# Automatic Fingerprinting of Vulnerable BLE IoT Devices with Static UUIDs from Mobile Apps

Chaoshun Zuo
zuo.118@osu.edu
The Ohio State University
Columbus, OH

Haohuang Wen
wen.423@osu.edu
The Ohio State University
Columbus, OH

Zhiqiang Lin
zlin@cse.ohio-state.edu
The Ohio State University
Columbus, OH

Yinqian Zhang
yinqian@cse.ohio-state.edu
The Ohio State University
Columbus, OH

## ABSTRACT

Being an easy-to-deploy and cost-effective low power wireless solution, Bluetooth Low Energy (BLE) has been widely used by Internet-of-Things (IoT) devices. In a typical IoT scenario, an IoT device first needs to be connected with its companion mobile app which serves as a gateway for its Internet access. To establish a connection, a device first broadcasts advertisement packets with UUIDs to nearby smartphone apps. Leveraging these UUIDs, a companion app is able to identify the device, pairs and bonds with it, and allows further data communication. However, we show that there is a fundamental flaw in the current design and implementation of the communication protocols between a BLE device and its companion mobile app, which allows an attacker to precisely fingerprint a BLE device with static UUIDs from the apps. Meanwhile, we also discover that many BLE IoT devices adopt "Just Works" pairing, allowing attackers to actively connect with these devices if there is no app-level authentication. Even worse, this vulnerability can also be directly uncovered from mobile apps. Furthermore, we also identify that there is an alarming number of vulnerable app-level authentication apps, which means the devices connected by these apps can be directly controlled by attackers. To raise the public awareness of BLE IoT device fingerprinting and also uncover these vulnerable BLE IoT devices before attackers, we develop an automated mobile app analysis tool BleScope and evaluate it with all of the free BLE IoT apps in Google Play store. Our tool has identified 1,757 vulnerable mobile apps in total. We also performed a field test in a 1.28 square miles region, and identified 5,822 real BLE devices, among them 5,509 (94.6%) are fingerprintable by attackers, and 431 (7.4%) are vulnerable to unauthorized access. We have made responsible disclosures to the corresponding app developers, and also reported the fingerprinting issues to the Bluetooth Special Interest Group.

ACM ISBN 978-1-4503-6747-9/19/11…$15.00
https://doi.org/10.1145/3319535.3354240

## CCS CONCEPTS

• **Security and privacy** → **Embedded systems security**; **Security protocols**; **Mobile and wireless security**; **Software reverse engineering**; **Privacy protections**; *Access control*; *Mobile platform security*.

## KEYWORDS

Bluetooth low energy, device fingerprinting, mobile app analysis, IoT security

**ACM Reference Format:**
Chaoshun Zuo, Haohuang Wen, Zhiqiang Lin, and Yinqian Zhang. 2019. Automatic Fingerprinting of Vulnerable BLE IoT Devices with Static UUIDs from Mobile Apps. In *2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19), November 11–15, 2019, London, United Kingdom.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3319535.3354240

## 1 INTRODUCTION

Over the past few years, we have witnessed a huge increase in the number of the Internet-of-Things (IoT) devices (e.g., sensors, and actuators) running in various areas such as transportation, healthcare, and smart homes. For an IoT device to be really useful and intelligent, it has to be connected to the Internet. There are three practical ways to do so today: using a cellular network, WiFi, or other radio technology such as Bluetooth. Connecting directly using a cellular network would be too costly. While it is cheaper to use WiFi technology, it would be too energy-consuming. Therefore, an easy-to-deploy, cost-effective, and low power solution is a key requirement for IoT devices, especially for smaller ones.

Among all the radio technologies, Bluetooth Low Energy (BLE) stands out and has been increasingly used by the IoT devices. It is well suited for applications with small amounts of data transferring as well as devices that require extremely low power consumption. For instance, with BLE, an IoT device is even able to run for years on a coin-cell battery. This is particularly appealing for industries such as sports, healthcare, fitness, retail, and home entertainment. In fact, many apps today have been built atop BLE connected IoT devices in these application scenarios such as computer gaming, fitness tracking, and indoor positioning.

Since there is a large amount of private information (e.g., health information in the wearable devices) collected by BLE IoT devices, there have been many attacks against them and their apps. For example, it has been discovered that BLE communication is subject to man-in-the-middle (MITM) attack [17]. BLE credentials can be sniffed [30, 42]. BLE devices can be penetrated [13]. Both mobile apps and BLE devices can be spoofed [27, 29]. In addition, the connection between BLE devices and mobile apps can be reused by unauthorized co-located apps [31]. These vulnerabilities were caused by a number of reasons such as a lack of secure pairing (e.g., via "Just Works" pairing protocol) or (weak) traffic encryption (e.g., no public key exchange in BLE v4.0 and v4.1 [30]) between devices and apps.

While the recent development of BLE IoT devices has started to involve authentication between devices and apps where users need to enter credentials to connect with the IoT devices, in this work we discover that many BLE IoT devices and mobile apps today actually do not properly implement the app-level authentication. In particular, we find that a great number of BLE IoT devices use "Just Works" for pairing (no invocation of app-device bonding at all) , which allows any nearby attackers to arbitrarily connect to them and possibly compromise device data and user privacy. Second, even though they have app-level authentication, some of the implementations are flawed and the credentials can be directly extracted.

With such vulnerable BLE IoT devices and apps, malicious attackers can easily break into the defense of these weakly authenticated IoT devices and access the privacy-sensitive data in them. While there could be still a number of challenges, such as how to identify the vulnerable devices from various BLE peripherals around them, we fortunately discovered that the universally unique identifier (UUID) from the advertisement packets broadcast by the BLE devices can fingerprint a BLE device and these broadcast packets are not encrypted at all. In addition, these UUIDs can be obtained from not only the BLE traffic but also the IoT companion mobile apps.

Astonishingly, UUID-based fingerprinting is universal and hard to defeat. It comes from the fundamental design flaw requiring BLE advertisement packets to contain predetermined UUIDs that must be known to the nearby mobile apps otherwise the apps will not be able to discover the BLE devices. Therefore, as demonstrated in this paper, if attackers can first scan all mobile apps in an app store, such as Google Play, to find all possible UUIDs, they can fingerprint all BLE devices statically. Then with the fingerprinted UUIDs, they can sniff all nearby advertisement packets in the field (e.g., a metropolitan area such as New York City) to locate these devices based on the fingerprinted UUIDs, thereby leading to a serious privacy attack. If mobile apps also tell them that the device uses "Just Works", or has weak or no authentications, then the attackers can directly exploit these BLE devices.

To validate our discovery and raise the public awareness, in this paper, we develop an automatic tool BLESCOPE to scan the vulnerable BLE devices directly from mobile apps in Google Play. Not all BLE devices are of our focus. Instead, we particularly focus on the devices that are vulnerable to device UUID fingerprinting (privacy attack), eavesdropping (including both passive and active), and unauthorized access. Our key objective is to identify these devices directly from mobile apps. With automated binary analysis techniques such as backward slicing [26] and value-set analysis [12], we have implemented BLESCOPE to automatically scan mobile apps to directly recognize the UUIDs, and identify insecure bonding (such as "Just Works") and vulnerable app-level authentications. We have tested BLESCOPE with all of the free Bluetooth apps in Google Play, in which our tool discovered 168, 093 UUIDs (13, 566 unique ones) as well as 1, 757 vulnerable apps.

While typically BLE signals can only travel up to 100 meters, with special receiver adapters and amplifiers, an attacker can sniff the BLE signals up to 1,000 meters [8]. In this work, we have actually built such a long range passive BLE sniffer with a Raspberry-PI and a special BLE Antenna, and used it to detect real world instances of these insecure BLE IoT devices. In a small area of 1.28 square miles, our sniffer identified 5, 822 BLE devices, among which 5, 509 (94.6%) of them are fingerprintable based on the UUIDs extracted from mobile apps. We also located 431 vulnerable devices, including 369 eavesdroppable devices and 342 unauthorizable access devices in this area.

**Contributions.** In short, we make the following contributions:

- We are the first to discover that vulnerable BLE IoT devices can be directly identified and fingerprinted due to the use of pre-determined static UUIDs in both mobile apps and BLE devices for BLE advertisement.
- We have implemented an automatic tool BLESCOPE using binary code analysis to directly scan mobile apps to harvest UUIDs and meanwhile detect insecure IoT devices such as those vulnerable to eavesdropping or completely taken over.
- We have tested our tool with 18, 166 BLE mobile apps from Google Play store, and found 168, 093 UUIDs (13, 566 unique ones) and 1, 757 vulnerable BLE IoT apps.
- We also present a set of countermeasures against the attacks from three dimensions: channel-level protection, app-level protection, and protocol-level protection (with dynamic UUID generation).

**Roadmap.** The rest of this paper is organized as follows. We provide necessary background related to Bluetooth Low Energy and its security in §2. Next, we describe the threat models and define the analysis scope in §3. Then, we present an overview of BLESCOPE in §4, detail the design and implementations in §5, and present the evaluation results in §6. In addition, we provide a set of countermeasures against our attack in §7. Next, we discuss limitations and future work in §8, followed by related works in §9. Finally, we conclude in §10.

## 2 BACKGROUND

### 2.1 Bluetooth Low Energy

Bluetooth Low Energy (BLE), also known as Bluetooth 4.0, is a wireless network technology designed for devices consuming extremely low energy, and is ubiquitous in our daily lives. It has been widely deployed in various platforms including desktops, mobile smartphones, and IoT devices for various applications such as health care, entertainment, and smart home. Figure 1 provides an overview of the important procedures within the BLE workflow, which includes (I) Connection, (II) Pairing and Bonding, and (III) Communication.

**(I) Connection.** When a BLE peripheral device wishes to establish a connection, it will first constantly broadcast its advertisement packets to indicate its willingness. The time interval between the advertisement packets ranges from 20ms to 10.24 seconds, in step of

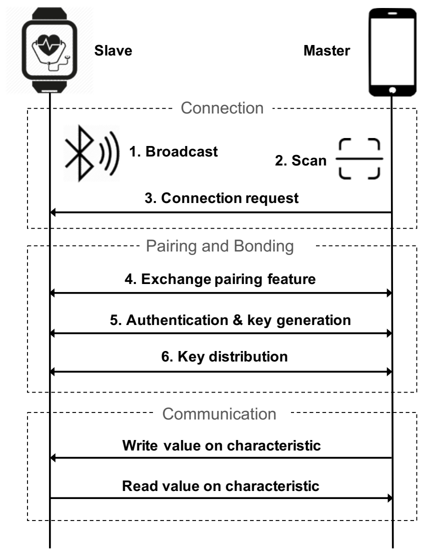
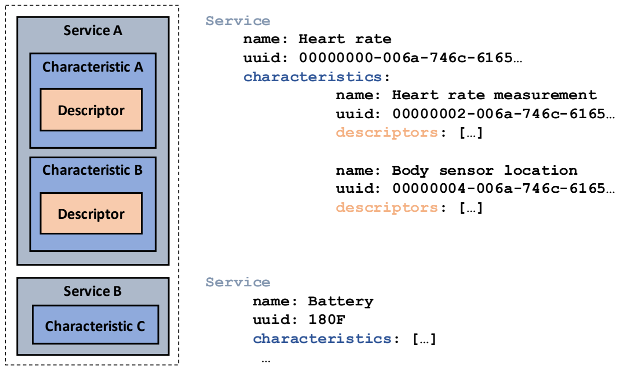

Slave
Master
Connection
1. Broadcast
2. Scan
3. Connection request
Pairing and Bonding
4. Exchange pairing feature
5. Authentication & key generation
6. Key distribution
Communication
Write value on characteristic
Read value on characteristic

**Figure 1: Bluetooth Low Energy workflow.**



```
Service
    name: Heart rate
    uuid: 00000000-006a-746c-6165…
    characteristics:
            name: Heart rate measurement
            uuid: 00000002-006a-746c-6165…
            descriptors: […]

            name: Body sensor location
            uuid: 00000004-006a-746c-6165…
            descriptors: […]


Service
     name: Battery
     uuid: 180F
     characteristics: […]
      …
```

**Figure 2: Illustration of GATT hierarchy.**

0.625ms, with an optional random delay ranging from 0ms to 10ms added to avoid collisions [4]. Note that BLE advertising is one of the most important aspects of BLE, and is mandatory for connection between two BLE devices before communicating with each other. During broadcast stage, all recipients within the range can scan and receive all the advertised packets, and then decide to initiate a connection with it. The peripheral that broadcasts advertisement packets is called "*slave*", while the other that scans for packets and actively initiates the connection is called "*master*". For example, in a typical IoT scenario, a peripheral IoT device (e.g., a wearable device) will broadcast packets to indicate its presence so that a companion mobile app can scan for it and establish a connection.

**(II) Pairing and Bonding.** Right after the connection is established, the master and the slave will start a pairing process, which aims at establishing a secure channel by negotiating an encryption key for communication. In general, they first exchange their pairing features (e.g., input and output capabilities such as keyboard and display) to decide which pairing protocol should be adopted. There are usually four pairing protocols, including "Just Works", "Passkey Entry", "Numeric Comparison" and "Out of Band (OOB)" [2]. For example, "Passkey Entry" requires a user to enter a password to the peripheral so input capability is required on the slave device. Next, the key exchange process starts on the two devices to negotiate a long term key based on the protocol. As for "Just Works", it is the only choice when the previous protocols are not applicable. After the pairing process, a long term key (LTK) will be generated for data encryption between the two devices. The bonding process will store the LTK for later communication to make sure the established channel is encrypted.

**(III) Communication.** After pairing and bonding, the master and the slave are able to exchange data. The structure of data strictly follows the Generic Attribute Profile (GATT) [7], which typically has a hierarchical structure as illustrated in Figure 2.

A BLE device usually provides several services each of which represents a specific property. Each service involves a number of characteristics that store the actual values for the property. Additionally, several descriptors are defined in a characteristic for description purpose. Considering a heart rate monitor as an example, it provides services including heart rate, battery, and device information, as illustrated in Figure 2. For the heart rate service, characteristics are defined for various properties such as heart rate measurement and body sensor location, which contain the actual values of heart rate and location [40]. There are two ways for a master device to know the provided services and characteristics from the slave. First, the master can browse the service list from the advertisement packets, which is optionally defined. Second, a master can request for a list of supported services and characteristics after the connection is established.

**UUID.** A key ingredient in BLE world is the universally unique identifier, short as UUID, which is a hexadecimal string used to identify a specific BLE attribute including service, characteristic, and descriptor. In general, BLE UUIDs can be categorized into two types: standard ones and customized ones. The standard UUIDs are defined and documented by the Bluetooth SIG [3], which share a common base and use a 16-bit invariant to uniquely identify themselves. The customized UUIDs are arbitrarily generated and are vendor-specific. Note that the customized UUIDs cannot collide with the standard ones on the common base [39].

The UUIDs are involved in the BLE packets along with their indicated attributes such as services and characteristics if there is any. They can be harvested either from the advertisement packets or data packets after the connection is established. Also, interestingly, for a mobile app to connect with a particular BLE device, it needs to know the UUIDs of the device (which will be checked against the ones in the advertisement packet). Therefore, UUID can also be extracted from the BLE IoT mobile apps.

## 2.2 The Security of BLE

Since BLE data is transferred over the air, it is important to protect the communication channel between the BLE slave and master. To this end, BLE supports link layer encryption which is transparent to the applications. However, the exchange of the cryptographic keys for both encryption and authentication significantly varies depending on the user interfaces (e.g., keyboards, displays) provided by the BLE devices, and how the BLE devices and mobile apps are paired. For instance, a great number of IoT devices do not have any user interface such as keyboard that allows external input so that some pairing protocols such as "Numeric Comparison" and "Passkey Entry" are not available. Also, "OOB" is not always practical since not all devices contain Near Field Communication (NFC). Therefore, many of the IoT devices today use "Just Works" pairing, which can be insecure since they may be vulnerable to Man-In-The-Middle (MITM) attack. Note that, technically "Just Works" is just a special model of "Passkey Entry", in which the passkey is a hardcoded PIN (e.g., 000000 or 123456) [32]. Furthermore, application programmers can introduce additional app-level authentication if they have to use "Just Works". For instance, they can ask users to enter credentials from mobile apps, and deliver (through encryption) to the IoT devices to authenticate the apps. That is, with additional cryptography protection, the app and device can still establish a secure channel.

## 3 THREAT MODEL AND SCOPE

BLE devices are ubiquitous today. In this work, we focus on BLE devices that use mobile apps (as the gateway to connect to the Internet and interact with the users). In the following, we discuss the possible attackers (§3.1), and the scope of attacks under our consideration (§3.2).

### 3.1 Types of Attackers

According to where the attackers are located, there could be two types of attackers against a BLE IoT device:

- **Nearby attackers.** Being a short-range radio technology, BLE devices can communicate with a peripheral within at most 100 meters [25]. Any nearby attackers who are in this range could attack the device. While this may constrain many attackers, we also learn that there are long range Bluetooth adapters that support the sniffing of BLE devices up to 1,000 meters [8].
- **Remote attackers.** Since BLE IoT devices are accessed by mobile apps, there could be remote attackers through malware in the phone. There are various ways for attackers to install malware in the phone (e.g., social engineering, backdoors, or direct software vulnerability exploitation). However, unlike nearby attackers who can attack at anytime, the remote attackers have to rely on the nearby mobile devices that need to be power-on.

### 3.2 Types of Attacks of Our Interest

While malware in the phone can attack a BLE device, we exclude it in our scope since the attacks performed by malware could be so broad, especially if the malware has obtained the root privilege of the phone. Therefore, in this work, we particularly focus on the nearby attackers and systematically understand their attack capabilities. To this end, we have actually built a long range BLE sniffer with a Raspberry-PI and a special BLE Antenna (with a cost below $150) that can scan the BLE devices in the range of 1, 000 meters.

With respect to the nearby attackers, there could be two types of major attacks: passive attacks that only listen to the BLE traffic, and active attacks that can aggressively connect, pair, read, and even write to the devices.

**Passive Attack.** This attack can be launched by passively sniffing the BLE traffic and obtaining information from the packets. Note that only BLE advertisement packet is in plain-text, and all other packets are encrypted at the BLE link layer. As such, there could be two types of passive attacks:

- **Passive Fingerprinting.** Through sniffing the BLE advertisement packets, the attacker is able to obtain the advertisement UUIDs. By knowing the UUIDs, the attacker can fingerprint which IoT devices and also the corresponding mobile apps the victims are using, especially if the UUID and mobile app have a one-to-one mapping. Moreover, if certain privacy sensitive BLE devices (e.g., blood pressure monitors) are used, attackers can even learn some privacy knowledge about the victims.
- **Passive Eavesdropping.** Since BLE packets except the advertisement ones are encrypted, attackers need to sniffer the cryptographic keys in order to get the plain text of the intercepted traffic. This attack can succeed if the BLE device and mobile app use "Just Works" for the pairing [24], and the BLE version is before 4.2 (since Elliptic Curve Diffie-Hellman (ECDH) key exchange is introduced in this version to prevent the sniffing of long-term key [1]). This is because "Just Works" before 4.2 uses the hardcoded short-term key (e.g., 000000) to encrypt the long-term key, which can be sniffed if the attacker constantly listen to the communication channel. However, if there is any app-level encryption on the transferred data, this passive sniffing will not succeed.

**Active Attack.** Unlike passive attacks that only listen to the traffic, which could learn a significant amount of privacy information of the victim from either the plain-text UUID or the decrypted traffic if the long-term key is obtained, active attacks can cause direct damages to the victim if the vulnerable BLE devices allow them to do so. More specifically, there could be four types of active attacks:

- **Denial of service (DoS) attack.** Due to the reason that one peripheral BLE IoT device is usually designed to connect with only one master at the same time, it is possible to conduct denial-of-service (DoS) attack by the nearby attackers. They just only need to constantly listen to the traffic, and if there is an advertisement packet and then the attackers can connect with the device (disallowing legitimate users to use the device). Since DoS attack is trivial to launch, we do not consider it in this work.
- **Active Fingerprinting.** If in passive fingerprinting, multiple mobile apps contain the same UUID, which may be caused by multiple apps use the same scheme-specific BLE chip or UUID configuration, then the nearby attackers cannot precisely know which device the victim is using and may

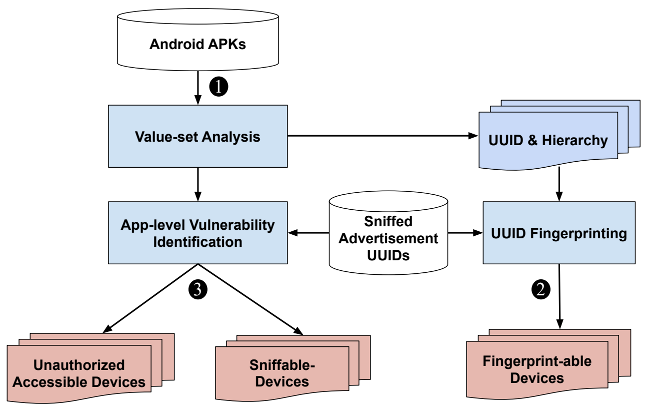


**Figure 3: Overview of BleScope.**

need to further narrow it down. To this end, an active attacker can further connect to the devices to inspect the next layer UUIDs (recall BLE devices often organize UUIDs in a hierarchical structure), and use the hierarchical structure of the UUIDs to fingerprint a victim BLE device.

- **Active Eavesdropping.** If a BLE device uses version after 4.2, then "Just Works" pairing will use ECDH key exchange. Consequently, passive eavesdropping will not work any more, and attackers must perform active MITM attack to gain an attack controlled long-term key.
- **Unauthorized access attack.** An unauthorized access is the most dangerous vulnerability for the IoT devices, since attackers can read or even write sensitive data to the devices. This attack can succeed if a device uses "Just Works" pairing and meanwhile there is no (or flawed) app-level authentication.

## 4 OVERVIEW

### 4.1 Objectives and Attack Overview

**Objectives.** The key objective of this work is to systematically investigate, from an adversary perspective, the insecure IoT devices that are attackable to a nearby attacker, and meanwhile the various attacks that can be launched by attackers in a specific region (e.g., a metropolitan city such as New York City). To this end, we have to develop an automated program analysis tool, and we name it BleScope that takes the following input and produces the following output:

- **Input.** The input to our attack is all of the IoT Android apps available from Google Play Store, along with the advertisement UUID sniffed by our passive sniffer in the region.
- **Output.** The output is the specific IoT-devices that are vulnerable to Fingerprinting, Traffic sniffing/eavesdropping, and Unauthorized access attacks (described in §3.2).

**Tool Overview.** An overview of our BleScope is presented in Figure 3. At a high level, it consists of three steps of analysis:

- **Step ❶.** When given a set of Android IoT apps, it first performs value-set analysis [12] on the low-level APIs. On one hand, it produces the resolved UUIDs and the reconstructed hierarchies for UUID fingerprinting (Step ❷). On the other hand, it identifies the data-definition and data use of BLE-transmitted data, the involved APIs and cryptographic operations, for app-level vulnerability identification (Step ❸).
- **Step ❷.** When provided with the extracted UUIDs and hierarchies, as well as the real world traces of the field UUIDs, it then identifies the fingerprint-able IoT devices based on the app that contains the UUID. Note that one UUID in the filed may map to multiple apps due to the use of the same scheme-specific BLE chip or UUID configuration, and it may need to further narrow down the device by gaining more data by connecting with the device.
- **Step ❸.** Next, it identifies devices that are vulnerable to sniffing or unauthorized access from the app code, based on whether there is any flawed-authentication, or no authentication at all, among the devices that use "Just Works" (since the other three pairing is considered secure and we cannot directly test them with only mobile apps). BleScope takes two disjoint approaches. One is to inspect whether the app has used any cryptographic functions for authentication. If not, it implies the data transferred between the app and the device is vulnerable to sniffing attack. The other one is to detect whether there is any flawed authentication implementation (even though it has used cryptographic functions).

### 4.2 Challenges

There are a number of challenges we must solve in order to identify the nearby vulnerable BLE IoT devices. In particular, we must extract the UUIDs, reconstruct their hierarchies from the mobile apps and identify whether there is insecure pairing as well as app-level vulnerability including absent cryptography usage and flawed authentication. While it may be easier to identify the insecure pairing and cryptography APIs from the disassembled app code, it is actually challenging to identify the UUIDs and its hierarchy structure, as well as the flawed authentication directly from app code. Therefore, we have the following three major challenges.

**C1: UUID extraction.** UUIDs play an important role in BLE communication, not only for advertisement (such that the nearby mobile app knows) but also for accessing each specific service provided in the BLE. UUIDs are typically 128-bit hexadecimal strings which can be found in the BLE packets along with the BLE attributes such as services and characteristics. When providing a mobile app, if UUIDs are directly hardcoded in the app, then it is easier to extract them (by simply `grepping`). For instance, as shown in Figure 4, which is the decompiled code from a real world BLE IoT thermometer companion app Kinsa, we can see clearly there are a number of UUIDs that are hardcoded in the app as constant strings and they can be easily extracted.

However, we also notice that UUID may be generated through some complicated calculations (e.g., string concatenation, and bit shifting). Therefore, we have to design a principled approach to extract these UUIDs from the mobile app binaries.

**C2: UUID hierarchy reconstruction.** One single UUID may not directly fingerprint a BLE device and we need more information

```
public class KelvinDeviceProfile {
  private KelvinDeviceProfile(BlueToothLeGatt arg3) {
    super();
    BluetoothGattService v0 = arg3.getService(KelvinGatt.KINSA_SERVICE);
    if(v0!=null) {
      this.request = v0.getCharacteristic(KelvinGatt.REQUEST_CHARACTERISTICS);
      this.response = v0.getCharacteristic(KelvinGatt.RESPONSE_CHARACTERISTICS);
    }

    BluetoothGattService v3 = arg3.getService(KelvinGatt.BATTERY_SERVICE_UUID);
    if(v3!=null) {
      this.batterylevel = v3.getCharacteristic(KelvinGatt.BATTERY_VALUE_CHAR_UUID);
    }
  }
}

public class KelvinGatt {
  public static final UUID KINSA_SERVICE = UUID.fromString("00000000-006a-746c-6165-4861736e694b");
  public static final UUID REQUEST_CHARACTERISTICS = UUID.fromString("00000004-006a-746c-6165-4861736e694b");
  public static final UUID RESPONSE_CHARACTERISTICS = UUID.fromString("00000002-006a-746c-6165-4861736e694b");
  public static final UUID BATTERY_SERVICE_UUID = UUID.fromString("0000180F-0000-1000-8000-00805f9b34fb");
  public static final UUID BATTERY_VALUE_CHAR_UUID = UUID.fromString("00002A19-0000-1000-8000-00805f9b34fb");
}
```

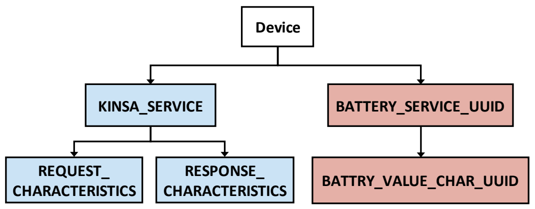


**Figure 4: Decompiled code snippet from IoT app Kinsa showing UUIDs extraction and their hierarchical structures.**

about the UUID. Interestingly, we notice that UUIDs associated with an IoT device typically have a hierarchical structure. As shown in Figure 4, a service contains multiple characteristics and thus a service UUID can have "children" UUIDs from its characteristics. Such a UUID hierarchy could perfectly provide additional information to accurately determine which IoT app maps to a particular BLE device.

Therefore, it is necessary to reconstruct hierarchical structure of the UUIDs for our IoT app and device fingerprinting. Unfortunately, the hierarchy cannot be directly inferred from the app code because there is no structural rule on defining parent and children UUIDs.

**C3: Flawed authentication identification.** For a nearby attacker to sniff the encrypted traffic or gain unauthorized access to "Just Works" paired IoT devices, the corresponding apps must not use any app-level authentication or use flawed authentication. To implement proper authentication, we assume app must use cryptography to either encrypt the authentication token with nounces (preventing replay attack) or even use additional layer of encryption of the traffic atop BLE link layer encryption. Therefore, if we cannot find any use of cryptography in the app code, then we can conclude the channel is not secure (both passive/active sniffing and unauthorized access can be performed on the devices).

Meanwhile, we also notice flawed authentication in the app. For instance, all the credentials are hardcoded in the app. Therefore, even though the app has used the cryptography, this is still not secure. However, how to identify these flawed authentication is a challenge since there is no specific code pattern on implementing authentication in the apps, and we are not able to rely on any documented APIs to identify them and extract the hardcoded credentials.

## 4.3 Our Solutions

While each of the above challenges sounds hard to address, fortunately we can develop mobile app binary analysis techniques to solve them. More specifically, we have the following corresponding solutions:

**S1: Resolving UUIDs using context and value-set analysis.** Directly `grepping` strings in app byte code may reveal some UUIDs, especially those hardcoded ones. However, it may have false positives if a UUID is never used by the app or other type of UUIDs (in fact, many Java objects in Android also have UUID that has the same format as the BLE UUID), and it will also have false negatives if the UUID is dynamically generated through computation. We have to design a principled program analysis approach to resolve and extract UUIDs.

Fortunately, we notice that while we may not know the concrete value of the final UUIDs used by the app statically, we actually know where the UUIDs are used (i.e., the execution context). In particular, we find that there are seven documented APIs defined by the Android BLE framework that carry the UUIDs as parameters, to generate the instances for accessing the related service, characteristic and descriptor in the paired BLE devices [6]. For instance, the UUIDs in Figure 4 are used as arguments by the official BLE API `getService()` and `getCharacteristic()`. Therefore, we can target these APIs to extract UUIDs from the app.

In addition to extracting those constant string UUIDs, we can also compute those that are not hardcoded ones by using program slicing [38] and value-set analysis (VSA) [12], which aims at statically tracking the values of data object and is an effective solution to our problem. Note that VSA was originally designed to resolve possible values for registers and memory cells on x86 platforms, and we have to implement it to extract and compute UUIDs in Android app code. In fact, one of our prior projects LeakScope [43] has also leveraged VSA.

**S2: Reconstructing UUID hierarchy with control dependence.** While an app could have multiple UUIDs, the usage of them actually has dependencies. In particular, the instance of a GATT service must be first initialized by BLE API `getService()` with the corresponding UUID, and then all of its characteristics can be browsed by API `getCharacteristic()`, or updated with specific data to the peripheral according to the characteristics. Therefore, characteristic must be derived from the service instance, and `getService()` must be executed first, then followed by `getCharacteristic()`, forming a control dependence. Similarly, a descriptor must be derived from the corresponding characteristic instance.

Meanwhile, the same layer of the characteristic is often guarded by control dependence as well. For instance, as shown in Figure 4, at line 4, variable `v0` (a GATT service) first initialized by API `getService()`, then between line 5 and line 8, two characteristics of `v0` are derived, and they both are guarded by control statement at line 5. Therefore, the UUIDs of `REQUEST_CHARACTERISTICS` and `RESPONSE_CHARACTERISTICS` are at the same layer and they are siblings. Also, we can learn from Figure 4 that there are two services `KINSA_SERVICE` (initialized at line 4) and `BATTERY_SERVICE_UUID` (initialized at line 10), and these two services are also siblings. Therefore, a hierarchical tree structure of UUIDs can be derived based on the app code.

**S3: Identifying flawed authentication with data dependence.** When using "Just Works" pairing, a device would be vulnerable if there is no application-level authentication or the authentication is useless. Unfortunately, application-level authentication can be implemented in completely different ways across apps, and it is therefore extremely challenging to design a general approach to identify flawed authentication.

However, we notice there is one special type of flawed authentication which can be identified systematically: the case that uses hardcoded credentials, and we can model this case using a data flow analysis. Our key insight is that to securely authenticate a mobile app to a BLE device, the app must provide a credential that comes from the external input, such as letting the user enter a password. Otherwise, if all of the commands sent out are hardcoded, then there will be no authentication at all. Therefore, we can use a data flow analysis algorithm to identify these apps. In particular, since all the data sent out to the peripheral must go through low-level BLE APIs, so called the "sink", starting from the sink, we can apply program slicing to trace back to the "source" of the data. If none of the sources comes from any external input (e.g., network return, user input), then the app has used hardcoded commands including possible passwords if there is any to interact with the BLE devices.

## 5 DETAILED DESIGN

In this section, we present the detailed design of BleScope, which aims at (*i*) extracting the value of each UUID and reconstructing the hierarchy of each group of UUIDs, and (*ii*) identifying vulnerable implementation in terms of absent cryptographic usage and flawed authentication. As the fundamental technique of our tool, we first explain how our value-set analysis works in §5.1, and then describe how to configure the value-set analysis technique to achieve the above two purposes in §5.2 and §5.3, respectively.

### 5.1 Value-set Analysis

At a high level, the value-set analysis (VSA) consists of backward program slicing and forward value computation. In this work, we leverage VSA to locate the definition of parameter of our interest and obtain its computation path to the destination. In particular, for a given list of target APIs and their parameters of our interest, our VSA runs as follows:

- **Backward slicing.** Program slicing [38] is a technique to identify program code of interest based on slicing criteria (e.g., control dependence or data dependence), which has been widely used to tackle important program analysis problems. Given the target system APIs and parameters, the backward slicing algorithm starts from constructing a control flow graph (CFG) where each node of the graph represents a block, which contains a set of instructions without transfers, and every edge connecting blocks indicates a transfer of program control flow. Based on the CFG, the backward slicing starts tracing from the instruction which contains our targeted APIs (i.e., the sink) backwardly to the definition (i.e., the source) of the interested parameter. Specifically, it iterates each instruction within each block and records the instructions that either have data dependency with the interested parameter (i.e., modifying its value) or control dependency with recorded instructions (i.e., preceding condition). These recorded instructions and parameters are maintained in the instruction stack and parameter stack respectively. We implement our backward slicing in a flow-sensitive and context-sensitive manner, where we fork the instruction stack at each branch, and maintain each stack separately.
- **Value computation.** With the sliced instructions, the value of each target parameter can be computed by following the execution path from its source to its sink. This process is automated by continuously popping the top instruction on the stack and simulating its execution of the instructions if it is data-arithmetic or well-known APIs according to its definition provided by the official documentation until the stack is empty.

The extraction of the UUIDs, their hierarchy and the vulnerability identification are all built atop our backward slicing and value computation. In the following, we describe how we configure it to achieve these.

### 5.2 UUID Extraction and Hierarchy Reconstruction

The first step for UUID extraction and the hierarchy reconstruction is to acquire a list of target APIs that are related to the UUID generation and its hierarchy. We have identified seven system APIs according to the documentation from official Android framework, and the details of these APIs and their parameters are presented in Table 1, where the target parameters are highlighted in the last column. In particular, the extraction of UUID depends on all seven APIs listed in Table 1 for the corresponding category of UUID, while the hierarchy reconstruction relies only on the first three of them.

Unlike the extraction of UUID that can be resolved by the standard procedure of value-set analysis, the reconstruction of hierarchy requires extra processes. To systematically solve this problem, we designed and implemented a hierarchy reconstruction algorithm, which is presented in algorithm 1. More specifically, the algorithm starts with the initialization process (line 2-3), where the value-set analysis (VSA) procedure is triggered when encountering the two functions `getDescriptor()` and `getCharacteristic()`. Note that the VSA is invoked with a focus on the base of the function (i.e., the variable $v0$ of $v0$`.getDescriptor(`$v1$`)`), to obtain the program slice of the descriptor and characteristic instances. A tree node $T$ is initialized to record the hierarchy of the UUIDs (line 4). Next, the algorithm iterates through all the program slices obtained from API `getCharacteristic()` (line 5). For each value-set result of the slice $S_D$, the algorithm adds all the (characteristic, descriptor) pairs

| Category | API Name | Parameters |
|---|---|---|
| UUID | BluetoothGatt: BluetoothGattService getService | **UUID uuid** |
| | BluetoothGattService: BluetoothGattCharacteristic getCharacteristic | **UUID uuid** |
| | BluetoothGattCharacteristic: BluetoothGattDescriptor getDescriptor | **UUID uuid** |
| | ScanFilter.Builder: ScanFilter.Builder setServiceUuid | **ParcelUuid serviceUuid** |
| | ScanFilter.Builder: ScanFilter.Builder setServiceUuid | **ParcelUuid serviceUuid**, ParcelUuid uuidMask |
| | ScanFilter.Builder: ScanFilter.Builder setServiceData | **ParcelUuid serviceDataUuid**,byte[] serviceData |
| | ScanFilter.Builder: ScanFilter.Builder setServiceData | **ParcelUuid serviceDataUuid**,byte[] serviceData,byte[] serviceDataMask |
| BLE | BluetoothGattCharacteristic: boolean setValue | **String value** |
| | BluetoothGattCharacteristic: boolean setValue | **int value**,**int formatType**,**int offset** |
| | BluetoothGattCharacteristic: boolean setValue | **byte[] value** |
| | BluetoothGattCharacteristic: boolean setValue | **int mantissa**,**int exponent**,**int formatType**,**int offset** |
| Cryptography | Cipher: byte[] doFinal | **byte[] data** |
| | Mac: byte[] doFinal | **byte[] data** |
| | MessageDigest: byte[] digest | **byte[] data** |

**Table 1: Targeted APIs for BleScope**

**Algorithm 1:** UUID hierarchy reconstruction.

```
Input  : G: The control flow graph
Output : T: A hierarchy tree
1  Function HierarchyReconstuction(G)
2      $S_D \leftarrow VSA(G, SIG\_GETDESCRIPTOR, base)$
3      $S_C \leftarrow VSA(G, SIG\_GETCHARACTERISTIC, base)$
4      $T \leftarrow$ New TreeNode
5      for $slice \in S_D$ do
6          for $(characteristic, descriptor) \in slice.valueset$ do
7              Add the characteristic-descriptor UUID binding to $T$
8          end
9      end
10     for $slice \in S_C$ do
11         for $(service, characteristic) \in slice.valueset$ do
12             Add the service-characteristic UUID binding to $T$
13         end
14     end
15     return $T$
```

to $T$ (line 6-9). Similarly, the service-characteristic hierarchy can also be reconstructed (line 10-14). Finally, the algorithm outputs a tree node $T$ that stores all the hierarchical information (line 15).

## 5.3 Vulnerability Identification

The focus of this step is to identify vulnerable implementations in term of absent cryptographic usage and flawed authentication. In addition, the precondition of these two vulnerable implementation is that a BLE IoT device uses the "Just Works" paring, which indicates that the BLE channel is insecure due to the fact that the long term encryption key can be sniffed by nearby attackers. Therefore, the first step in this procedure is to recognize whether such precondition exists in an app.

To recognize the existence of such precondition, BleScope has to look into how an app pairs with its BLE IoT devices. According to the official Android BLE developer guide, there are two ways for an app to implement a secure pairing process: one is to invoke the `createBond()` API and the other one is to define responses when receiving a system broadcast event that carries `ACTION_BOND_STATE_CHANGED`, a constant value indicating a change in the bond state [5]. If neither of these two implementations exists in the app code, then the app can be concluded for using "Just Works" paring with the peripheral. Therefore, BleScope recognizes an app implementing the "Just Works" paring if no such pairing process can be identified.

**Absent cryptographic usage detection.** The detection of this vulnerable implementation depends on identifying whether the data exchanging between an app and its BLE IoT device is encrypted. In other words, we have to check whether the generation of the data for exchanging involves encryption. To this end, we first create a list of four system APIs whose parameters carry the data for communication, which is shown in Table 1 under the category of "BLE". With this list of target APIs and parameters, we backward iterate each instruction that is related to them, which can be obtained by backward slicing in the value-set analysis, to detect whether it involves encryption or hashing by comparing with the list of cryptographic APIs, part of which are shown in Table 1. If no encryption or hashing is detected, then the app is recognized as vulnerable.

**Flawed authentication.** The identification of flawed authentication is to detect if all the data sent out to the peripheral is generated with hardcoded sources, which means it is possible to recover the value based on the program code. For this regard, the target APIs and parameters are the same as that for absent cryptographic usage detection. Unlike the algorithm to detect the existence of cryptographic operation, here we only focus on all the sources, which can be identified by the backward slicing, that contribute to the final data sent to the BLE IoT device. If all sources are hardcoded (no external input), then the final data should definitely be considered hardcoded.

## 6 EVALUATION

We have implemented a prototype of BleScope with a number of open source tools and hardware components. In particular, we implemented the Android app analysis component atop Soot [9], which is a powerful and popular static analysis framework for reverse engineering of Android apps. We implemented ou BLE devices sniffer with Raspberry-PI running Linux connected with a SIM7000A GPS module, and a special Bluetooth adapter, Parani-UD100, with an amplified Patch Antenna RP-SMA-R/A. In this section, we present our evaluation results. We first describe how we setup the experiment (§6.1), then present our analysis results with mobile apps (§6.2), followed by the field test result (§6.3).

## 6.1 Experiment Setup

**BLE IoT Apps Collection.** In this work, we focus on analyzing BLE IoT apps that are available from the Google Play. However, the Google Play does not provide information that directly indicates an app is of BLE IoT type. Therefore, we apply a heuristic to find these apps. In particular, we first checked whether an app has Bluethooth related permissions in its manifest file, and we found 135, 359 of them out of 2 million free apps crawled from the Google Play as of April 2019. Since we only focus on BLE related apps, not the classic Bluetooth apps, so we checked each of them to see if the app invokes BLE related APIs (shown in Table 2) and we found 68, 908 apps. However, many of them are beacon apps, e.g., Macy's, which only invoke scanning related APIs (the first 6 APIs in Table 2), which means these devices will not be connected (are not within our attack scope). Consequently, we further identified the apps supporting BLE connection by searching for after-connection BLE APIs (the 7th to 11th APIs in Table 2). Eventually, we found 18, 166 BLE IoT apps for our analysis, as reported in the first row of Table 3.

**Environment setup.** Our evaluation consists of two sets of experiments: the static analysis of mobile apps, and the passive sniffing of advertisement UUIDs in the field (due to ethics considertion, we did not perform any active operations with the devices). In particular, the static analysis including value-set analysis and vulnerability identification was conducted on a Linux server running Ubuntu 16.04 equipped by two Intel Xeon E5-2695 CPUs. The advertisement UUIDs sniffing was conducted by the sniffer we built.

## 6.2 Mobile App Analysis Result

It took approximately 96 hours to finish the analysis with these 18, 166 BLE IoT apps. At a high level, the experiment results of the static analysis are broken down into two parts: (1) UUID extraction and hierarchy reconstruction, and (2) Vulnerable app identification. The statistics and descriptions are presented in the following.

**Results of UUID extraction and hierarchy reconstruction.** In total, BleScope has extracted 168, 093 UUIDs from 18, 166 IoT companion apps. Many of these UUIDs are repeated , and only 13, 566 of them are unique. In addition to the UUIDs extraction, BleScope also reconstructed their hierarchies which are used for active fingerprinting. As shown in Table 3, our system reconstructed 316, 379 (58.5%) UUID hierarchy service edges and 224, 418 (41.5%) UUID hierarchy characteristic edges, which indicate 316, 379 service-characteristic pairs and 224, 418 characteristic-descriptor pairs. During the UUID analysis procedure, there are multiple UUIDs that cannot be directly identified because their generation involves computations, such as concatenation and shifting. In order to identify these UUIDs, BleScope has to use VSA to compute them. The statistics of these computations is presented in Table 4.

In addition, we also report the mapping between the UUIDs and the apps, since there could be multiple apps using the same UUID (that is why UUID hierarchy and active fingerprinting is needed). This result is presented in Table 5, where we group the UUIDs based on the number of apps they are mapped to. As shown in the table, a majority (65.4%) of the UUIDs can be uniquely mapped to only 1 app, which shows the corresponding devices can be easily passively fingerprinted. As for the remaining 34.6% UUIDs, they are mapped to multiple apps (which requires further active fingerprinting if needed to narrow them down).

We investigated the reasons of why multiple apps could use the same UUID and discovered that (*i*) multiple apps from the same vendors (e.g., HP) could manage the same single device (e.g., the printer); (*ii*) different apps (e.g., fitness apps) from different vendors can manage the same device as well (e.g., a wrist band); (*iii*) apps can reuse standard service UUIDs; (*iv*) apps accidentally share the same UUIDs. Therefore, while some UUIDs are mapped to several apps, they actually represent a specific kind of devices or those from the same manufacture, which still can be passively fingerprinted (and we can actually connect to these devices to fetch next layer UUIDs to uniquely fingerprint them if needed).

**Vulnerable mobile app identification.** The statistics of the apps whose BLE IoT devices are vulnerable to sniffing (both passive and active) and unauthorized access is summarized in Table 6. The identification process strictly follows the analysis steps defined in §4.1. Overall, from the 18, 166 apps we analyzed, BleScope has reported that 11, 141 (61.3%) apps adopt "Just Works" pairing which indicates that their BLE channel is insecure. Among the 11, 141 insecure apps, we further discovered that 1, 510 (13.6%) of them do not use any cryptographic function to encrypt BLE-related data, 1, 434 (12.9%) have implemented flawed authentication (i.e., hard-code their authentication credentials) or no authentication. Even worse, 1, 187 (10.7%) apps have both vulnerabilities, directly implying that their IoT devices are vulnerable to traffic sniffing and unauthorized access attacks. *Note that we applied a very strict rule (e.g., to make sure each data to be sent is hardcoded) to identify vulnerable apps. Therefore, we can have false negatives in our result.*

We categorized the vulnerable apps according to their categories on Google Play, and the distribution of vulnerable apps is shown in Table 7. Interestingly, we find that health and fitness apps contribute most to all the vulnerable apps, followed by tool apps, lifestyle apps and business apps. In addition, we further break down the statistics according to different vulnerabilities, which reveals that a majority apps adopt insecure pairing ("Just Works"), and it is equally common for apps to be vulnerable to sniffing attack (absent cryptographic usage) and unauthorized access (flawed authentication).

## 6.3 Field Test Result

Our mobile app analysis only tells "*what are those vulnerable IoT apps and their corresponding devices*". Then the next question is "*where are they located*". To answer this question and also for the demonstration of our attack, we took our long-range BLE advertisement packet sniffer and drove around a small area nearby our campus for a field test. To precisely locate where these devices are, we added a GPS location record to the first appeared UUIDs in our sniffer, along with the received MAC address to uniquely identify a particular BLE device. The summary of the scanned UUID statistics is shown in Table 8.

Overall, we have collected 30, 862 unique Bluetooth devices (based on the MAC address), and 5, 822 of them contain UUIDs,

| API Name | Parameters |
|---|---|
| BluetoothAdapter: void startLeScan | UUID uuid, BluetoothAdapter.LeScanCallback callback |
| BluetoothAdapter: void startLeScan | BluetoothAdapter.LeScanCallback callback |
| ScanFilter.Builder: ScanFilter.Builder setServiceUuid | ParcelUuid uuid |
| ScanFilter.Builder: ScanFilter.Builder setServiceUuid | ParcelUuid uuid0, ParcelUuid uuid1 |
| BluetoothLeScanner: void startScan | ScanCallback callback |
| BluetoothLeScanner: void startScan | List list, ScanSettings settings, ScanCallback callback |
| BluetoothGatt: List getServices | |
| BluetoothGatt: BluetoothGattService getService | UUID uuid |
| BluetoothGattService: UUID getUuid | |
| BluetoothGattService: BluetoothGattCharacteristic getCharacteristic | UUID uuid |
| BluetoothGattCharacteristic: UUID getUuid | UUID uuid |

**Table 2: Targeted APIs used to identify the BLE related IoT apps**

| Item | Value | % |
|---|---|---|
| # Apps Collected | 18,166 | |
| # UUID Identified | 168,093 | |
| # Unique UUID Identified | 13,566 | |
| # UUID Hierarchy Edges | 540,797 | 100.0 |
| # UUID Hierarchy Service Edges | 316,379 | 58.5 |
| # UUID Hierarchy Characteristics Edges | 224,418 | 41.5 |

**Table 3: Experimental result of UUID extraction and hierarchy reconstruction.**

| opcode | # operations | opcode | # operations |
|---|---|---|---|
| + | 79,743 | \| | 1,398 |
| / | 9,684 | & | 1,266 |
| * | 5,364 | >>> | 894 |
| << | 1,860 | ^ | 462 |
| - | 1,775 | >> | 17 |

**Table 4: The statistics of operations executed to resolve UUIDs.**

| # Apps Mapped to a Single UUID | Value | % |
|---|---|---|
| # 1 | 8,870 | 65.4 |
| # 2 | 1,831 | 13.5 |
| # 3 | 688 | 5.0 |
| # 4 | 469 | 3.5 |
| # 5 | 330 | 2.4 |
| # ≥ 6 | 1,378 | 10.1 |

**Table 5: The mapping between UUID and mobile apps.**

| Category | # App | "Just Works" | Absent Crypto | Flawed Auth. |
|---|---|---|---|---|
| Health & Fitness | 3,849 | 2,639 | 221 | 207 |
| Tools | 2,833 | 1,895 | 385 | 362 |
| Lifestyle | 2,173 | 1,081 | 147 | 141 |
| Business | 1,660 | 972 | 90 | 85 |
| Travel & Local | 967 | 582 | 90 | 87 |
| Productivity | 834 | 453 | 76 | 75 |
| Education | 562 | 377 | 44 | 43 |
| Sports | 526 | 296 | 50 | 49 |
| Medical | 496 | 223 | 41 | 39 |
| Entertainment | 443 | 302 | 53 | 49 |
| Auto & Vehicles | 418 | 285 | 52 | 44 |
| Maps & Navigation | 386 | 209 | 33 | 33 |
| Communication | 331 | 236 | 49 | 46 |
| Game | 285 | 227 | 24 | 24 |
| House & Home | 279 | 177 | 22 | 22 |
| Events | 263 | 51 | 2 | 2 |
| Food & Drink | 252 | 166 | 10 | 9 |
| Music & Audio | 243 | 144 | 8 | 8 |
| Finance | 239 | 96 | 10 | 10 |
| Beauty | 224 | 135 | 5 | 4 |
| Shopping | 195 | 135 | 9 | 9 |
| Photography | 162 | 96 | 21 | 20 |
| Libraries & Demo | 100 | 55 | 9 | 9 |
| Social | 100 | 62 | 9 | 9 |
| News & Magazines | 66 | 46 | 1 | 1 |
| Personalization | 62 | 48 | 13 | 13 |
| Books & Reference | 48 | 41 | 6 | 6 |
| Video Players & Editors | 48 | 33 | 11 | 9 |
| Art & Design | 45 | 31 | 7 | 7 |
| Weather | 40 | 23 | 8 | 8 |
| Parenting | 32 | 21 | 4 | 4 |
| Dating | 3 | 2 | 0 | 0 |
| Comics | 2 | 2 | 0 | 0 |

**Table 7: Distribution of the BLE IoT apps across category.**

| Item | Value | % |
|---|---|---|
| # Apps Support BLE (Fingerprintable) | 18,166 | 100.0 |
| # "Just Works" Pairing | 11,141 | 61.3 |
| # Vulnerable Apps | 1,757 | 15.8 |
| # Absent Cryptographic Usage (Sniffable) | 1,510 | 13.6 |
| # Flawed Authentication (Unauthorized-accessible) | 1,434 | 12.9 |

**Table 6: Experimental result of insecure app identification.**

which are identified as BLE devices. The rest of them are likely Bluetooth classic. Surprisingly, among the 5, 822 BLE devices, 5, 509 (94.6%) can be fingerprinted with UUIDs, which means that our fingerprinting approach is quite effective to identify real IoT devices. To have a high level understanding of the popular IoT device manufactures, we extract those standard UUIDs and search their company name from the Bluetooth SIG [3]. Table 9 shows the company distribution of the UUIDs from our field test. According to the table, Google's IoT devices are the most prevalent ones in our scanned region (which contains a number of apartment complex), far more popular than those from Tile, Logitech, Nest Labs, etc.

**Device fingerprinting result.** With these 5, 509 fingerprintable devices, we further looked into top 10 specific fingerprinted devices. This result is presented in Table 10, along with the company name of the standard UUIDs from the Bluetooth SIG, and the fingerprinted apps as well as the number of their installation.

According to Table 10, 6 UUIDs are perfectly mapped to only one app, while the remaining 5 UUIDs are mapped to multiple apps. We

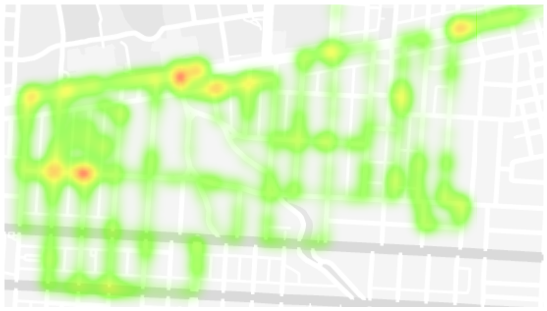


**Figure 5: Part of the geolocation of the scanned BLE devices in our field study.**

| Item | Value | % |
|---|---|---|
| # Unique Bluetooth Device | 30,862 | |
| # Unique BLE Device | 5,822 | 18.9 |
| # Fingerprintable BLE Device | 5,509 | 94.6 |
| # Vulnerable Device | 431 | 7.4 |
| # Sniffable Device | 369 | 6.7 |
| # Unauthorized Accessible Device | 342 | 6.2 |

**Table 8: Experimental result of our field test.**

| Company | # Devices |
|---|---|
| Google | 2,595 |
| Tile, Inc. | 441 |
| Logitech International SA | 131 |
| Nest Labs Inc | 114 |
| Hewlett-Packard Company | 74 |
| LG Electronics | 32 |
| Sonos, Inc. | 30 |
| Amazon.com Services, Inc. | 15 |
| Google Inc. | 10 |
| Anhui Huami Information Technology Co., Ltd. | 8 |
| Tencent Holdings Limited. | 8 |
| August Home Inc | 7 |
| Zebra Technologies | 5 |
| CSR | 5 |
| Apple, Inc. | 5 |
| UTC Fire and Security | 3 |
| Molekule, Inc. | 3 |
| Microsoft Corporation | 3 |
| Polar Electro Oy | 2 |
| GN ReSound A/S | 2 |
| B&O Play A/S | 2 |
| GoPro, Inc. | 2 |
| Dialog Semiconductor GmbH | 1 |
| RF Digital Corp | 1 |
| Clover Network, Inc | 1 |
| Facebook, Inc. | 1 |
| Gimbal, Inc. | 1 |
| Dexcom Inc | 1 |
| Snapchat Inc | 1 |
| Microsoft | 1 |
| Aterica Health Inc. | 1 |

**Table 9: Company names that can be found from Bluetooth SIG and corresponding devices in our field test.**

manually investigated apps that are mapped to the same UUID, and confirmed the reasons of UUID collision discussed in §6.2. For the two apps developed by HP company (row 8), they actually manage the same IoT device (a printer). In row 9 and row 10, though the UUIDs are both mapped to two apps respectively, these apps from various vendors control the same category of devices (fitness devices at row 9, and electric meters at row 10). For the UUID collision in row 4, it is possibly due to accidental reuse. Since we can only use the advertised service UUID for coarse-grained fingerprinting, the precision can be improved if we are able to connect to the device to obtain the complete UUID hierarchy.

**Vulnerable device identification.** Among the 5, 509 fingerprintable device, we have identified 431 (7.4%) of them that are vulnerable to either sniffing or unauthorized access. Among them, there are 369 (6.7%) snifferable devices and 342 (6.2%) unauthorized accessible devices, and 280 devices are vulnerable to both of the attacks. Moreover, we also count the top 10 vulnerable devices and their descriptions (found in the fingerprinted companion app) in Table 11. The most popular vulnerable devices include digital thermometer, car dongle, key finder, smart lamp, etc.

**Device distribution across location.** In the field test, we also recorded the GPS location where a device is scanned by us at the first time, such location can be used to infer the proximate location of a device. The field test was conducted at a roughly 1.28 square miles area. To illustrate the popularity and the usage intensity of the IoT devices, we draw a heat map showing the identified BLE IoT devices in a part of the area we tested in Figure 5. The areas with green color indicate the presence of IoT devices. The red color indicates where the BLE IoT devices are more intense, e.g., intersections because of the open area and the residential area.

## 7 COUNTERMEASURE

In this work, we have discovered two kinds of BLE vulnerabilities, which are rooted from the system level (BLE UUID fingerprinting from mobile apps) and the application level (no or weak authentication) implementations with the BLE devices and mobile apps. In the following, we discuss how to eliminate or mitigate these vulnerabilities.

**Mitigating app-level vulnerability.** The app-level vulnerabilities include absent cryptographic usage and flawed authentication, which are caused by the careless developers who do not implement encryption and hardcode the credentials in the app. To get rid of such vulnerabilities, the app should implement secure cryptographic function to encrypt the data to be sent. A secure cryptographic function also means that all the factors involved in the encryption should not be hardcoded. Besides, to eliminate flawed authentication, developers should hide the authentication credentials in the cloud or let users enter them in the app.

**Anti-UUID fingerprinting.** The root cause of our UUID fingerprinting is that BLE devices need to broadcast advertised packets to inform nearby apps. The UUID can be sniffed either from the advertisement packets or by browsing for services after connection is established. In addition, UUIDs are fixed values and do not change over time. Therefore, to anti-UUID fingerprinting, we can prevent them from being sniffed in the air or reverse-engineered

| UUID | Company Name | # Devices | APP Package Names | # Installed |
|---|---|---|---|---|
| 0000fe9f-0000-1000-8000-00805f9b34fb | Google | 2,436 | com.google.android.gms | 5,000,000,000 |
| 0000feed-0000-1000-8000-00805f9b34fb | Tile, Inc. | 441 | com.thetileapp.tile | 1,000,000 |
| 0000b13d-0000-1000-8000-00805f9b34fb | - | 243 | co.bird.android | 1,000,000 |
| adabfb00-6e7d-4601-bda2-bffaa68956ba | - | 208 | com.fitbit.FitbitMobile, de.afischer.aftrack.plugin.sensbox | 10,000,000 |
| 0000fe61-0000-1000-8000-00805f9b34fb | Logitech International SA | 131 | com.logitech.vc.parsec, com.logi.brownie | 5,000 |
| 0000feaf-0000-1000-8000-00805f9b34fb | Nest Labs Inc. | 114 | com.nest.android | 1,000,000 |
| 0000fea0-0000-1000-8000-00805f9b34fb | Google | 92 | com.google.android.apps.chromecast.app | 100,000,000 |
| 0000fe78-0000-1000-8000-00805f9b34fb | Hewlett-Packard Company | 74 | hp.enterprise.print, com.hp.printercontrol | 10,000,000 |
| fb694b90-f49e-4597-8306-171bba78f846 | - | 46 | com.lf.lfvtandroid, com.paofit.RideSocial, com.paofit.runsocial | 500,000 |
| 730a0ce2-9042-4ef1-870d-debe79a601f3 | - | 44 | com.powerley.aepohio, com.dteenergy.insight | 100,000 |
| d2d3f8ef-9c99-4d9c-a2b3-91c85d44326c | - | 44 | com.nest.android | 1,000,000 |

**Table 10: Top 10 devices found in the field test.**

| UUID | # Device | Device Description |
|---|---|---|
| 00001910-0000-1000-8000-00805f9b34fb | 7 | Digital Thermometer |
| 00001814-0000-1000-8000-00805f9b34fb | 6 | Car Dongle |
| 00001804-0000-1000-8000-00805f9b34fb | 6 | Key Finder |
| 0000fef1-0000-1000-8000-00805f9b34fb | 5 | Smart Lamp |
| 0000f000-0000-1000-8000-00805f9b34fb | 5 | Key Finder |
| 00001820-0000-1000-8000-00805f9b34fb | 4 | Smart Toy |
| bc2f4cc6-aaef-4351-9034-d66268e328f0 | 4 | Smart VFD |
| 0000ffd0-0000-1000-8000-00805f9b34fb | 4 | Air Condition Sensor |
| 000018f0-0000-1000-8000-00805f9b34fb | 4 | Smart Toy |
| 0000ec00-0000-1000-8000-00805f9b34fb | 4 | Accessibility Device |

**Table 11: Top 10 vulnerable devices found in the field test.**

in the app. In particular, we notice there could be solutions from three dimensions: (1) App-level (§7.1), (2) Channel-level (§7.2), and (3) Protocol-level (using dynamic UUIDs) (§7.3). In the rest of this section, we describe in detail how these defenses could be designed and implemented.

## 7.1 App-Level Protection

Since our fingerprinting attack relies on mobile app analysis to reveal the UUIDs and their hierarchies, accordingly any attempts to defeat or slow down the reverse engineering of mobile apps will be helpful. In general, it requires app developers to make effort to prevent the UUIDs and their hierarchies from being reverse engineered, in order to disable attackers for binding UUIDs to specific apps to fingerprint BLE IoT devices.

There are multiple ways to implement this type of protection. To begin with, as one of the reasons that UUIDs and their hierarchies can be obtained from the app is that they are hardcoded in plaintext, intuitively app developers can obfuscate the app to encode the UUIDs or use encryption to hide the UUIDs. Similar strategy had been implemented in [35]. Also, app developers can preserve the UUIDs in a cloud server, which is not accessible to the attackers. In this way, whenever a mobile app tries to connect to a nearby desired BLE device, it will dynamically retrieve the UUID from the cloud, so that attackers cannot obtain the UUIDs from statically reverse engineering of the mobile apps.

Although the protection methods in the app-level are seemingly plausible, these methods cannot fundamentally prevent the UUIDs from being reverse engineered from the mobile apps. This is because the obfuscation and encryption can only increase the difficulty for attackers to retrieve the UUIDs due to the fact that the plaintexted UUID should be interpreted in somewhere of the app. While storing UUIDs outside the mobile apps can prevent the UUIDs from being statically reverse engineered, the attackers can still obtain the plaintexted UUIDs at run-time, because the UUIDs are static and somewhat public in this scheme.

## 7.2 Channel-Level Protection

The second reason of why our BLE IoT devices fingerprinting can succeed is the recognizable UUIDs from the broadcasting packages of these devices. Therefore, any attempts at network channels to prevent an adversary from receiving complete signals of UUIDs would work.

The mitigation in the channel-level to disallow attacks to receive sufficient signals and packages for information recognition can be implemented with methods that disrupt signals broadcast from BLE IoT devices. In this way, attackers can only sniff either disrupted or interrupted instead of continuous signals, which is supposed to avoid the complete recognition of UUIDs. This type of mitigation has been implemented in BLE-Guardian [20] that depends on an additional hardware to broadcast disrupting signals to prevent packages sniffing. While this is an promising approach, it requires additional hardware support.

## 7.3 Protocol-Level Protection

Since the static UUID that can be extracted from the companion mobile apps is the root cause of our fingerprinting attack, a fundamental countermeasure would be to construct a one-time dynamic UUIDs for broadcast and communication. It may appear to modify the device hardware, but it turns out this countermeasure only require software update. That is, by only updating both the apps and the device firmware, we can achieve a dynamic UUID scheme.

In the following, we present how a one-time dynamic UUID can be generated and synchronized between the device and app. Since one device can be used by multiple users, we also put cloud in the scheme to help synchronize the UUIDs among users. At a high level, after an app is successfully connected with an IoT device for the first time, it negotiates a dynamic UUID for future broadcast and communication. As illustrated in Figure 6, this scheme can be broken down into three steps: (1) Dynamic UUID generation, (2) App-Device synchronization, and (3) Cloud synchronization, which are detailed as follows.

**(1) Dynamic UUID generation.** When the app connects with the device for the first time, it uses the *default UUIDs* to recognize the target device as usual. The default UUIDs are needed since they are necessary for broadcast and communication before negotiation of dynamic UUIDs. Though these default UUIDs enable an attacker to leverage them to fingerprint the device, the time window for the attack is quite narrow. After the first connection is established (Step 3), a dynamic UUID will be generated (Step 4). Since there are nearly $2^{128}$ possible UUIDs in theory, a random function would be able to generate a sufficient random UUID (impossible to be brute forced). The generation can be deployed on mobile app, device firmware, or even cloud server. In our design, we choose to generate the dynamic UUIDs on the app side, since it is more cost-effective (generation on firmware or cloud can bring additional energy or network consumption).

**(2) App-Device synchronization.** When the dynamic UUIDs are successfully generated, the next step is to synchronize them on both the app and the device so that they can agree on using them for future broadcast and communication. To achieve this, the app sends the dynamic UUIDs to the device (Step 5). When the device receives the dynamic UUID, it responses back to the app to inform the success of synchronization, and starts to use dynamic UUIDs for broadcast and communication instead of the default UUIDs fingerprintable by attackers (Step 6). Similarly, if the dynamic UUIDs are generated on the device or cloud, the synchronization can be implemented correspondingly.

**(3) Cloud synchronization.** In typical IoT scenarios, a BLE device usually needs to be connected with multiple companion apps, since an IoT device (e.g., a smart home device) can have several legitimate users. Therefore, we introduce a cloud synchronization step to save the dynamic generated UUIDs to the cloud (Step 7). Multiple apps can share a BLE device by using the same dynamic UUIDs on the cloud. As a result, in order to establish a connection after the dynamic UUIDs have been generated, an app will first synchronize the dynamic UUIDs and leverage them to establish a connection (Step 8-11). Moreover, cloud synchronization also prevents regenerate dynamic UUIDs for each connection, which is vulnerable to fingerprinting attack as previously discussed.

**Implementation and Deployment.** To prove the above scheme is viable, we have implemented a prototype using a real BLE chip in a software development board with series number "nRF52-DK". Specifically, this chip provides programming interfaces to configure UUIDs for advertisement packets, services, characteristics, and descriptors. It should be quite common for chip manufactures to provide such an interface, since each specific IoT device vendor needs to configure the UUIDs during the firmware development. With less than 500 LOC in both the mobile app and device firmware, we implemented the above proposed scheme, and tested it works as expected. To deploy our scheme, it requires software updates at both mobile apps and device firmware. This is actually consistent with today's IoT software ecosystem in which both app and firmware can be upgraded.

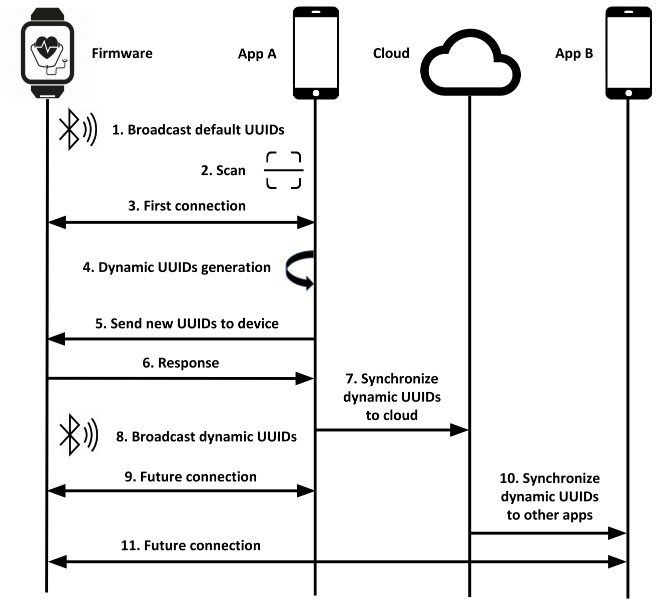

Firmware
App A
Cloud
App B
1. Broadcast default UUIDs
2. Scan
3. First connection
4. Dynamic UUIDs generation
5. Send new UUIDs to device
6. Response
7. Synchronize dynamic UUIDs to cloud
8. Broadcast dynamic UUIDs
9. Future connection
10. Synchronize dynamic UUIDs to other apps
11. Future connection

**Figure 6: A practical dynamic UUID defense without hardware modification.**

## 8 DISCUSSION

### 8.1 Limitation and Future Work

While BleScope has identified a great number of UUIDs and vulnerable IoT apps, there are still limitations that can be improved. First, when recognizing flawed authentication in mobile apps, BleScope applies a very strict rule which requires all the data to be sent should be hardcoded. As a matter of fact, this causes false negatives since we are not able to identify which data is the authentication credential. As a result, although some apps allow external input to be sent to the peripherals, it does not necessarily mean that the data is for authentication. For example, some network return data is only for firmware update, but unfortunately we cannot figure out in an automatic way to distinguish them. With a more sophisticated approach in the future, we may be able to precisely identify these vulnerable apps as well.

In addition, our backward slicing attempts to exhaustively explore all possible branches, which may cause branch explosion. If so, we will terminate our analysis for such apps, leading to another false negatives. One cause for the branch explosion is that Android apps are developed using object-oriented programming, with which the callers of some inherited object instances can only be determined at run-time. Therefore, our algorithm has to exhaustively search for all potential callers from its super classes in these cases, thereby causing branch explosion. One of our future works will attempt to address this problem.

Finally, in our scanning experiment we only obtained the public service UUIDs from the advertisement packets for fingerprinting. As a result, the fingerprint results we have obtained are not that precise, since in §5 we mention that hierarchy of UUIDs could precisely determine a device in the field test. Due to ethical reasons, it is impossible for us to reconstruct the hierarchy of UUIDs because it requires connection to devices. However, the attack is still possible for real attackers since they do not have ethical considerations.

### 8.2 Ethics Consideration

Since our experiment is conducted on real IoT devices which contain sensitive and private user information, we never exploit any of the vulnerabilities due to ethics consideration. Also, we never actively connect with any of the devices and only passively scanned the advertised packets to obtain UUIDs, which is public to all. As a result, as stated in §6.2, we only collected those service UUIDs exposed in the advertised BLE packets. Therefore, we never evaluated the hierahical UUIDs for the fingerprinting (only tested with our own BLE devices), since it requires connection to fetch the next layer UUIDs from the device.

**Responsible Disclosure.** We have reported the BLE IoT device fingerprinting with static UUIDs to the Bluthooth SIG. We have also reported the vulnerabilities to the developers of the vulnerable apps based on their contact information from Google Play.

## 9 RELATED WORK

**IoT security.** Recently, there were significant amount of efforts in IoT security. These efforts have uncovered numerous vulnerabilities of various IoT devices including BLE ones such as bands, watches, smart locks, and smart homes [14–18, 21, 23, 25, 36, 42]. Typical vulnerabilities are pairing credential leakage [16, 17], unchanged address [14, 18], unencrypted channel [42], information leakage [18], privilege misconfiguration [21, 23], and even memory corruptions [15]. These insecure devices expose attack surfaces to nearby attackers with various attacks such as man-in-the-middle (MITM), denial-of-service (DoS), location tracking, and even gain unauthorized access. Compared to these efforts, which usually focus on several devices of the same category with a relatively small scale, our research systematically studies all kinds of BLE IoT devices in the market from the perspective of companion mobile apps.

There are also defenses against the emerging security problems in the IoT area, such as SmartAuth [34], FlowFence [22], a gateway based system [19] and BLE-Guardian [20], which defend against MITM attacks, information leakage, and unauthorized accesses. BleScope is inspired by these works and we enrich them with more systematic countermeasures.

**BLE security.** While there are a great amount of effort on IoT security, only a handful of them focused on BLE security. In particular, Ryan et al. studied the insecure pairing protocol for the LE legacy connection in BLE 4.0 and 4.1, and demonstrated that it is vulnerable to eavesdropping [30], which made the Bluthooth SIG patch this vulnerability in later versions [1]. More recently, researches have targeted latest BLE generations to uncover vulnerabilities, such as the MITM attack on Passkey Entry pairing protocol [32, 33], brute-force attack to calculate long term key [41]. In our work, we study a unique and interesting problem: fingerprinting BLE devices with static UUIDs available in mobile apps.

**Vulnerability discovery based on mobile apps analysis.** In the past several years, there are tremendous efforts of uncovering vulnerabilities based on mobile app analysis by using techniques such as taint analysis. The vulnerabilities discovered could be just in the phone (e.g., privacy sensitive data such as address book leakage by Flowdroid [11] and Amandroid [37]), or in the server (e.g., vulnerable authentication [44] and authorization [45], or cloud data leakage [43], or missing security check [28], or lack of software updates [10]) to which the app communicates. BleScope complements with these works by identifying vulnerabilities in both BLE IoT devices and companion mobile apps.

## 10 CONCLUSION

We have presented BleScope, a tool to fingerprint BLE devices by using static UUIDs extracted from the companion mobile apps. As a side product, it also identifies the vulnerable apps that do not have any user authentication with the devices. We have tested BleScope with 18, 166 apps from Google Play, and it discovered 168, 093 UUIDs (13, 566 unique) and 1, 757 vulnerable apps among them. With the harvested UUIDs and vulnerable apps, we conducted a field test in a 1.28 square miles area, and BleScope discovered 5, 822 BLE devices and fingerprinted 5, 509 (94.6%) of them. Among the identified devices, 431 (7.4%) of them are vulnerable to attacks including sniffing and unauthorized access.

## ACKNOWLEDGMENT

We would like to thank the anonymous reviewers for their helpful feedbacks. This research was supported in part by NSF awards 1718084, 1750809, 1834215, and 1834216. Any opinions, findings, conclusions, or recommendations expressed are those of the authors and not necessarily of the NSF.

## REFERENCES

[1] 2014. BLUETOOTH SPECIFICATION Version 4.2. https://www.bluetooth.org/DocMan/handlers/DownloadDoc.ashx?doc_id=286439.
[2] 2017. Bluetooth Pairing Part 4: LE SecureConnections. https://www.bluetooth.com/blog/bluetooth-pairing-part-4/.
[3] 2019. 16 Bit UUIDs for Members. https://www.bluetooth.com/specifications/assigned-numbers/16-bit-uuids-for-members/.
[4] 2019. BLE Advertising Primer. https://www.argenox.com/library/bluetooth-low-energy/ble-advertising-primer/.
[5] 2019. BluetoothDevice | Android Developers. https://developer.android.com/reference/android/bluetooth/BluetoothDevice.
[6] 2019. BluetoothGatt | Android Developers. https://www.bluetooth.com/specifications/gatt/generic-attributes-overview/.
[7] 2019. GATT Overview | Bluetooth Technology Website. https://developer.android.com/reference/android/bluetooth/BluetoothGatt.
[8] 2019. Parani-UD100 Bluetooth 4.0 Class1 USB Adapter. http://www.senanetworks.com/ud100-g03.html.
[9] 2019. Soot - a framework for analyzing and transforming java and android applications. http://sable.github.io/soot/.
[10] Omar Alrawi, Chaoshun Zuo, Ruian Duan, Ranjita Kasturi, Zhiqiang Lin, and Brendan Saltaformaggio. 2019. The Betrayal At Cloud City: An Empirical Analysis Of Cloud-Based Mobile Backends. In *28th USENIX Security Symposium (USENIX Security 19)*.
[11] 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. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 259–269. https://doi.org/10.1145/2594291.2594299
[12] Gogul Balakrishnan and Thomas Reps. 2004. Analyzing memory accesses in x86 executables. In *International conference on compiler construction*. Springer, 5–23.
[13] BlueBorne. 2019. The Attack Vector "BlueBorne" Exposes Almost Every Connected Device. https://armis.com/blueborne/.
[14] Redjem Bouhenguel, Imad Mahgoub, and Mohammad Ilyas. 2008. Bluetooth security in wearable computing applications. In *2008 international symposium on high capacity optical networks and enabling technologies*. IEEE, 182–186.

[15] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. 2018. IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing.. In *NDSS*.
[16] Brian Cusack, Bryce Antony, Gerard Ward, and Shaunak Mody. 2017. Assessment of security vulnerabilities in wearable devices. (2017).
[17] Britt Cyr, Webb Horn, Daniela Miao, and Michael Specter. 2014. Security analysis of wearable fitness devices (fitbit). *Massachusets Institute of Technology* (2014), 1.
[18] Aveek K Das, Parth H Pathak, Chen-Nee Chuah, and Prasant Mohapatra. 2016. Uncovering privacy leakage in ble network traffic of wearable fitness trackers. In *Proceedings of the 17th International Workshop on Mobile Computing Systems and Applications*. ACM, 99–104.
[19] Charalampos Doukas, Ilias Maglogiannis, Vassiliki Koufi, Flora Malamateniou, and George Vassilacopoulos. 2012. Enabling data protection through PKI encryption in IoT m-Health devices. In *2012 IEEE 12th International Conference on Bioinformatics & Bioengineering (BIBE)*. IEEE, 25–29.
[20] Kassem Fawaz, Kyu-Han Kim, and Kang G Shin. 2016. Protecting Privacy of BLE Device Users. In *25th USENIX Security Symposium (USENIX Security 16)*. 1205–1221.
[21] Earlence Fernandes, Jaeyeon Jung, and Atul Prakash. 2016. Security analysis of emerging smart home applications. In *2016 IEEE symposium on security and privacy (SP)*. IEEE, 636–654.
[22] Earlence Fernandes, Justin Paupore, Amir Rahmati, Daniel Simionato, Mauro Conti, and Atul Prakash. 2016. Flowfence: Practical data protection for emerging iot application frameworks. In *25th USENIX Security Symposium (USENIX Security 16)*. 531–548.
[23] Grant Ho, Derek Leung, Pratyush Mishra, Ashkan Hosseini, Dawn Song, and David Wagner. 2016. Smart locks: Lessons for securing commodity internet of things devices. In *Proceedings of the 11th ACM on Asia conference on computer and communications security*. ACM, 461–472.
[24] Sławomir Jasek. 2016. Gattacking Bluetooth smart devices. In *Black Hat USA Conference*.
[25] Arun Cyril Jose and Reza Malekian. 2015. Smart home automation security. *SmartCR* 5, 4 (2015), 269–285.
[26] Raghavan Komondoor and Susan Horwitz. 2001. Using slicing to identify duplication in source code. In *International static analysis symposium*. Springer, 40–56.
[27] TAL MELAMED. 2018. An Active Man-in-the-middle Attack On Bluetooth Smart Devices. *Safety and Security Studies* (2018), 15.
[28] Abner Mendoza and Guofei Gu. 2018. Mobile Application Web API Reconnaissance: Web-to-Mobile Inconsistencies and Vulnerabilities. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (SP'18)*.
[29] William Oliff, Avgoustinos Filippoupolitis, and George Loukas. 2017. Evaluating the impact of malicious spoofing attacks on Bluetooth low energy based occupancy detection systems. In *Software Engineering Research, Management and Applications (SERA), 2017 IEEE 15th International Conference on*. IEEE, 379–385.
[30] Mike Ryan. 2013. Bluetooth: With Low Energy Comes Low Security. In *Proceedings of the 7th USENIX Conference on Offensive Technologies (WOOT'13)*. USENIX Association, Berkeley, CA, USA, 4–4. http://dl.acm.org/citation.cfm?id=2534748.2534754
[31] Pallavi Sivakumaran and Jorge Blasco. 2018. A Study of the Feasibility of Co-located App Attacks against BLE and a Large-Scale Analysis of the Current Application-Layer Security Landscape.
[32] Pallavi Sivakumaran and Jorge Blasco Alis. 2018. A Low Energy Profile: Analysing Characteristic Security on BLE Peripherals. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*. ACM, 152–154.
[33] Da-Zhi Sun, Yi Mu, and Willy Susilo. 2018. Man-in-the-middle attacks on Secure Simple Pairing in Bluetooth standard V5. 0 and its countermeasure. *Personal and Ubiquitous Computing* 22, 1 (2018), 55–67.
[34] 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)*. 361–378.
[35] Pei Wang, Qinkun Bao, Li Wang, Shuai Wang, Zhaofeng Chen, Tao Wei, and Dinghao Wu. 2018. Software protection on the go: A large-scale empirical study on mobile app obfuscation. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 26–36.
[36] Xueqiang Wang, Yuqiong Sun, Susanta Nanda, and XiaoFeng Wang. 2019. Looking from the Mirror: Evaluating IoT Device Security through Mobile Companion Apps. In *28th USENIX Security Symposium (USENIX Security 19)*. 1151–1167.
[37] Fengguo Wei, Sankardas Roy, Xinming Ou, et al. 2014. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1329–1341.
[38] Mark Weiser. 1981. Program slicing. In *Proceedings of the 5th international conference on Software engineering*. IEEE Press, 439–449.
[39] Tae-Hun Woo, Hwa-Ju Jo, Yong-Hwan Lee, and Sung-Young Kim. 2017. Infant Body Temperature Monitoring System using Temperature Change Detection Algorithm. In *Proceedings of the 2017 International Conference on Computer Science and Artificial Intelligence*. ACM, 270–274.
[40] Bin Yu, Lisheng Xu, and Yongxu Li. 2012. Bluetooth Low Energy (BLE) based mobile electrocardiogram monitoring system. In *2012 IEEE International Conference on Information and Automation*. IEEE, 763–767.
[41] Wondimu K Zegeye. 2015. Exploiting Bluetooth low energy pairing vulnerability in telemedicine. In *International Telemetering Conference Proceedings*. International Foundation for Telemetering.
[42] Qiaoyang Zhang and Zhiyao Liang. 2017. Security analysis of bluetooth low energy based smart wristbands. In *Frontiers of Sensors Technologies (ICFST), 2017 2nd International Conference on*. IEEE, 421–425.
[43] Chaoshun Zuo, Zhiqiang Lin, and Yinqian Zhang. 2019. Why Does Your Data Leak? Uncovering the Data Leakage in Cloud From Mobile Apps. In *Proceedings of the 2019 IEEE Symposium on Security and Privacy*. San Francisco, CA.
[44] Chaoshun Zuo, Wubing Wang, Rui Wang, and Zhiqiang Lin. 2016. Automatic Forgery of Cryptographically Consistent Messages to Identify Security Vulnerabilities in Mobile Services. In *Proceedings of the 23rd Annual Network and Distributed System Security Symposium (NDSS'16)*. San Diego, CA.
[45] Chaoshun Zuo, Qingchuan Zhao, and Zhiqiang Lin. 2017. AuthScope: Towards Automatic Discovery of Vulnerable Authorizations in Online Services. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS'17)*. Dallas, TX.