISHER tracks all these functions to explore the complete SC data flow. Specifically, it compares each return value with the parameter values of the other candidates and labels each as a *related function* if any of its parameter values matches with the return value. KINGFISHER identifies the related functions iteratively until no related function is found.

Fine-grained SC Recognition. Based on the related functions and their values, KINGFISHER next recognizes the used SCs, i.e., ASC and CSC. Through our manual observation, we found that most ASCs are encoded in the format of JSON or based on the format of JSON Web Token, and CSCs are commonly taken as parameters of the cryptographic functions. KINGFISHER further examines the SC values with reference to such an observation. If a value contains a sequence of Base64 strings, KINGFISHER regards the value as an ASC. Otherwise, when a value is in JSON format, KINGFISHER parses the JSON string to extract the value from specific fields [46], [47], which is labeled as an ASC. Alternatively, it labels a value as a CSC if used as an encryption key of a cryptographic function.

As some cryptographic functions are customized, it is difficult to locate the parameters of their encryption keys. To address such an issue, we manually abstract the common characteristics of the encryption keys [25], namely: (1) the key length is a multiple of 16; (2) the key length does not exceed 64 bytes[5]. KINGFISHER then analyzes input/return values of

[5]A common CSC used for encryption key is no more than 32 bytes and considering the hex value of the key, we set the maximum length is 64 bytes.

the customized cryptographic functions with respect to such characteristics. If a value satisfies those two characteristics, KINGFISHER regards the value as a CSC. After labeling all the SCs, KINGFISHER further labels the corresponding function candidates as a SC-related function.

D. Security Violation Detection

Having the identified SCs and SC-related functions, KINGFISHER assesses whether the security properties listed in Section III are violated from the perspectives of SC generation, distribution, validation, protection and revocation.

Detecting Insecurely Generated SCs. To check whether a SC is securely generated, KINGFISHER examines the SC length and its randomness. In particular, KINGFISHER labels the SCs whose length is less than 16 bytes as vulnerable (i.e., violating **P1**). With respect to randomness, KINGFISHER triggers the SC generation procedure for n times by resetting the device, re-provisioning network, and reconnecting the device to collect a sequence of SCs. We set $n = 10$ in our experiment to make sure that the device itself and the remote server of the manufacturer would not be affected by any harmful impacts, such as request explosion to the server. The SC sequence is evaluated with respect to repetition and consistency [48], where repetition refers to the periodical appearance of a subsequence and consistency refers to the use of constant values. If a subsequence in the SC sequence is generated periodically, KINGFISHER considers such a sequence as violating **P1**. Beside, a sequence using a constant value multiple times (i.e., over 3 times in our experiment) is considered vulnerable, violating **P1**[6].

Detecting Insecurely Distributed SCs. Since the procedure of SC distribution should always be protected, KINGFISHER checks the traffic to detect whether such distribution is protected by TLSv1.2 [28] or multi-factor authentication. If this not the case, KINGFISHER labels the distribution procedure as vulnerable, violating **P2**. KINGFISHER then checks whether the SC is disclosed to any untrusted third parties. By using Burp Suite [49], KINGFISHER parses the communication packet transmitted between the cloud and the smartphone. We consider a SC as secure if it exists in the packet transferred from the cloud to the smartphone, which is a cloud-centric distribution. On the contrary, a SC is vulnerable if a smartphone transmits it to the cloud without having received it in advance, violating **P3**.

Detecting Insecurely Validated SCs. A secure IoT-to-Mobile communication should include a SC for authentication and a SC for cryptography purposes. Thus a communication not including any SC or using the same SC for multiple purposes is insecure, violating **P4**. To detect such a violation, KINGFISHER first checks whether the ASC and CSC are identified from IoT-to-Mobile communication. The lack of either of them is considered insecure. When both ASC and CSC are identified, KINGFISHER compares both value. The SC is insecure if both values are the same.

[6]To avoid false positives, we assume that the periodical subsequence and constant value longer than 4 bytes are the same.

To examine whether validation error messages leak information, KINGFISHER modifies the SC value and other data fields (e.g., device identity information) to verify the error responses. First, KINGFISHER generates a pseudorandom character and appends the character to the end of the SC value to modify its length, or replaces the last character of the SC to modify its value only. Then, it utilizes the instrumenting component to hook and replace the parameter values by the incorrect values to trigger IoT-to-Mobile communication. After receiving the responses from the IoT device, KINGFISHER monitors the device responses. If the responses for both incorrect values are different, the IoT-to-Mobile communication is considered as insecure by violating **P5**.

To further examine the protection scheme against DDoS and brute force attacks and avoid sending too many requests to the cloud server, KINGFISHER generates 20 pseudorandom characters to create 20 incorrect values by appending them at the end of original value. If all the response messages are the same, the SC implementation is considered to violate **P6**.

Detecting Insecurely Protected SCs. KINGFISHER performs string match by comparing the SC with files stored at the app local internal storage (i.e., '/data/data/xxx') and the external storage (i.e., '/sdcard/xxx'). If there is any match, a vulnerability is identified (i.e., a violation of **P7**).

Detecting Insecurely Revoked SCs. We assume that a secure SC can only remain constant up to eight hours. Thus, KINGFISHER reuses the SC after eight hours. If it can set up IoT-to-Mobile communication successfully, then the SC implementation is considered as violating **P8**.

After SC revocation, the previous SC needs to be disabled. When the SC is revoked or a new SC is distributed, KINGFISHER sends a message by using the previous SC. The SC revocation security property (**P9**) is violated, if the device correctly responds to the message.

E. Running Example

We use Tuya [50] as an example to demonstrate the workflow of KINGFISHER. Tuya is a global vendor that provides various connectivity solutions for IoT devices in different scenarios. Figure 3 shows the main code implementing IoT-to-Mobile communication in Tuya app and how KINGFISHER works on this code. The Tuya app first utilizes the CSC `localtoken` to encrypt the plain data through function `encryptRequestWithLocalKey`, and then prepares message `data` by combining the encrypted data and other information based on its customized protocol format via function `buildRequest`. Afterwards, a cross-process communication flow across functions `transact` and `onTransact` is implemented to transfer the encrypted data from the client-proxy process to the service-stub process. In the service-stub process, the received data is handled by both Java code and native code, and is then sent by function `buffevent_write` in another thread to the bound IoT device.

In order to detect security violations related to SCs, KINGFISHER first identifies the functions potentially related to SCs (e.g., `encryptRequestWithLocalKey`,

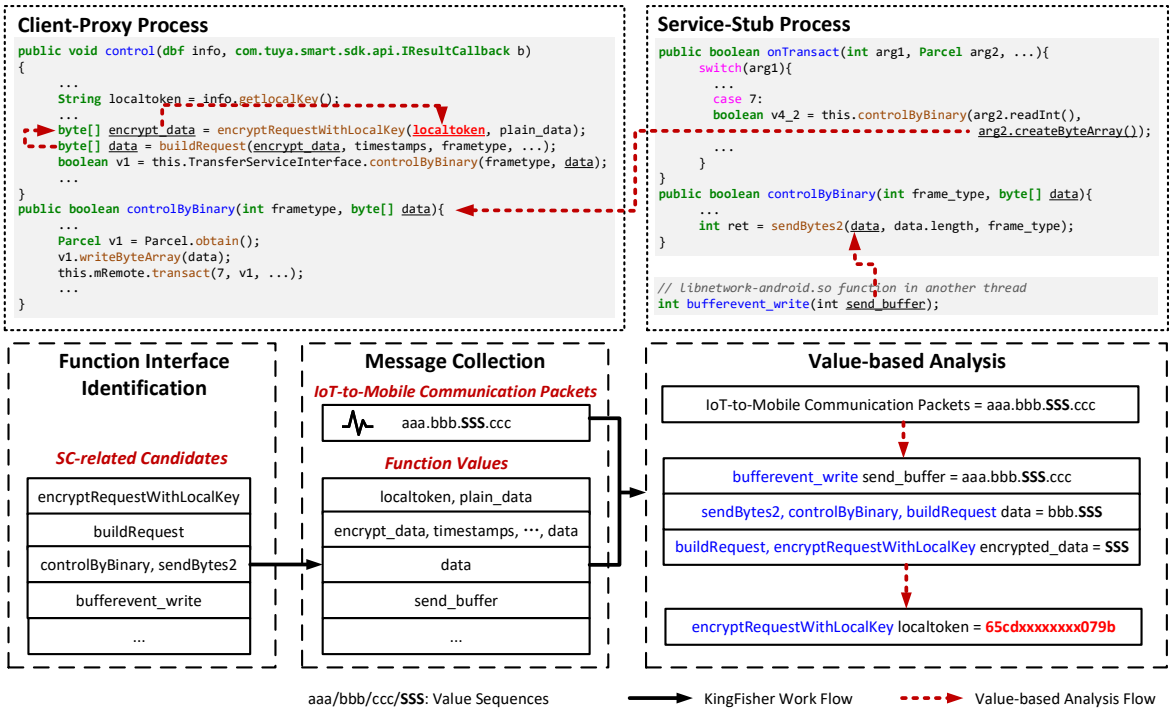

Fig. 3. A Running Example of Tuya IoT-to-Mobile communication

`buildRequest`, `controlByBinary`, `sendBytes2` and `bufferevent_write`). It then records the runtime values including function parameters (e.g., `localtoken`, `data`, `send_buffer`) and return values (e.g., `encrypted_data`) and meanwhile, collects the SC-related packets from network traffics. Having these messages, KingFisher finally executes the value-based comparison to track the SC data flow and check if there are property violations.

V. Experimental Results

In this section, we report our experiments on eight popular IoT solutions that adopt IoT-to-Mobile communication and use SCs.

A. Experiment Setup

Tested Devices. To assess SC security and thoroughly cover different IoT-to-Mobile communication solutions, we carefully selected the solutions to be analyzed. Specifically, we first identified a set of mainstream IoT vendors and then selected the vendors offering solutions based on device companion apps. We then referred to device product descriptions, inquiries with these vendors, and actual testings to check whether devices could communicate locally with their companion apps. Finally, we selected the products most widely used.

Totally, we assessed eight popular IoT-to-Mobile communication solutions, BroadLink [51], Haier [52], Horn [53], Qihoo [54], Tuya [50], Xiaomi [55], Xiaoyi [56], and ZTE [57]. According to the sales data of e-commercial platforms, such as Alibaba, Amazon, and JD.com, as of September 2021, the total shipment number of these devices exceeded 35 million. Furthermore, downloads of each companion app ranged from 2,461,900 (BroadLink) to 6,023,150,000 (Xiaomi).

Testing Environment. We first purchased eight IoT devices that support our selected solutions and registered the user accounts for them. These devices cover smart plugs, smart gateways, and smart cameras, which are commonly used in daily life. Next, we connected the IoT devices and an Android smartphone (ONEPLUS A5000) within the same Wi-Fi, and installed each companion app on the phone. We also deployed the analysis engine of KingFisher on a laptop with Intel Core i7 1.80 GHz and 16G RAM, and the instrumentation engine of KingFisher on the rooted ONEPLUS phone. Then we tested each communication solution.

Ethical Consideration. We registered experimental accounts for all of our evaluations. In our experiments, we simulated the attacks against our own IoT devices and smartphones in order not to cause any usability impact on IoT cloud servers or other IoT devices. Furthermore, we contacted the IoT vendors of the solutions we analyzed and reported them the identified vulnerabilities. We have received eight CNVD IDs (China National Vulnerability Database IDs).

B. SC Extraction Results

The SC extraction results are shown in Table I. Generally, KingFisher successfully analyzed all the eight companion apps and labeled messages containing the used SCs. It accurately extracted SCs for five solutions except BroadLink, Qihoo and Xiaomi. In the following, we discuss the analysis results of KingFisher in detail.

1) Function Interface Identification: The function interface identification results are shown in the `Function Interface Identification` column of Table I. Each app contained at least 100,000 Java functions and 100,000 native functions, shown in the `#All Functions` column.

TABLE I
SC EXTRACTION RESULTS

Vendor	Function Interface Identification				Message Collection					SCs			
	All Functions		SC-related		Function Value Collection		Traffic Clustering			ASC		CSC	
	Java	Native	Java	Native	Functions	Values	All	Local	Clustered	E	V	E	V
BroadLink	101,720	105,952	458	77	4	860	130	47	11	1	0	0	1
Haier	453,721	137,357	1,844	38	17	5,205	542	312	19	1	1	0	0
Horn	208,710	104,437	667	0	65	11,840	133	44	7	0	0	2	2
Qihoo	225,999	121,994	226	373	5	1,615	892	729	12	0	1	1	1
Tuya	256,310	154,178	1,047	57	6	65	255	25	10	0	0	1	1
Xiaomi	538,277	125,808	1,570	11	6	665	141	14	6	0	0	3	1
Xiaoyi	111,126	141,960	157	121	32	8,415	2,391	1,757	60	2	2	1	1
ZTE	148,077	126,650	245	627	24	6,515	1,669	517	25	1	1	0	0

E refers to SC extraction results of KINGFISHER; **V** refers to the SC benchmark built by experienced experts.

KINGFISHER filtered and identified no more than 2,000 SC-related functions in Java code and native code for each companion app (see `#SC-related` column). Through our manual inspections, we observed that SC-related functions are commonly implemented in specific components, such as the third-party Android SDK or libraries. To invoke the libraries correctly, these components are seldom obfuscated or stripped. Therefore, KINGFISHER can easily identify the SC-related functions without being affected by app obfuscation/stripping.

Notice that KINGFISHER did not locate any native functions in the Horn companion app. We manually examined the app and found that Horn app implements the whole IoT-to-Mobile communication in Java code. In general, the filtered SC-related functions only accounted for a small proportion (around 0.3%) of all app functions. The results indicated that our interface locating method is effective in improving analysis efficiency, as it avoids analyzing functions not related to SCs.

False Positive (FP) and False Negative (FN). Due to the large number of all functions and identified SC-related functions in a companion app and the lack of benchmark, it is costly to manually inspect the FP and FN of KINGFISHER *Function Interface Identification* results. But FP and FN do not affect the subsequent SC extraction to a large extent because the value-based data flow analysis double-checks the identified SCs.

2) Message Collection: After locating the SC-related functions, we executed each companion app to start the IoT-to-Mobile communication and meanwhile KINGFISHER collected the runtime SC-related messages (shown in the `Message Collection` column of Table I).

Function Value Collection. KINGFISHER collected information about SC-related functions; the results are shown in the `#Function Value Collection` column. Generally, the number of invoked SC-related functions (`#Functions` column) ranges from 4 (BroadLink) to 65 (Horn). Also on the average KINGFISHER recorded 3,909 pieces of function values (`#Values` column) for each companion app. Among them, from Horn we collected more then 10,000 pieces of function values. The reason is that Horn launches the IoT-to-Mobile communication during the device binding procedures, which involve a higher number of operations, such as device discovery broadcast and SC negotiation.

Traffic Clustering. The `#Traffic Clustering` column shows the results of traffic clustering. To ensure that enough transmitted packets were collected, we executed IoT-to-Mobile communication for each companion app many times until more than 100 packets were captured. As a result, the total number of transmitted packets KINGFISHER collected for each app (Column `#All`) ranges from 130 (BroadLink) to 2,391 (Xiaoyi) and the number of IoT-to-Mobile communication packets (`#Local` column), filtered by IP addresses, ranges from 14 (Xiaomi) to 1,757 (Xiaoyi), accounting for about 60%. After that, KINGFISHER clustered these packets to obtain the IoT-to-Mobile communication packets that contain SC-related packets; the results are listed in the `#Clustered` column. On the average, 19 packets were filtered for each companion app, accounting for about 2.4% of the total number of packets. This proportion also indicates that our traffic cluster methods is able largely reduce redundancy and improve efficiency. After clustering, KINGFISHER randomly chose one packet from each cluster for further SC extraction.

False Positive (FP) and False Negative (FN). For the function value collection, we should only guarantee there is no FN in our results. Because we adopt the value-based analysis to recognize the SCs, its accuracy depends on the similarity between values. Since the IoT-to-Mobile communication related values are usually present only in specific protocol patterns, it is unlikely that other irrelevant parameters or return values would have high similarity with them. Hence, we just need to collect enough SC-related values to track the SC data flow. So we manually checked the results of the *Function Value Collection* and found that all the SC values were contained in our results.

To evaluate the accuracy of traffic clustering, we executed a manual validation to check the packets in the IoT-to-Mobile communication packet list. The result of the validation showed that at least one packet fully met our requirements (that is, to be related to SCs) for each app, which confirms that our cluster method is effective in obtaining at least one packet payload to be used for further analysis to extract the SC.

3) Extracted SCs: After collecting SC-related messages, KINGFISHER conducted the value-based analysis to extract the SCs. To evaluate the accuracy of SC extraction results, we first built the SC benchmark for the eight products. Specifically, we invited some security experts, who are experienced in

Android app analysis and familiar with IoT systems and their communication, to help us build a reference dataset for the used SCs in the eight products. By reverse engineering and dynamically debugging the apps, constructing control flow from network interfaces and tracking the SC data flow, we successfully extracted the used SCs of all the eight products and their related functions. This procedure took about seven days and each result was agreed on by at least two experts to avoid human bias/errors. We took the reference dataset as the benchmark to evaluate the SC extraction results of KINGFISHER. Note that there were some SCs changed in every session, so we also compared the SC-related functions to double confirm the results.

As shown in the V columns in Table I, KINGFISHER accurately identified the used SCs in five products, i.e., Haier, Horn, Tuya, Xiaoyi and ZTE. The accuracy of KINGFISHER SC extraction is around 69% with 9 correct SCs in 13 identified SCs. Interestingly, we found that for Horn and Xiaoyi, KINGFISHER identified more than one SC; these results were manually confirmed. Both solutions assemble two values to construct the SC, and at the code level, it is correct for KINGFISHER to label each of them as a separate SC. The labeled SCs and SC-related functions were further used to detect security violations. For the three products (i.e., BroadLink, Qihoo and Xiaomi), in which KINGFISHER did not identify their used SCs successfully, we manually labeled their SC values and the SC-related functions for further security violation detection.

False Positive (FP) and False Negative (FN). The extraction results by KINGFISHER have two false positives; one is related to the CSC used in BroadLink and the other in the ASC used in Qihoo. BroadLink implements its IoT-to-Mobile communication encryption procedure, which manages the CSC, in native code, so KINGFISHER could not split the CSC from the string memory buffer since the code contiguously stores other data directly following the CSC, which does not end with a sequence of '00'. As a result, KINGFISHER recognized the value as a longer string and excluded it. For Qihoo, KINGFISHER successfully identified its CSC but could not find the ASC. This is because Qihoo uses same values and CSC format as described in Section IV-C to implement both ASCs and CSCs. Hence, KINGFISHER identified this value as a CSC rather than an ASC.

There are two false negatives in the KINGFISHER SC extraction results, because KINGFISHER mistakenly identified the ASC in BroadLink and the CSC in Xiaomi. For BroadLink, KINGFISHER labeled the CSC value as ASC because one data construction function (`dnaControl`) takes a JSON string including the CSC as a parameter, conforming the ASC format as described in Section IV-C. The CSC of Xiaomi is similar to CSC of BroadLink, whose CSC is used as a part of a string parameter of `AES_cbc_encrypt` function in `libmiio.so`, and KINGFISHER mistakenly identified three SC candidates but excluded the correct one.

Generally, KINGFISHER did not recognize the SCs used in those three products successfully because KINGFISHER did not identify some SC-related candidates, in which the SC are used as individual values and can be parsed correctly, due to the obfuscation technologies. But as mentioned before, such failures can be avoided by manually labeling the SC-related candidates as well as their semantics information in *Function Interface Identification* without a significant time overhead.

TABLE II
SECURITY VIOLATION RESULTS

Vendors	P1	P2	P3	P4	P5	P6	P7	P8	P9
BroadLink	✗	✗	✗	✗	✗	✗	✓	✗	✓
Haier	✓	✗	✓	✗	✗	✓	✓	✓	✗
Horn	✓	✗	✓	✗	✗	✗	✓	✓	✓
Qihoo	✓	✗	✓	✗	✗	✓	✓	✗	✓
Tuya	✓	✗	✓	✗	✗	✗	✓	✗	✓
Xiaomi	✓	✗	✗	✗	✗	✗	✓	✗	✓
Xiaoyi	✓	✗	✓	✗	✓	✗	✗	✓	✓
ZTE	✓	✗	✓	✗	✗	✗	✓	✓	✗

✗ insecure implementations that violate this property.
✓ no violation found.

C. Security Violations

After obtaining the SCs, we analyzed the whole SC life cycle and checked whether it satisfies the security properties described in Section III. Table II shows the results of KINGFISHER analysis for the SC security. As we can see, all the eight products violate several security properties. In general, most IoT vendors adopt secure solutions for SC generation and storage. The only exceptions are BroadLink that implements an insecure SC generation, and Xiaoyi that violates the security property of SC storage. On the other hand, all solutions violate the SC distribution and usage security best practices. As for SC update, all vendors, except Horn and Xiaoyi, have a vulnerable implementation. Among the eight IoT vendors, five vendors, i.e., Haier, Qihoo, Tuya, Xiaoyi and ZTE, adopt Cloud-Centric schemes for SC generation and distribution, whereas two vendors, i.e., BroadLink, Xiaomi, adopt Device-Centric schemes, in which the SC is generated and distributed by the IoT device. Only Horn implements a local Mutual Negotiation between the device and companion app for the SC. In the following, we describe in details the security violation detection results obtained by KINGFISHER.

1) Insecurely Generated SCs: Among the analyzed solutions, only BroadLink violated **P1** as its SC was predictable in that it had repeated subsequences. We further checked and found that the SC of BroadLink was composed by using four subsequences from a fixed set of 15 subsequences. That means that the device can only generate a limited number of different SCs. We infer that this may be caused by constrained device resources (i.e., memory and processor).

2) Insecurely distributed SCs: All products did not provide enough protections for the distribution of SCs. For five devices (i.e., Haier, Qihoo, Tuya, Xiaoyi and ZTE) adopting Cloud-Centric schemes, though they utilized TLS to protect the SC distribution, none of their SC distributions were protected with secure TLS protocol or multi-authentication, thus violating **P2**. The two products using device-centric schemes and Horn with the mutual negotiation scheme ((i.e., BroadLink and Xiaomi)

implemented their SC distribution without TLS protection, thus also violating **P2**. Moreover, two products (i.e., BroadLink and Xiaomi) generated their SCs locally; however, the SCs were still reported to the cloud after generation. Thus, we also consider them as violating **P3**.

To validate those findings, we manually checked their distribution mechanisms and the results confirmed the findings of KINGFISHER. Specifically, Haier, Qihoo, Tuya, Xiaoyi and ZTE implemented cloud-centric schemes, by which the SCs were distributed by the cloud and then transmitted to both IoT devices and companion apps over an HTTPS connection. The other three vendors implemented their customized SC distribution in the Wi-Fi network. BroadLink utilized the `AES-CBC` encryption algorithm to protect the SC distribution. However, it used the default distribution key and initial vector, which were embedded in the companion app, so that an attacker could easily extract them by reverse engineering the app, and then decrypt the communication to obtain the transmitted SC. The Xiaomi companion app first connected to the subnet of the device during the binding phase. The device then distributed the SC in plaintext to the app under the subnet. This implementation is vulnerable since the attacker would be able to obtain the SC by connecting to the subnet and sending a fixed request message; then the device would transmit the SC without authenticating the request message. As for Horn, the SC negotiation process is insecure because of the lack of encryption protection, so that the attacker can easily obtain the exchanged nonces by just passive eavesdropping and then calculate the SC.

3) Insecurely Validated SCs: All the eight devices implement vulnerable SC validation. Generally, all the eight devices violated **P4**. Except Xiaoyi, seven devices violated **P5**. Moreover, six devices (i.e., BroadLink, Horn, Tuya, Xiaomi, Xiaoyi and ZTE) violated **P6**.

Six devices implemented one type of SCs, that is, two devices (i.e., Haier and ZTE) only implemented ASCs while four devices (i.e., BroadLink, Horn, Tuya and Xiaomi) only implemented CSCs. Hence, they are considered as violating **P4**. Moreover, although Qihoo and Xiaoyi implemented both ASCs and CSCs, they used the same value to configure ASCs and CSCs. Therefore, KINGFISHER also labeled them as violating **P4**.

For the oblivious validation feedback, we found that the SC responses by all the devices, except for the ones by Xiaoyi, were different depending on whether the SCs or some other data (e.g., device identifier, protocol format) were incorrect. As for Xiaoyi, its responses were always the same independently of whether we modified its SCs or other data fields.

When checking brute attack resistance, we found that six devices, i.e., BroadLink, Horn, Tuya, Xiaomi, Xiaoyi and ZTE, would accept more than 20 erroneous connection attempts without restricting the connection, such as denying requests from one source IP address. For Haier, it replied with the same error messages after the fifth attempt even if the SC was correct. Similarly, Qihoo adopted a strict scheme by which it did not respond after the second attempt. For the above two devices, only when resetting the connection, the devices would return to a normal state, which limits attack efficiency to a certain extent.

In addition to the three security properties, we checked the detailed SC usages to verify the security of their IoT-to-Mobile communication implementations. BroadLink, Horn, Tuya and Xiaomi only implemented the CSCs used to encrypt their IoT-to-Mobile communication. Specifically, BroadLink, Xiaomi and Horn adopted the `AES-CBC` algorithm to encrypt the IoT-to-Mobile communication messages. But there is slight difference. Specifically, BroadLink and Horn used their CSC as the encryption key directly without any further processing, while Xiaomi calculated the `MD5` value of the CSC and used it as encryption key. Tuya adopted the `AES-ECB` encryption algorithm with CSC directly as encryption key to protect the transmitted messages. Since the `AES-ECB` cryptographic algorithm has been proved to be insecure [58], it cannot guarantee the confidentiality of the IoT-to-Mobile communication. Haier and ZTE only implemented the ASCs used for authentication credentials. And none of their IoT-to-Mobile communication was configured with encryption protection. In particular, Haier only encoded the IoT-to-Mobile communication messages, which contained the ASC, and sent them to the devices rather than encrypting them, while ZTE just transmitted the ASCs in plaintext TCP streams. As a result, their ASC transmission is highly insecure. Because once the attacker recovers the corresponding decoding algorithm from the companion app or just eavesdrops the traffics, the attacker can obtain the ASC and then perform an active Man-in-the-Middle attack. Qihoo and Xiaoyi implemented the same ASC and CSC, which are also vulnerable because the attacker can not only decrypt the communication but also impersonate as the victim to pass authentication as long as he obtains any one of the SCs.

4) Insecurely Protected SCs: Only Xiaoyi implemented insecure SC storage. Xiaoyi stored its SCs encrypted, but after a manual analysis, we found that it also stored its encryption key in the app local storage without any protection, so that KINGFISHER also labeled it as violating **P7**.

5) Insecurely Revoked SCs: We found that six devices did not implement a secure SC update and revocation mechanism, i.e., BroadLink, Haier, Qihoo, Tuya, Xiaomi and ZTE. Among them, four devices (i.e., BroadLink, Qihoo, Tuya and Xiaomi) would not update the SC before unbinding or network re-provisioning. Their SCs remained fixed more than eight hours so that we consider them as violating **P8**. Two devices (i.e., Haier and ZTE) implemented an insecure SC revocation as the old SCs could still be valid after a new SC was distributed, violating **P9**.

D. Attacks

1) Device Hijacking: Since the SCs are shared by both the IoT device and the companion app, vulnerable SCs may not only cause device data injection, but also device hijacking. That is, if the attacker obtains the SCs, the attacker would be able to construct device control commands and messages, which involve the SCs (for either authentication or encryption).

Like data injection attacks, if BroadLink SCs are obtained by an attacker, they can be used to construct user messages with the same protocol as device messages. Hence, the attacker can impersonate the legitimate companion app to send fake commands to control the device.

2) Data Injection: When an attacker obtains the ASC used in the IoT-to-Mobile communication, the attacker can forge device status messages. As a result, data from "device" cannot be trusted. Take BroadLink as an example; since the ASC and CSC it uses are equal, the attacker can also obtain the ASC after obtaining the CSC. Moreover, the BroadLink IoT-to-Mobile communication protocol can be recovered by reversing the companion app. Therefore, the attacker is able to construct legal messages with the ASC. Since the IoT-to-Mobile communication security is guaranteed by SCs only, the user cannot distinguish the forged messages from actual device messages if they use same SCs. This is even more dangerous when the device owner configures action-trigger rules, which have cascade effects resulting in the automatic execution of other operations.

3) Privacy Leakage: If a CSC is insecurely used (e.g., the CSC is generated with repeated subsequences like the case of BroadLink), the number of possible CSCs that an attacker has to try decreases and thus the attacker can quickly find a correct CSC. As a result the attacker can decrypt all IoT-to-Mobile communications.

VI. Discussion

Problem Scope. In this paper, we focus on analyzing Wi-Fi based IoT-to-Mobile communication. Although we did not include the other channels (e.g., Bluetooth and Zigbee), KingFisher can be easily extended to analyze the other channels by collecting information about app code and communication traffic.

Manual Efforts. KingFisher is a partial-automated analysis tool, which involves manual operations, such as enabling the procedure of IoT-to-Mobile communication for SC generation, distribution, validation, protection and revocation. These manual operations are inevitable because these operations are heterogeneous in different solutions, in which user participation is necessary. Besides, SCs are required to be confirmed manually if an app code is obfuscated or stripped.

Extended Application Scenarios. KingFisher utilizes companion apps to explore the security of SCs and IoT-to-Mobile communication, so source code is not necessary. Moreover, in this paper, we have used KingFisher to analyze the most popular ones — Android companion apps. But as KingFisher is not tied to a specific framework and is suitable for both Java and C/C++, it can also be used to analyze other apps (e.g., iOS apps) as long as network traffic of IoT-to-Mobile communication can be collected.

VII. Related Works

Prior works on IoT security mainly focus on devices. Firmalice [59] is a binary analysis framework to automatically detect authentication bypass flaws in embedded device firmware. Costin *et al.* [60] performed the first large-scale static analysis of firmware images, while Kim *et al.* [61] proposed an approach for the dynamic analysis of firmware in scale via arbitrated emulation. Some work also analyzed IoT device security via analyses of the companion app [62]. Unlike such previous work, we focus on device authentication security, especially on the security of the SCs used for IoT-to-Mobile communication, instead of code defects.

Other related work focuses on device communication. Chen *et al.* [2] performed a study of the life cycle of remote binding in IoT and demystified various design principles by using a finite state machines model. Sethi *et al.* [63] pointed out that most device pairing protocols are vulnerable to misbinding. They implemented related attacks and showed how the attacks can be found by using formal models of the protocols. Jia *et al.* [9] discussed the security risks in the use of the MQTT protocol and introduced some new general design principles. Such previous work is related to the communications between the cloud platform and the IoT devices. In contrast, our analysis focuses on IoT-to-Mobile communication between the IoT devices and their companion apps.

There is also work focusing on credential security. Ma *et al.* [48] [64] performed an experimental study to analyze the security of the SMS One-Time Password (OTP) authentication protocols. Rahat *et al.* [65] and Wang *et al.* [66] analyzed the vulnerabilities of OAuth Implementations, including the security of OAuth access token. Some well known manufacturers and organizations also proposed their own token or IoT security mechanisms, such as Microsoft Azure [67], Amazon AWS [68], Google Cloud [69], IBM [70], Kaspersky [71], OWASP [18] and IETF RFCs [15], [16]. Unlike such previous works, we focus on security of the SCs used in IoT-to-Mobile communication and propose a comprehensive and complete set of best practices covering the SC entire life cycle. Also our work focuses on tokens in IoT environments, in which specific security vulnerabilities (such as misusing tokens as encryption keys) exist due to the limited resources of many IoT systems, rather than in traditional browser-server or the OAuth model.

VIII. Conclusion

In this paper, we performed a comprehensive analysis of IoT-to-Mobile communication with focus on shared credentials (SCs). We defined nine security proprieties of SC implementations, covering the whole life cycle. Following these proprieties, we proposed an SC-centric analysis framework — KingFisher, to identify SCs by a value-based method and determine whether they are vulnerable by static and dynamic testing. Then we evaluated the IoT-to-Mobile communication solutions of eight popular IoT vendors. Based on the analysis by KingFisher, we found all these products implement an insecure SC life cycle, which may result in sensitive data leakage, persistent denial-of-service, and even device hijacking.

References

[1] D. A. Gratton, *The Handbook of Personal Area Networking Technologies and Protocols*. Cambridge University Press, 2013.

[2] J. Chen, C. Zuo, W. Diao, S. Dong, Q. Zhao, M. Sun, Z. Lin, Y. Zhang, and K. Zhang, "Your iots are (not) mine: On the remote binding between iot devices and users," in *Proceedings of the 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2019.

[3] B. Yuan, Y. Jia, L. Xing, D. Zhao, X. Wang, and Y. Zhang, "Shattered chain of trust: Understanding security risks in cross-cloud iot access delegation," in *Proceedings of the 29th USENIX Security Symposium (Usenix Security)*, 2020.

[4] "EU general data protection regulation (GDPR)," https://gdpr-info.eu/, Accessed 2021.

[5] "101 complaints on eu-us transfers filed," https://noyb.eu/en/101-complaints-eu-us-transfers-filed, Accessed 2021.

[6] "Amazon gdpr violation," https://www.sec.gov/ix?doc=/Archives/edgar/data/1018724/000101872421000020/amzn-20210630.htm, Accessed 2021.

[7] T. Yu, V. Sekar, S. Seshan, Y. Agarwal, and C. Xu, "Handling a trillion (unfixable) flaws on a billion devices: Rethinking network security for the internet-of-things," in *Proceedings of the 14th ACM Workshop on Hot Topics in Networks (HotNets)*, 2015.

[8] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis *et al.*, "Understanding the mirai botnet," in *Proceedings of the 26th USENIX Security Symposium (Usenix Security)*, 2017.

[9] Y. Jia, L. Xing, Y. Mao, D. Zhao, X. Wang, S. Zhao, and Y. Zhang, "Burglars' iot paradise: Understanding and mitigating security risks of general messaging protocols on iot clouds," in *Proceeding of the 41st IEEE Symposium on Security and Privacy (S&P)*, 2020.

[10] Y. Zhou, L. Wu, Z. Wang, and X. Jiang, "Harvesting developer credentials in android apps," in *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks (WiSec)*, 2015.

[11] H. Wen, J. Li, Y. Zhang, and D. Gu, "An empirical study of sdk credential misuse in ios apps," in *Proceedings of the 25th Asia-Pacific Software Engineering Conference (APSEC)*, 2018.

[12] C. Zuo, Q. Zhao, and Z. Lin, "Authscope: Towards automatic discovery of vulnerable authorizations in online services," in *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[13] C. Zuo, Z. Lin, and Y. Zhang, "Why does your data leak? uncovering the data leakage in cloud from mobile apps," in *Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P)*, 2019.

[14] "The oauth 1.0 protocol," https://datatracker.ietf.org/doc/html/rfc5849, Accessed 2021.

[15] "The oauth 2.0 authorization framework," https://datatracker.ietf.org/doc/html/rfc6749, Accessed 2021.

[16] "Json web token (jwt)," https://datatracker.ietf.org/doc/html/rfc7519, Accessed 2021.

[17] "Http state management mechanism," https://datatracker.ietf.org/doc/html/rfc6265, Accessed 2021.

[18] "Owasp cheat sheet series," https://cheatsheetseries.owasp.org/, Accessed 2021.

[19] Y. Acar, M. Backes, S. Bugiel, S. Fahl, P. McDaniel, and M. Smith, "Sok: Lessons learned from android security research for appified software platforms," in *Proceedings of the 37th IEEE Symposium on Security and Privacy (S&P)*, 2016.

[20] Y. Lee, T. Li, N. Zhang, S. Demetriou, M. Zha, X. Wang, K. Chen, X. Zhou, X. Han, and M. Grace, "Ghost installer in the shadow: Security analysis of app installation on android," in *Proceedings of the 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2017.

[21] H. Altuwaijri and S. Ghouzali, "Android data storage security: A review," *Journal of King Saud University-Computer and Information Sciences*, 2020.

[22] D. Antonioli, N. O. Tippenhauer, and K. Rasmussen, "Bias: Bluetooth impersonation attacks," in *Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P)*, 2020.

[23] Y. Zhang, J. Weng, R. Dey, Y. Jin, Z. Lin, and X. Fu, "Breaking secure pairing of bluetooth low energy using downgrade attacks," in *Proceedings of th 29th USENIX Security Symposium (Usenix Security)*, 2020.

[24] E. Ronen, A. Shamir, A.-O. Weingarten, and C. O'Flynn, "Iot goes nuclear: Creating a zigbee chain reaction," in *Proceedings of the 38th IEEE Symposium on Security and Privacy (S&P)*, 2017.

[25] E. Barker, W. Barker, W. Burr, W. Polk, and M. Smid, "Nist special publication 800-57," *NIST Special publication*, 2020.

[26] "The aes-cbc cipher algorithm and its use with ipsec," https://datatracker.ietf.org/doc/html/rfc3602, Accessed 2021.

[27] "The oauth 2.0 authorization framework: Bearer token usage," https://datatracker.ietf.org/doc/html/rfc6750, Accessed 2021.

[28] "The transport layer security (tls) protocol version 1.2," https://www.ietf.org/rfc/rfc5246.html, Accessed 2021.

[29] "Cisa least privilege," https://us-cert.cisa.gov/bsi/articles/knowledge/principles/least-privilege, Accessed 2021.

[30] S. Vaudenay, "Security flaws induced by cbc padding—applications to ssl, ipsec, wtls..." in *Proceedings of the 21st Annual International Conference on the Theory and Applications of Cryptographic Techniques (Eurocrypt)*, 2002.

[31] M. Y. C. Wei, L. M. Grupp, F. E. Spada, and S. Swanson, "Reliably erasing data from flash-based solid state drives," in *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST)*, 2011.

[32] "Internet x.509 public key infrastructure certificate and certificate revocation list (crl) profile," https://datatracker.ietf.org/doc/html/rfc5280, Accessed 2021.

[33] "The definitive guide to encryption key management fundamentals," https://info.townsendsecurity.com/definitive-guide-to-encryption-key-management-fundamentals, Accessed 2021.

[34] Y. Ye, Z. Zhang, F. Wang, X. Zhang, and D. Xu, "Netplier: Probabilistic network protocol reverse engineering from message traces," in *Proceedings of the 28th Annual Network and Distributed System Security Symposium (NDSS)*, 2021.

[35] G. Bossert, F. Guihéry, and G. Hiet, "Towards automated protocol reverse engineering using semantic information," in *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security (AsiaCCS)*, 2014.

[36] W. Cui, J. Kannan, and H. J. Wang, "Discoverer: Automatic protocol reverse engineering from network traces," in *Proceedings of the 16th USENIX Security Symposium (Usenix Security)*, 2007.

[37] "Android binder ipc," https://source.android.com/devices/architecture/hidl/binder-ipc, Accessed 2021.

[38] J. Caballero, H. Yin, Z. Liang, and D. Song, "Polyglot: Automatic extraction of protocol message format using dynamic binary analysis," in *Proceedings of the 14th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2007.

[39] Z. Lin, X. Zhang, and D. Xu, "Automatic reverse engineering of data structures from binary execution," in *Proceedings of the 11th Annual Information Security Symposium (CERIAS)*, 2010.

[40] "Classloader," https://developer.android.com/reference/java/lang/ClassLoader, Accessed 2021.

[41] "Frida," https://frida.re/, Accessed 2021.

[42] "Android tcpdump," https://www.androidtcpdump.com/, Accessed 2021.

[43] "Sequence alignment," https://en.wikipedia.org/wiki/Sequence_alignment, Accessed 2021.

[44] "Needleman-wunsch algorithm," https://en.wikipedia.org/wiki/Needleman%E2%80%93Wunsch_algorithm, Accessed 2021.

[45] R. R. Sokal, "A statistical method for evaluating systematic relationships," *Univ. Kansas, Sci. Bull.*, 1958.

[46] J. Huang, Z. Li, X. Xiao, Z. Wu, K. Lu, X. Zhang, and G. Jiang, "{SUPOR}: Precise and scalable sensitive user input detection for android apps," in *Proceedings of the 24th USENIX Security Symposium (Usenix Security)*, 2015.

[47] S. Lounici, M. Rosa, C. M. Negri, S. Trabelsi, and M. Önen, "Optimizing leak detection in open-source platforms with machine learning techniques," in *Proceedings of the 7th International Conference on Information Systems Security and Privacy (ICISSP)*, 2021.

[48] S. Ma, R. Feng, J. Li, Y. Liu, S. Nepal, E. Bertino, R. H. Deng, Z. Ma, and S. Jha, "An empirical study of sms one-time password authentication in android apps," in *Proceedings of the 35th Annual Computer Security Applications Conference (ACSAC)*, 2019.

[49] "Burp suite," https://portswigger.net/burp, Accessed 2021.

[50] "Tuya," https://www.tuya.com/, Accessed 2021.

[51] "Broadlink," https://www.ibroadlink.com/, Accessed 2021.

[52] "Haier," https://www.haier.com/global/, Accessed 2021.

[53] "Horn," http://www.ihorn-tech.com/, Accessed 2021.

[54] "Qihoo," https://jia.360.cn/, Accessed 2021.

[55] "Xiaomi," https://www.mi.com/global/, Accessed 2021.

[56] "Xiaoyi," https://www.xiaoyi.com/, Accessed 2021.

[57] "Zte," https://www.zte.com.cn/global/, Accessed 2021.

[58] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, "An empirical study of cryptographic misuse in android applications," in *Proceedings of the 20th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2013.

[59] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, "Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware," in *Proceedings of the 22nd Annual Network and Distributed System Security Symposium (NDSS)*, 2015.

[60] A. Costin, J. Zaddach, A. Francillon, and D. Balzarotti, "A large-scale analysis of the security of embedded firmwares," in *Proceedings of the 23rd USENIX Security Symposium (Usenix Security)*, 2014.

[61] M. Kim, D. Kim, E. Kim, S. Kim, Y. Jang, and Y. Kim, "Firmae: Towards large-scale emulation of iot firmware for dynamic analysis," in *Proceedings of the 37th Annual Computer Security Applications Conference (ACSAC)*, 2020.

[62] N. Redini, A. Continella, D. Das, G. D. Pasquale, N. Spahn, A. Machiry, A. Bianchi, C. Kruegel, and G. Vigna, "Diane: Identifying fuzzing triggers in apps to generate under-constrained inputs for iot devices," in *Proceedings of the 42nd IEEE Symposium on Security and Privacy (S&P)*, 2021.

[63] M. Sethi, A. Peltonen, and T. Aura, "Misbinding attacks on secure device pairing and bootstrapping," in *Proceedings of the 14th ACM Symposium on Information, Computer and Communications Security (AsiaCCS)*, 2019.

[64] S. Ma, J. Li, H. Kim, E. Bertino, S. Nepal, D. Ostry, and C. Sun, "Fine with "1234"? an analysis of sms one-time password randomness in android apps," in *Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering (ICSE)*, 2021.

[65] T. Al Rahat, Y. Feng, and Y. Tian, "Oauthlint: An empirical study on oauth bugs in android applications," in *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019.

[66] H. Wang, Y. Zhang, J. Li, H. Liu, W. Yang, B. Li, and D. Gu, "Vulnerability assessment of oauth implementations in android applications," in *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC)*, 2015.

[67] "Microsoft azure iot security deployment," https://docs.microsoft.com/en-us/azure/iot-fundamentals/iot-security-deployment, Accessed 2021.

[68] "Aws iot security," https://docs.aws.amazon.com/iot/latest/developerguide/iot-security.html, Accessed 2021.

[69] "Google cloud device security," https://cloud.google.com/iot/docs/concepts/device-security, Accessed 2021.

[70] "Ibm security token," https://www.ibm.com/docs/en/was/9.0.5?topic=authentication-security-token, Accessed 2021.

[71] "Kaspersky best practices for iot security," https://www.kaspersky.com/resource-center/preemptive-safety/best-practices-for-iot-security, Accessed 2021.