PeriScope: Comprehensive Vulnerability Analysis Of Mobile App-defined Bluetooth Peripherals

Qingchuan Zhao
cs.qczhao@cityu.edu.hk
City University of Hong Kong
Hong Kong SAR

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

Jorge Blasco
Jorge.BlascoAlis@rhul.ac.uk
Royal Holloway, University of London
Egham, England

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

ABSTRACT

Many IoT devices today talk to each other via Bluetooth Low Energy (BLE) which is a wireless communication technology often being used to exchange data between a paired central and peripheral. These peripheral devices include not only firmware-defined bare-metal Bluetooth peripherals but also software-defined peripherals (*e.g.*, *mobile app-defined Bluetooth peripherals* where a mobile app turns a smartphone into a peripheral instead of the central that a phone mostly serves). However, this role reversal increases the attack surface and brings vulnerabilities in bare-metal Bluetooth peripherals to mobile apps where relevant security and privacy have not been well studied. To take the first step towards filling this knowledge gap, this paper presents PeriScope, an automated tool to unveil the security and privacy vulnerabilities at the link layer of app-defined Bluetooth peripherals in the procedures of broadcasting, pairing, and communication by systematically analyzing their companion mobile apps. PeriScope has analyzed 1,160 Bluetooth peripheral apps from Google Play and identified 69.13% of them that broadcast device or personal identifiable information in cleartext, and, in addition, there are 95% pieces of data managed by these apps (*e.g.*, personal health data and digital keys to unlock doors) to exchange with connected devices can be accessed without authentication. Finally, a set of guidelines for secure app-defined Bluetooth peripherals development is provided.

CCS CONCEPTS

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

The main body of this work was finished when the first author was in his last year as a Ph.D. candidate at The Ohio State University.

ASIA CCS '22, May 30-June 3, 2022, Nagasaki, Japan

ACM ISBN 978-1-4503-9140-5/22/05…$15.00
https://doi.org/10.1145/3488932.3517410

KEYWORDS

Bluetooth low energy, mobile app analysis, IoT security

ACM Reference Format:
Qingchuan Zhao, Chaoshun Zuo, Jorge Blasco, and Zhiqiang Lin. 2022. PeriScope: Comprehensive Vulnerability Analysis Of Mobile App-defined Bluetooth Peripherals. In *Proceedings of the 2022 ACM Asia Conference on Computer and Communications Security (ASIA CCS '22), May 30-June 3, 2022, Nagasaki, Japan.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3488932.3517410

1 INTRODUCTION

Bluetooth Low Energy (*BLE*) is ubiquitous today, especially among small Internet-of-Things (*IoT*) devices (*e.g.*, Apple AirTag), because it allows a pair of devices to exchange data consuming extremely low energy. In particular, communications between two connected BLE devices are typically operated in a client-server mode, where one device called *central* acts as the client and the other device called *peripheral* functions as the server. When being disconnected, the peripheral constantly broadcasts advertising packets to declare its existence, and the central keeps scanning for advertising packets to discover nearby peripherals, and then establishes a connection with them if necessary (*e.g.*, smartphone requesting heart rate data from a fitness tracker).

App-defined Bluetooth Peripheral (*AdBP*). While Bluetooth peripherals are often referred to as firmware-defined bare-metal IoT devices (*e.g.*, AirTag), they can also be software-defined Bluetooth peripherals. We call these peripherals that are enabled via mobile apps with support from both the hardware (*i.e.*, the Bluetooth chip in the smartphone) and the operating system (no bare-metal anymore) as app-defined Bluetooth peripherals (*AdBP*). In particular, to ease the development effort, mobile operating systems (*e.g.*, Android) hide all low level details of BLE communications, such as the management of the operations at both BLE link layer and physical layer, and provide system APIs for developers to configure a smartphone as a BLE peripheral. Its development convenience as well as the rich sensors and the ubiquity of smartphones bring *AdBP* promising potentials in many scenarios. For instance, most recently, *AdBP* has been used for automated contact tracing to fight against COVID-19 pandemic in many countries.

Unfortunately, *AdBP* arises serious security and privacy concerns because of its enlarged attack surface which is discovered

in bare-metal Bluetooth peripherals. However, there only exist a few relevant studies that either focused on a particular problem (*e.g.*, MAC address based tracking [8]) or vulnerabilities in a specific application (*e.g.*, digital contact tracing [2]), and there lacks a unified perspective on the security and privacy in *AdBP* leaving this field largely unexploited. In particular, it is unknown that how many mobile apps are able to define Bluetooth peripherals, what are their security and privacy practices, and to which extent their consequent issues would impact normal users. For example, whether these peripherals would suffer similar vulnerabilities as those in bare-metal peripherals (*e.g.*, AirTag) including broadcasting device identifiable information (*i.e.*, static service UUIDs) [11, 53] and leaking sensitive data [15, 16, 50].

These aforementioned concerns are not hypothetical. In our preliminary study, we have identified an industry leading app, Lyft Driver, that places the driver ID in its advertising packets when it turns the smartphone into a BLE peripheral. This driver ID represents the identity of a driver and allows attackers to track the movements of a driver which may reveal fine-grained personal identifiable information. We also have discovered an account unlocking app that configures no security protection on a piece of its sensitive data, *i.e.*, the keys to unlock user accounts. Such a vulnerable configuration makes these credentials accessible to any connected centrals without authentication.

Objectives. To shed light on these aforementioned questions, this paper takes the first step towards unveiling the security and privacy in *AdBP*. To this end, this paper first develops a threat model including unique attack surfaces and adversarial objectives particular in the link layer of an *AdBP*, and then presents the design and implementation of an automated tool, PeriScope, to automatically uncover the vulnerabilities by systematically analyzing the companion apps. First, considering there is no specific category of the companion apps of *AdBP* in public markets, PeriScope takes a mobile app as input and scans its system API usages which endorse the presence of a unique and essential functionality of a Bluetooth peripheral (*i.e.*, broadcasting service UUID) to recognize an *AdBP*. Next, it combines inter-procedural static backward slicing alongside forward string value analysis to capture the generation procedures of a piece of data that is involved in an *AdBP* communication at the link layer, and then resolve its associated value to obtain its content, semantic, and applied security protections. Finally, PeriScope inspects both the semantic and content of a piece of data and its associated security protections to identify potential vulnerabilities according to the threat model. Moreover, after systematizing vulnerabilities in real-world applications at scale, this paper aims to propose a set of guidelines for secure development with the hope to prevent severe security and privacy impacts at the early stage.

Our Findings. PeriScope has recognized 1,160 *AdBP* companion apps after scanning *all* free apps in Google Play as of the end of September 2021 and identified vulnerabilities at the link layer in the procedures of broadcasting (*i.e.*, broadcasting device or personal identifiable information in cleartext) and communication (*i.e.*, sensitive data access without authentication). Specifically, in the broadcasting procedure, PeriScope has identified 799 *AdBP* companion apps that broadcast device identifiable information, *i.e.*, static UUID, which makes them subject to the device (and in this case app) fingerprinting attacks [11, 53], three apps that even use user identifiable information as their broadcast UUIDs, and 537 apps that also broadcast another potential of user identifiable information, *i.e.*, the user customized phone name (*e.g.*, `Alice's pixel`).

On the other hand, in the communication procedure, PeriScope has identified a total of 662 apps managing 1,752 pieces of data in exchange between a pair of connected devices. Specifically, 1,430 pieces of data are configured to allow read operations and 1,233 of them are set to accept write operations. However, these apps configure no protection on 95.10% of those data that allow reading and 94.81% of them that accept writing. After manual inspecting the readable data without protection, we have identified an improper access control that can leak a wide range of sensitive data, which include but not limited to personal health data, digital identifiers of users, and even digital keys to unlock doors.

In addition to the identified vulnerabilities, this study also presents an overview of the ecosystem of *AdBP*. As of this writing, *AdBP* has been in presence of 28 categories labelled by Google Play, and the leading categories include "Tools", "Lifestyle", and "Education", according to the number of apps in a category. In addition, based on the manufacturer ID which is a unique number assigned by the Bluetooth SIG, this ecosystem currently involves 61 manufacturers (at least) that contribute to 501 apps. To our surprise, among these apps, this study discovers 351 of them that violate the manufacturer ID usage policy defined by Bluetooth SIG. Specifically, there are 243 apps that place IDs belonging to other manufacturers in their advertising packets, 91 apps that use unassigned IDs, and 17 apps that even use the preserved ID for internal testing in shipping products, which is disallowed [32].

Contribution. Our study makes the following contributions:

- **Novel Problem.** This paper conducts the first comprehensive security analysis on an emerging category of mobile apps that configure smartphones to be Bluetooth peripherals to understand the current security and privacy practice and present a set of guidelines for secure development of mobile app-defined Bluetooth peripherals with the hope to prevent severe security impacts at the early stage.
- **New Tool.** We design and implement PeriScope to automatically demystify Bluetooth configurations in mobile app-defined Bluetooth peripherals by uncovering their link layer configurations via reverse engineering Bluetooth peripheral apps, and detect their vulnerabilities that are subject to passive eavesdropping and active man-in-the-middle attacks.
- **Empirical Evaluation.** PeriScope has recognized 1,160 Bluetooth peripheral companion apps, in which it has detected 69.13% apps that are subject to passive eavesdropping attacks in the broadcast procedure leaking device or personal identifiable information and around 95% of GATT attributes have been assigned with weak protections leading to active MITM attacks leaking sensitive information. In addition, it has also discovered a severe manufacturer ID abuse that many app-defined Bluetooth peripherals violate the policy defined by Bluetooth SIG of using unassigned manufacture IDs, manufacturer IDs belonging to others, and a preserved ID that is disallowed to appear in shipping products.

2 BACKGROUND

2.1 Bluetooth Low Energy

Bluetooth Low Enery (BLE), initially released within the Bluetooth 4.0 specification, is a special version of the Bluetooth technology. Compared to Bluetooth Classic, BLE maintains essential Bluetooth functionality with extremely lower power consumption (*e.g.*, up to 1% of power usages in Bluetooth Classic). To communicate with connected devices, BLE devices follow the Generic Access Profile (**GAP**) [41] and make use of the Generic Attribute Profile (**GATT**) to transfer data between them [29].

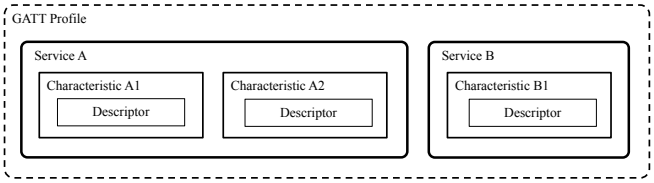

Figure 1: Illustration of GATT Architecture.

Generic Access Profile (GAP). The GAP defines two communication mechanisms for BLE devices: broadcasting and connecting [41]. Specifically, broadcasting is a one direction communication in which a BLE device only transfers information to other devices by constantly broadcasting advertising packets. In addition, a BLE device can also connect to another device to transfer data in two directions. In this mechanism, there are two roles, *i.e.*, *Peripheral* and *Central*. In particular, a peripheral device will constantly broadcast advertising packets to declare its existence if disconnected. The broadcasting will be terminated when it is connected with a central, and a central device will periodically scan nearby advertising packets to search for the specific peripheral. If wishing, it will initiate a connection request to the peripheral.

GATT. The GATT specifies how a piece of data is stored in a local device and accessed by a remote device. In particular, the GATT has three attributes: *service*, *characteristic*, and *descriptor*, and they are organized hierarchically (illustrated in Figure 1) where a service can contain several characteristics and each characteristic can consist of several descriptors. Additionally, each attribute is addressable with a fixed-length universally unique identifier (**UUID**), and a piece of data is stored in a characteristic or a descriptor, both of which are specified with security **permissions** to enforce the access control.

The *UUID* is a 128-bit number that makes each GATT attribute addressable. There are two types of UUIDs: the standard and the custom UUIDs. Standard ones are specified by the Bluetooth SIG for dedicated services and custom ones are created by developers for their own usages. Particularly, a standard UUID is specified in short (16-bit or 32-bit), and its full length (128-bit) is reconstructed by concatenating a fixed base (`0000-1000-8000-00805F9B34FB`) to its short version. Additionally, custom UUIDs are all 128-bit in length and they cannot share the same value with any standard UUID.

GATT *permissions* specify the security requirement for a remote device to access a piece of data that is stored in a GATT attribute. In total, GATT defines three permissions [29]: (*a*) *Access* permission that specifies whether the data stored in an attribute is readable, writable, neither or both, to a remote device; (*b*) *Encryption* permission that enforces the data of an attribute to be accessed or transferred to a remote device through an encrypted channel; and (*c*) *Authentication* permission that protects the data from being accessed by unauthenticated devices.

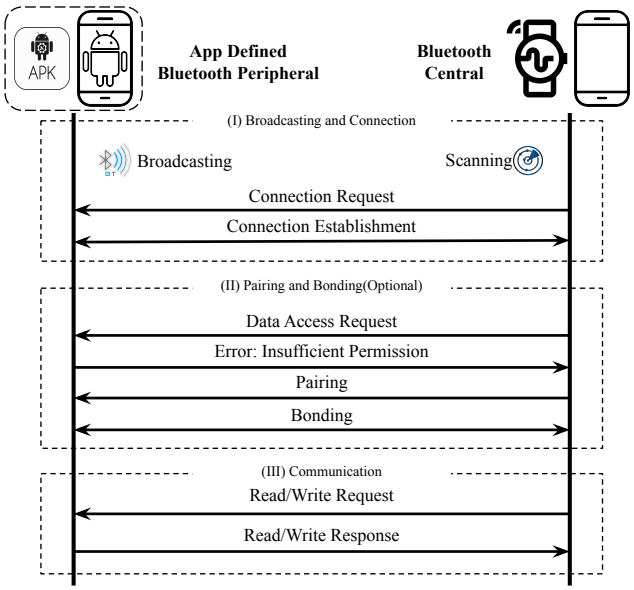

Figure 2: Workflow of Mobile App Defined Bluetooth Peripherals.

Moreover, the data transmission is between a *GATT server* and a *GATT client*. The server manages GATT attributes and responses to remote requests sent by a client. It is worth noting that GAP roles (*i.e.*, central and peripheral) are essentially independent to GATT roles (*i.e.*, server and client).

2.2 App-defined Bluetooth Peripherals

An app-defined Bluetooth peripheral, *AdBP*, is a BLE-enabled smartphone that is configured by a mobile app to function as a peripheral. Specifically, it acts as a normal BLE peripheral. In Android, defining a peripheral requires the app has to use the system APIs to implement its application logic and configure the security requirements for a remote device to access to its data.

Defining An *AdBP*. To support the above two working modes, the operating system provides an app with two types of configurations : (*a*) the configuration of broadcasting behaviors (for beacons) and (*b*) the configuration of the GATT server (for connectable peripherals).

(A) Broadcasting Configuration.. It is an essential functionality for a BLE peripheral to broadcast advertising packets. These packets in an *AdBP* can be configured by its companion Android app invoking relevant system APIs including the properties of emitting packets and custom data in these packets.

In particular, Android apps can configure three *broadcasting properties*: mode, timeout, and the transmission power. Specifically, the mode determines the time interval between two adjacent advertising packets, the timeout limits the duration of each broadcasting session, and the transmission power (*TxPower*) controls the power

to emit packets. On the other hand, Android apps can also customize several data fields in the advertising packets. First, apps can customize the fields for specific services including the service UUID, service data, the manufacturer ID, and manufacturer specific data. Second, apps can also choose to place additional information about the peripheral in the packets, *e.g.*, device name. Third, apps also can declare its connectivity in the packets.

(B) GATT Server Configuration.. A GATT server is required to be maintained by an app-defined Bluetooth peripheral if it supports the connectable mode since it manages the data transmissions. In particular, only attributes that have been registered to a GATT server can be accessed by remote devices. To work properly, all attributes should be well configured and Android provides different configuration options for each attribute based on its unique nature:

- **Service**: An Android app can configure a service as either a primary or a secondary service. A primary service represents a main functionality such as the *Blood Pressure* service which is defined by the SIG [39], while a secondary service should be included within a primary service to provide complementary functionality [1].
- **Characteristic:** A characteristic contains a piece of information of a service. For example, *Blood Pressure Measurement* is one characteristic of the Blood Pressure service [39]. An Android app can specify both the property and the permission of a characteristic. Specifically, the property defines actions that can be performed by remote devices and the permission declares security requirements for the data access [33].
- **Descriptor:** A descriptor is contained within a characteristic to provide additional information [23]. Unlike characteristics, descriptors only have permissions allowing configuration.

```
public User(...) {
    ...
    String userId;
    ...
}
```

```
public String getUserUuid(String arg3) {
    . . .
    String uBase =
                    "00000123-0123-0123-0123-00000"
    return uBase + arg3;
}
```

```
1 private void bleAdvFunc(DriverRide arg2) {
2     . . .
3     String userId = this.a(arg2.getUser().getUserId());
4     String uuid = getUserUuid(userId);
4     ParcelUuid pUuid = new ParcelUuid(UUID.fromString(uuid))
5     AdvertiseSettings$Builder v1 = new AdvertiseSettings$Builder()
                                       .setAdvertiseMode(0)
                                      .setConnectable(true)
                                       .setTxPowerLevel(3).build();

6     AdvertiseData$Builder v2 = new AdvertiseData$Builder()
                                 .addServiceUuid(pUuid).build();
7     bleAdvertiser.startAdvertising(v1,v2,((AdvertiseCallback)v3));
8     . . .
```

Figure 3: PII Broadcast in Cleartext by Lyft Driver.

The Workflow of *AdBP*. Figure 2 illustrates the workflow of an *AdBP*, which contains three procedures:

- **(I) Broadcasting and Connection**: At the start, an *AdBP* will constantly broadcast advertising packets, which contain device identifiable information (*e.g.*, service UUID and device name). Meanwhile, a central device keeps scanning nearby packets and extracts such identifiable information from these packets to discover the specific peripheral of interest. Next, if the peripheral allows the connection, then the central may initiate a connection request. The connection will be established when such a request is accepted.
- **(II) Pairing and Bonding**: The paring and bonding procedures are optional to create a secure channel for two devices to transfer data [29]. Specifically, these procedures will be initiated by a central in response to a data access request error due to insufficient authentication (due to the peripheral configuration). The pairing procedure starts by exchanging security features between the two devices to negotiate the pairing protocol, *i.e.*, "Just Works", "Numeric Comparison", "Passkey Entry", and "Out of Band (OOB)" [31]. After the exchange, both devices generate and synchronize a temporary key to encrypt the communication channel [7]. After the pairing, the central can start the bonding procedure to let a paired devices remember each other for subsequent connections. To this end, both devices will negotiate a long term key (*LTK*) through the channel encrypted by the temporary key and store such a LTK locally to finish the bonding.
- **(III) Communication**: After the connection is established, even without pairing and bonding, the central can try to access a piece of data stored in the peripheral. Since such a piece of data is stored in a GATT attribute (*e.g.*, characteristic) on which the GATT server enforces access control. As such, only access from qualified central would be accepted. Otherwise, a corresponding error message will be responded. Having the error message, the central may need to initiate the pairing and bonding procedures to satisfy the declared security requirements for access (step II).

3 OVERVIEW

In this section we present an overview of PeriScope. We motivate our design with two real world examples (§3.1).Based on these examples, we develop the threat model (§3.2) so we can identify the attack surface of *AdBP* and possible adversary objectives. Finally, defines the scope (§3.3) of this study.

3.1 Motivating Examples

As presented in §2.2, an app can customize the data in advertising packets, which usually stores data in cleartext, and configure the permissions on characteristics and descriptors to enforce access control. Unfortunately, these two capabilities can both result in vulnerabilities. In the following, we use two real world examples to illustrate their presence.

PII in Advertising Packets. An *AdBP* will constantly broadcast advertising packets if it is disconnected. These packets store data in cleartext by design. Unfortunately, there are several data fields in advertising packets that can be customized by developers, and they may mistakenly place sensitive data in any of these fields. In the following, we use an industry leading app, Lyft Driver, to illustrate how it mistakenly broadcasts the personal identifiable information (*PII*).

In particular, as shown in Figure 3, we notice this app adds the `pUuid` into its advertising packets at line 6. This uuid is generated from method `getUserUuid` at line 3, which takes the `userId` as input to produce the uuid by concatenating such an input value to a fixed string `00000123-0123-0123-0123-00000`. Next, by tracing the input value of `userId`, we can understand that it is a unique ID

for each driver. As such, the service UUID in advertising packets is actually the user ID that can be used for user identity recognition and, even worse, location tracking.

Credentials Leakage From GATT Server. An *AdBP* that supports the mode of connectable peripheral has to maintain a GATT server which enforces access control on GATT attributes to ensure the secure data transmission. Consequently, if such a security mechanism is vulnerably implemented on the sensitive and private data, it can lead to severe privacy leakages. In the following, we use a popular account unlocking app, which allows users to automatically unlock the account to log into a desktop or laptop using a mobile phone, as an example to demonstrate how it exposes user credentials to attackers without protection in Figure 4.

Specifically, we can observe that this app declares itself as a connectable device at line 2, and invokes the method `prepareAnd AddService` at line 5 to configure the GATT server right after broadcasting advertising packets. In the invoked method, this app reads a device list at line 9, and then iterates such a list to create a characteristic for each device at line 14. When creating a characteristic, the app specifies the `PERMISSION_READ` permission and stores the decrypted key that can unlock the account in such a device at line 16. Since the `PERMISSION_READ` has no security requirement, each central connecting to such a peripheral can read the list of keys used to unlock accounts of the user. Therefore, the improper access control in this peripheral can leak sensitive data, *i.e.*, user credentials.

```
1 public void startAdvertise() {
2     AdvertiseSettings v2 = new AdvertiseSettings$Builder()
                                          .setAdvertiseMode(2)
                                          .setTxPowerLevel(3)
                                          .setConnectable(true).build();
3     UUID v4 = UUID.fromString(this.getAdvertiseUUID());
4     AdvertiseData v0 = new AdvertiseData$Builder()
                               .addServiceUuid(new ParcelUuid(v4)).build();
      . . .
5     this.prepareAndAddService(v4);
      . . .
6 }
```

```
7 private void prepareAndAddService(UUID arg9) {
8     this.mGattService = new BluetoothGattService(arg9, 0);
9     List v2 = this.getDevicesList();
10    if(v2 != null) {
11        Iterator v4 = v2.iterator();
12        while(v4.hasNext()) {
13            Object v1 = v4.next();
14            BluetoothGattCharacteristic v0 =
                                          new BluetoothGattCharacteristic(
                                              (Device)v1).getUuid()), 2, 1);
15            if(((Device)v1).getKey() != null) {
16                v0.setValue(this.decryptDeviceKey(((Device)v1).getKey()));
17            }
18            this.mGattService.addCharacteristic(v0);
19        }
20    }
21    this.mGattServer.addService(this.mGattService);
22 }
```

Figure 4: Credentials Leakage in an Account Locking App.

3.2 Threat Model

Attack Surface. According to the aforementioned real-world examples and the nature of an *AdBP*, an attack can succeed during procedures (I) Broadcasting and Connections, or (III) Communication. In the latter case, the attack must go through all workflow procedures of an *AdBP*. Therefore, this study defines the attack surface of an *AdBP* in respect of its workflow.

(I) Broadcasting and Connections. When being disconnected, an *AdBP* would broadcast advertising packets to declare its existence, and the data carried in these packets are in cleartext by design. Therefore, if developers mistakenly place sensitive data in such packets, a nearby attacker can sniff and interpret such sensitive information, and conduct consequent attacks. Additionally, an attacker can also establish connection to the victim device.

(II) Pairing and Bonding. In this procedure, attackers launch the downgrade attack [51] that forces a pair of devices to use insecure pairing and bonding protocol (*i.e.*, "Just Works"). In this way, attacker can bypass the *encryption protection* assigned to a GATT attribute and access its stored data in the next procedure.

(III) Communication. Since the connection procedure can be done without victim involvement and the procedure of pairing and bonding is triggered when there is an request without sufficient permission, an attacker can access a piece of data stored in an unprotected or ill-protected GATT attribute without victim's notice. Specifically, if developers place inappropriate permission on sensitive data stored in a GATT attribute, an attacker secretly connect to a victim's device and steal that sensitive data to finish the attack.

Adversarial Capabilities and Objectives. In our threat model, an attacker is equipped with BLE-enabled devices that can passively sniff BLE packets and actively establish connections with BLE peripherals. Additionally, the attacker is able to reverse engineer mobile apps to retrieve the UUIDs that are placed in advertising packets or used to label GATT attributes. Also, the attacker is capable of uncovering the permissions assigned on each GATT characteristic and descriptor. On the other hand, the primary objective of an attacker is to steal sensitive data either from advertising packets or GATT attributes, and then conduct specific attacks based on the type of sensitive data, such as using PII for movement tracking.

3.3 Scope

We focus on unveiling security and privacy practices in *AdBP* that are defined during Bluetooth peripheral creation (*i.e.*, configuration of broadcasting and GATT server). While there are other types of vulnerabilities of Bluetooth peripherals in the link or physical layer (*e.g.*, eavesdropping [22, 45]), since related configurations in these two layers are hidden to the apps and handled by the mobile operating system which is assumed to not be compromised, such vulnerabilities are excluded from this work. Additionally, while the proposed approach scales to apps working in other mobile operating system, we focus on Android *AdBP* as the first step towards understanding security and privacy in this emerging category of devices. In particular, we focus on the peripheral behaviors that are defined by an app using system APIs at the Java bytecode level, and such behaviors that are implemented with custom APIs or other techniques (*e.g.*, native libraries and JavaScript in WebView) are out of the scope. Moreover, the analysis is based on the BLE 4.*x* specifications and above.

3.4 PeriScope Overview

Based on the presented motivating examples (§3.1), threat model (§3.2), and the scope (§3.3), PeriScope needs to: (I) detect the companion app of an *AdBP*, (II) uncover the components in the advertising packet and (III) reveal the configuration of the GATT server of

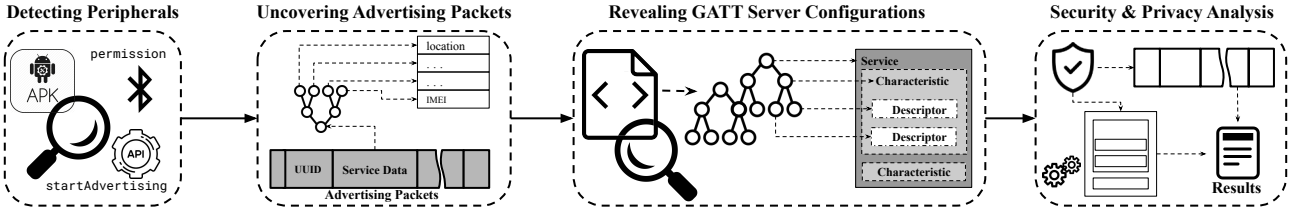

Figure 5: The Workflow of PeriScope.

this *AdBP*, and (IV) finally identify vulnerable security and privacy practices. Specifically,

Precise *AdBP* Companion App Recognition: Since there does not exist a central public repository of these specific category of apps and there is also no explicit label assigned to them by public app markets (*e.g.*, Google Play), we perform the first attempt to recognize *AdBP* companion apps in the literature, PeriScope has to propose a new solution to precisely recognize them.

Accurate Programming Value Resolution: After recognizing a companion app of an *AdBP*, PeriScope next has to uncover the components of its advertising packet and reveal its GATT server configuration. While these two tasks can be accomplished by resolving the input values to corresponding system APIs, these values may not always be defined statically (as shown in Figure 3. As such, it demands for an algorithm to resolve such values generated through complicated procedures.

Effective Vulnerability Identification: PeriScope finally has to identify vulnerabilities from the results obtained in previous tasks. Unfortunately, given the different nature of components in an advertising packet and attributes in a GATT server, it is unrealistic to use a unified rule to identify vulnerabilities in all of them. Therefore, PeriScope has to design different strategies to identify vulnerabilities specific to each of them.

4 DESIGN OF PERISCOPE

This section presents the detailed design of PeriScope to accomplish the above listed tasks. As shown in Figure 5, PeriScope first leverages a unique invocation feature of system APIs to recognize *AdBP* companion mobile apps (§4.1). Next, it performs an interprocedural program analysis to resolve values an app used to define an *AdBP* to uncover its advertising packets (§4.2) and reveal GATT server configuration (§4.3). Finally, PeriScope designs a series of security rules to identify vulnerabilities from different attack vectors in advertising packets and GATT attributes (§4.4).

4.1 Recognizing An *AdBP* Companion App

PeriScope starts from detecting whether an app is capable of turning a phone into a Bluetooth peripheral. Considering it requires invoking relevant system APIs, and an app should have the permissions to use them, PeriScope first checks the permissions requested by an app and then examines the presence of relevant system APIs.

Checking Bluetooth Permissions. An app of *AdBP* should request for the Bluetooth related permissions, *e.g.*, `BLUETOOTH` and `BLUETOOTH_ADMIN`. Since such requests have to be declared by an app in its *Manifest* file, its capability of using Bluetooth can be understood by inspecting such a file. As such, PeriScope uses `apktool` [4], a tool to reverse engineer APK files, to decompile an app to extract its *Manifest* file and check the existence of the required permissions.

Detecting Peripheral Functionality. Previous step excludes apps that cannot use Bluetooth, which is insufficient to recognize *AdBP* companion apps because an app may use Bluetooth to define a central device. As such, this step further excludes apps that cannot perform the unique and essential functionality of a Bluetooth peripheral, *i.e.*, broadcasting advertising packets. To broadcast, an app has to invoke corresponding APIs. While such APIs might be custom APIs, third-party APIs, and system APIs, without loss of generality in this large scale study, PeriScope only focuses on system APIs defined by Android (*e.g.*, `startAdvertising`). Accordingly, PeriScope depends on `dex2jar` [18] to detect the presence of these APIs to find *AdBP* companion apps

4.2 Uncovering Advertising Packets

Having detected whether an app is capable of defining Bluetooth peripherals, the next task for PeriScope is to uncover the custom data that have been placed in advertising packets. While it seems plausible to uncover such data by dynamically running the app and sniffing its advertising packets in the air, the unique requirement for each specific app (*e.g.*, user registration and navigation to a specific interface) to trigger such functionality makes it hard to scale. For the sake of scalability, PeriScope depends on static analysis to uncover the custom data in advertising packets by resolving the input value to the dedicated system APIs that are used to customize corresponding data fields.

Target System APIs. Android provides a variety of system APIs to customize data carried in advertising packets. Specifically, APIs that customize service UUID, service data, manufacturer ID, manufacturer data, and device name include `addServiceData`, `addServiceUuid`, `addManufacturerData`, and `setIncludeDeviceName` (the full list is shown in Table 6 in Appendix A).

Resolving Customized Data. PeriScope uncovers the customized data in advertising packets by resolving the input value to aforementioned system APIs. Unfortunately, not all input values could be resolved in the way that is as straightforward as the value of advertisement mode in the example of Hideez Lock (shown in line 2 in Figure 4) which can be resolved directly at place of usage. These values also could be generated in a similar way as the service UUID generation processes in Lyft Driver (§3.1) which has gone through a series of computations. Therefore, PeriScope has to capture all

computations associated to an input value and then repeat these procedures to resolve its value.

Specifically, PeriScope first leverages FlowDroid [6] and EdgeMiner [10] to construct the inter-procedural control-flow graph (ICFG) that contains both explicit (*i.e.*, direct-call) and implicit (*i.e.*, callbacks of Android components) edges. Upon the ICFG, its then builds an inter-procedural data-flow graph (IDDG) where each node represents an instruction and each edge is a control-flow transfer. Next, by backward traversing the IDDG, PeriScope traces every input to target APIs from where it is used to where it is initialized, and records associated computations of each value. According to the recorded procedures, for each input value, PeriScope performs the same computations in the same order (in reverse to the recording order) to resolve each input value. While it can resolve concrete values of those static one, such as the static service UUID, manufacturer ID, and boolean value indicating whether to include device name in advertising packets, it may not produce concrete values if the value cannot be resolved without real execution (*e.g.*, `getDeviceId`). In the latter case, PeriScope collects all related text information in the corresponding data-flow path such as the name of a variable, a method, or a class, for the convenience of further analysis in §4.4.

4.3 Revealing GATT Server Configuration

If an app-defined peripheral works as a connectable peripheral, it has to maintain a GATT server, which can be configured using relevant system APIs. For example, developers can invoke the `BluetoothGattDescriptor` API to initiate an instance of a descriptor with specified UUID and permissions. In addition, such an instance can use `setValue` to store or update its data. In particular, all APIs like this that initiates an instance of an attribute or places values in an attribute are of our interest because they could be the (*a*) the UUID of an attribute, (*b*) its permissions, and (*c*) its stored values (the full list is presented in Table 7 in Appendix A).

Revealing Configurations. To reveal configurations of a GATT server, PeriScope has to resolve the input values to the configuration APIs. Similar to resolving input values in APIs that customize data in advertising packets, PeriScope applies the same algorithm but focuses on different APIs. Additionally, since the GATT server may contain several GATT attributes that are labelled by UUIDs and they are constructed hierarchically, to comprehensively reveal the server configuration, PeriScope also reconstructs the hierarchy from the data dependencies among those attributes. That is, an instance of a GATT service can use `addCharacteristic` to add an instance of a GATT characteristic, and an instance of a GATT characteristic can leverage on `addDescriptor` to add an instance of a GATT descriptor. Therefore, by traversing the IDDG, PeriScope can identify such dependencies and reconstruct the hierarchy.

4.4 Security and Privacy Analysis

Once both the advertising and GATT server configurations have been uncovered, the final step of PeriScope is to conduct security and privacy analysis to identify related vulnerabilities. Considering different data fields may result in different vulnerabilities, either passive eavesdropping or active MITM, PeriScope designs a series of policies for their detection.

Passive Eavesdropping Vulnerability Detection. Since a BLE peripheral broadcasts advertising packets to all nearby devices, the passive eavesdropping vulnerability focuses on whether such packets contain sensitive data. Due to the different nature of the data stored in each customizable field, their vulnerabilities are determined by different policies.

- **Identifiable Service UUID**: The service UUID, can be a static value (*i.e.*, hardcoded value), a dynamic value, or a hybrid one. Unfortunately, the static UUIDs could be used for device fingerprinting [11, 53] (*i.e.*, device identifiable information) and the hybrid one could lead to user tracking (§3.1), *i.e.*, personal identifiable information. Therefore, PeriScope identifies these two types of identifiable information by inspecting the generation of each UUID, which is achievable in the process of advertising value resolution in §4.2.
- **Identifiable Device Name**: This field carries the name of a smartphone (*i.e.*, Alice's Pixel) by default if not been specified with a customized name. As such, it could be a PII resulting in identity tracking. To understand which value is carried in this data field, PeriScope depends on the system API (*i.e.*, `setName`) which is used to specify a customized name. Therefore, PeriScope identifies this vulnerability by inspecting whether an app has declared to include device name in advertising packets without invoking this API.
- **Sensitive Data in Specific Fields**: The two specific data fields, *i.e.*, advertising data and manufacturer specific data, are highly customized by each companion app for service purposes, which makes them possible to carry sensitive data. Since sensitive data may come from system APIs or app custom APIs which lack documentation, without loss of generality, PeriScope identifies this vulnerability if a piece of data in such fields are from sensitive system APIs (*e.g.*, `Location.getLatitude`).

In addition, in respect of the manufacturer ID which is uniquely assigned to a specific manufacturer by the SIG and publicly available, while its security risk is relatively low since there is no known attacks on it in the literature and in the preliminary study, its usages will also analyzed in this study because it must comply with the SIG policy that has not been well studied.

Active MITM Vulnerability Detection. The active MITM vulnerability can be detected by resolving the permissions assigned to each GATT attribute. Both access permission and encryption permission are vulnerable to the active MITM attack. In Android, it assigns the access permission to an attribute using `PERMISSION_READ` or `PERMISSION_WRITE` and the encryption permission via assigning `PERMISSION_WRITE_ENCRYPTED` or `PERMISSION_READ_ENCRYPTED`. Therefore, PeriScope uses these four Android permissions to detect the corresponding active MITM vulnerability.

In addition, to further evaluate the impact of active MITM attacks, it requires to understand the functionality of each weak protected GATT attribute, and PeriScope uses two strategies for this purpose. First, PeriScope leverages the UUID to understand the functionality. Since a GATT attribute is assigned with either a standard or a custom UUID and its usages of standard UUIDs are well defined, the functionally of an attribute labelled by a standard one can be understood by checking the documentation of such a UUID. Second,

with respect to the attributes labelled by custom UUIDs, since these UUIDs are rarely documented, PeriScope intends to depend on the data generation to infer the functionality of these attributes. While it had been studied in previous works to infer the meaning of a data structure from the code through applying natural language processing (NLP) techniques on the semantics-rich program elements such as the name of variables, constants, and methods (*i.e.*, ClueFinder [28]), the fundamental limitation of such approaches is the incapability of handling code obfuscation or if there is limited semantic information, which have been observed prevalently in peripheral apps. Therefore, PeriScope partially depends on the system APIs (*i.e.*, `getDeviceId`) since they are rarely obfuscated. Moreover, with respect to the data generated from custom methods, PeriScope requires human experts involvements.

5 EVALUATION RESULTS

5.1 Evaluation Setup

App Collection. The dataset is built atop the apps that are collected from AndroZoo [3] by the end of September 2021. While there are more than 10 million apps available in AndroZoo, many of them are duplicated and outdated. To filter such duplication, we inspect the metadata of these apps and found 4 million unique apps based on their package name. In addition, after cross-checking them with the Google Play, we finally find 2.4 million apps.

Environment Setup. This study has two parts. First, we use a server which is quipped with the Intel Xeon E5-2695 CPU with 256 GB memory running Ubuntu 16.04 to collect apps from AndroZoo. Second, the program analysis is conducted on another server that is equipped with the AMD EPYC 7251 CPU with 128 GB memory running Ubuntu 18.04.

Item	Value
App Defined Bluetooth Peripherals	1,160
Google Play Categories	28
Advertising Packets Customization	
Service UUID	802
Service Data	228
Manufacturer ID & Data	501
Device Name	650
Connectivity	662
GATT Servier Configuration	
Service	786
Characteristic	1,266
Descriptor	486

Table 1: Overall Statistics of Experimental Results.

5.2 Overall Results

We present the overall results of our analysis in Table 1. In total, PeriScope has identified 1,160 *AdBP* companion apps. Specifically, there are 802 apps that broadcast static or hybrid service UUIDs, 228 app that also place service Data in advertising packets, 501 apps that contain manufacturer IDs and manufacturer specific data in their advertising packets, and 650 apps that broadcast their device names. Moreover, it has also identified 662 apps that allow other devices to connect. In addition, the GATT servers of these apps contain 786 services, 1,266 characteristics, and 486 descriptors.

5.3 Ecosystem Analysis

Based on our best knowledge, this paper is the first study to systematically analyze the mobile app-defined Bluetooth peripherals at scale. As such, we would like to present a measurement study on the current ecosystem. To this end, we need to combine the metadata of the corresponding apps and the results from program analysis. At a high level, our ecosystem analysis involves the app distribution and the involved manufacturers.

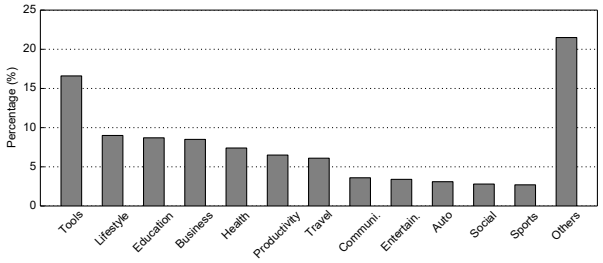

Figure 6: Distribution of the Apps that define BLE peripherals.

App Distribution. First, we use the metadata of apps to understand their distribution based on the category provided by Google Play. As presented in Table 1, 1,160 apps in our dataset come from 28 categories, and Figure 6 shows their distributions according to the number of apps in a category, which highlights categories accounting for more than 2.5% and puts the rest in "others". In particular, the top 5 categories are "Tools" (193), "Lifestyle" (104), "Education" (101), "Health" (99), and "Business" (86). Additionally, there are another two categories "Productivity" (76) and "Travel" (71) that contain more than 5% apps. In addition to the categories shown in Figure 6, there are 8 categories that contain less than 1% apps, such as "Shopping", "Dating", and "Parenting".

Manufacturers			
Ericsson	Nokia	Intel	Qualcomm
IBM	Microsoft	ST Micro.	Syntronix
MediaTek	Marvell	Apple	Avago
Nordic	MiCommand	Band XI	Zomm
Belkin	Quuppa	Typo Pro.	Swipp
Samsung	Nike	Alpwise	ARP
Quuppa Oy	Google	Comodule GMBH.	Unikey
Disney	WiSilica	Trividia	Typo
Enlighted	LINKIO SAS	BlueUp	Xiaomi
Huawei	Currant	Bestechnic	Powercast
Grundfos	MEGA-F	LEMONJOY	Withings
Frogblue	MIWA LOCK	Engineered Audio	OnAsset
Minew	SmartAction	Intellithings	Noodle
Wernher von Braun Center for ASdvanced Research			Kroger
Automotive Data Solutions Inc		Seitec Elektronik	LG
Bruel & Kjaer Sound & Vibration		GL Solutions K.K.	

Table 2: List of Manufacturer Associated with Indentified IDs.

Participated Manufacturer. A manufacturer can be identified by its manufacturer ID, which is a unique number assigned by the Bluetooth SIG. To understand how many manufacturers have been

involved in this ecosystem, we first convert the manufacturer ID uncovered from an app to the company to which such an ID belongs according to the public database provided by Bluetooth SIG [32]. In total, we have identified 61 manufacturers (shown in Table 2).

Unfortunately, *we also noticed a severe manufacturer ID abuse problem.* A manufacturer ID is uniquely assigned to a manufacturer by the Bluetooth SIG, manufacturers can temporarily use the preserved ID (*i.e.*, "65535") for internal testing when their requested IDs have not been assigned. In particular, the preserved ID is also disallowed to appear in shipping products [32]. By comparing the developer information of an app and the company associated with its used ID, we have identified three types of manufacturer ID abuses: (*a*) using manufacturer ID assigned to other companies, (*c*) using an assigned ID, and (*b*) using the preserved one for internal usages in the shipping product.

Specifically, our analysis has revealed that 351 out of 501 apps that include manufacturer ID in their advertising packets actually abuse such ID usages. As shown in Table 3, in total, we have identified 98 unique manufacturer ID from 501 apps that place this ID in their advertising packets. In particular, we only identified 28 apps that are developed by ID compliant manufacturers (20 in total) such as Samsung, Huawei, and Xiaomi use their own manufacturer ID. In addition, we also discovered 129 apps use the ID of Apple (*i.e.*, "76") in order to comply with the iBeacon protocol [5]. Finally, (*a*) there are 91 apps use IDs that have not been assigned to any manufacturer, (*b*) 243 apps that are developed by the groups that are inconsistent to the companies to which their manufacturer ID belongs, and (*c*) 17 apps that use the preserved ID for temporary testing in shipping products.

Item	Value
Peripherals w/ Manufacturer ID	501
Unique Manufacturer ID	98
Legitimate Usage	
Compliant ID Usage	28
Protocol Usage	129
Abuse Usage	
Incompliant ID Usage	243
Unassigned Manufacturer ID	91
Internal Manufacturer ID	17

Table 3: Experimental Results of Manufacturer ID Abuse.

5.4 Security Analysis

5.4.1 Passive Eavesdropping Vulnerability Analysis. According to the detection policies in §4.4, there are four data fields in the advertising packets will be evaluated to understand the impacts of the passive eavesdropping vulnerability, *i.e.*, service UUID, service data, manufacturer specific data fields, and device name.

(I) Identifiable Service UUID. This study has identified that a service UUID can be a piece of device identifiable information or personal identifiable information. Specifically, in respect of being a piece of device identifiable information, in total, we have identified 799 apps that broadcast static service UUIDs. Based on previous studies [11, 53], these apps are vulnerable to device fingerprinting attacks. On the other hand, there are 3 apps that broadcast personal identifiable service UUIDs, all of which have been verified by manually code reviewing. In particular, Sendai Broadcasting and ORIGOSafeDriver commit the same mistake to that in Lyft Driver (in §3.1) that generate their service UUID by concatenating a fixed string value to the user ID.

(II) Identifiable Device Name. There are 537 apps that put the custom phone name in device name field in advertising packets. These device names can be used as personal identifiable information for user identification and tracking.

5.4.2 Active MITM Vulnerability Analysis. As shown in Table 4, we summarize the type of service and the security permissions (*i.e.*, read and write) that assigned to characteristics and descriptors. Specifically, most of service (98.73%) are declared as primary service. Surprisingly, we found that the majority of both characteristics and descriptors is configured with access permission. In particular, 94.44% characteristics as well as 96.59% descriptors are only specified with read permission without further security requirement, and 94.78% characteristics along with 94.71% descriptors declare the lowest security level of the write permission. With respect to these vulnerable attributes, we further investigate them based on their functionality to identify whether they contain sensitive data.

We first inspect whether sensitive data generated from system APIs has been contained in attributes with access or encryption permission, and we have not identified a single case. Therefore, we next need to manually analyze these apps to identify whether their not well-protected attributes contain any sensitive data. While we wish to confirm all improper access controls on sensitive data for all apps, this turns out challenging because of the missing documentation, the limited information available in the app, and limited man power. Instead, we primarily focus on the apps belong to the top 5 categories and present the details of the top 5 apps in each of these categories in Table 5 with our best understanding.

				Access Permission		
Attribute	**Total**	**P**	**S**	Ac.	En.	Au.
Service	786	776	10	—	—	—
Characteristic	1,266					
R	990	—	—	935	28	27
W	760	—	—	721	19	20
Descriptor	486					
R	440	—	—	425	12	3
W	473	—	—	448	12	13

Table 4: Evaluation of GATT Attribute Configurations: P for primary, S for secondary, R for read, W for write, Ac. for access, En. for encryption, and Au. for authentication.

Vulnerable Attributes With Standard UUIDs. Our analysis have discovered 11 service, 30 characteristic, and 6 descriptor that are labelled by standard UUIDs. After reviewing their functionality, we have identified that many of them contain insensitive data, such as current time (the full list of the identified standard UUIDs are presented in Table 8 in Appendix A). We have identified 3 standard services and their associated 6 characteristics are privacy sensitive. Specifically, these sensitive services and characteristics

include (*i*) *Running Speed and Cadence (RSC)* (`0x1814`) service that contains two characteristics: *RSC Measurement* (`0x2A53`) and *RSC Feature* (`0x2A54`), (*ii*) *Heart Rate* (`0x180D`) service that contains *Heart Rate Measurement* (`0x2A7A`), and (*iii*) *Cycling Speed and Cadence (CSC)* (`0x1816`) service that contains three characteristics: *CSC Measurement* (`0x2A5B`), `CSC Feature` (`0x2A5C`), and *Sensor Location* (`0x2A5D`). In addition, these sensitive attributes belong to two apps in the "Health" category. Specifically, as shown in Table 5, one app contains two services: the RSC and Heart Rate, which allows any connected centrals to the running speed and hear rate of the user, and another app provides the CSC service allowing a connected device to access the cycling data, which includes the wheel and crank revolution.

Vulnerable Attributes With Custom UUIDs. Unlike standard UUIDs, custom UUIDs are rarely documented. As such, we manually review the code to understand the service and functionality these attributes provide. As shown in Table 5, there are 7 apps that put identifiers in a vulnerable characteristic that is subject to device and user identity recognition, 7 apps that store app specific commands that should be well protected as business secrets, 7 apps can leak a variety types of private information (*i.e.*, communication messages, personal preferences, fitness data, email address, and WIFI ssid and password), and the rest 4 apps could leak the credentials related to their safety. Specifically, while there are overlaps, apps from different categories manage different types of data. In particular, apps from the category of "tools" could store digital keys to open doors or unlock accounts, education apps would contain the digital identifiers for class attendance check-in, apps monitoring health and fitness record related private data, and apps from the rest two categories manage their application specific data such as configuration metadata and private commands to retrieve desired data.

		U. Type		Opr.		Perm.			
Cat.	App Name	St.	Cu.	R	W	No.	En.	Au.	Sensitive Data
Tools	T1	○	●	●	●	●	○	○	Door unlock key
	T2	○	●	●	●	●	○	○	Door unlock key
	T3	○	●	●	●	●	○	○	Device control commands
	T4	○	●	●	○	●	○	○	WIFI SSID and password
	T5	○	●	●	○	●	○	○	User accounts unlock keys
Education	E1	○	●	●	○	●	○	○	User check-in ID
	E2	○	●	●	○	●	○	○	User check-in ID
	E3	○	●	●	○	●	○	○	User check-in ID
	E4	○	●	●	○	●	○	○	Personal preferences
	E5	○	●	●	○	●	○	○	Personal preferences
Lifestyle	L1	○	●	●	●	●	○	○	Device control commands
	L2	○	●	●	●	●	○	○	Vehicle diagnostics commands
	L3	○	●	●	●	●	○	○	Vehicle diagnostics commands
	L4	○	●	●	●	●	○	○	Vehicle diagnostics commands
	L5	○	●	●	●	●	○	○	Vehicle diagnostics commands
Health	Aarogya Setu	○	●	●	○	●	○	○	Contact tracing User ID
	H2	○	●	●	○	●	○	○	Communication Tokens
	H3	○	●	●	○	●	○	○	Device control Commands
	H4	●	○	●	○	●	○	○	Fitness data
	H5	●	○	●	○	●	○	○	Fitness data
Business	B1	○	●	●	○	●	○	○	User ID
	B2	○	●	●	○	●	○	○	IoT device ID
	B3	○	●	●	○	●	○	○	User ID
	B4	○	●	●	○	●	○	○	Configuration metadata
	B5	○	●	●	○	●	○	○	Private messages

Table 5: Evaluation of Improper Access Control of Top App-Defined Bluetooth Peripherals: Cat. for category, U. Type ofr UUID type, St. for standard, Cu. for custom, Opr. for operation, R for read, W for write, Perm. for access permission, No. for no protection, En. for encryption protection, and Au. for authentication protection.

6 DISCUSSION

6.1 The Good, The Bad, and The Ugly

Mobile operating systems (*e.g.*, Android) provide a number of system APIs for an *AdBP* companion app to flexibly define a Bluetooth peripheral. The good aspect of this practice is that mobile operating systems take over the configuration of the link layer information in the advertising packets including the automatic rotation of MAC address, which is usually a fixed value in firmware defined Bluetooth peripherals leading to tracking vulnerabilities. On the other hand, *AdBP* developers have not been found in this study to mistakenly place sensitive data in the fields for specific data in advertising packets (*i.e.*, service data and manufacturer specific data), which has also been partially verified by manually inspecting the apps belong to the top 5 categories with our best understanding.

Unfortunately, it also impedes the secure development for mobile operating systems to entirely handle some critical security enforcement processes (*e.g.*, negotiating the pairing and bonding policy) in the link layer. While this implementation may ease the development effort, *AdBP* companion apps cannot enforce secure pairing policies and expose themselves to the MITM or downgrade attack [51] if forcing the "Just Works" pairing policy by connecting devices. Moreover, it may also let developers entirely neglect the potentials risks resulted from such processes. In this study, it has been identified that around 95% pieces of data stored in GATT attributes are subject to the active MITM attack, some of which have been found leaking sensitive data including but not limited to personal health data, digital identifiers of users, and even digital keys to unlock doors.

On the other hand, while turning mobile phones as Bluetooth peripherals can provide a variety of services, developers may need to double-thought whether it is necessary for this implementation. In particular, an arbitrary usage of such a functionality may interfere the normal functionality of other apps since only one app is allowed to function at each time. Moreover, a large number of developers have been identified to violate the manufacturer ID usage policy.

6.2 Limitations and Future Works

Though PeriScope has identified a great number of vulnerabilities in app-defined Bluetooth peripherals, it still has limitations that require further improvements. First, its program analysis focuses on system APIs at the Java bytecode level, and thus may miss the peripherals that are defined using other techniques such as JavaScripts in WebViews, which might raises false negatives. Therefore, one future work to extend the ability to cover additional implementations using other sets of APIs. Second, while PeriScope can automatically identify many types of sensitive data such as static UUIDs and the custom name of the phone, it still requires manual efforts to identify certain cases such as recognizing the types of data that are placed in GATT characteristics and descriptors. As such, another future work is to explore the feasibility to use machine learning expertise to automatically recognize the functionality of a custom UUID.

6.3 Ethics and Responsible Disclosure

We take ethics seriously and only launch such attacks on our own devices for the proof of concept prove purposes and never intend to exploit any vulnerability on other users' devices.

Responsible Disclosure. We immediately contacted developers of the vulnerable apps according to their contact information provided in Google Play. At the time of this writing, we have received confirmation and bug bounty from Lyft. We will keep in touch with developers and coordinate with them to fix our identified vulnerabilities. In addition, names of vulnerable apps are redacted unless being fixed or unavailable in Google Play.

6.4 Guidelines Of Secure *AdBP* Development

This study is motivated by a wish to identify the potential vulnerabilities and prevent associated severe security and privacy impacts in the emerging category of *AdBP* applications at their early stage. In order to improve the security of these applications, we propose a set of guidelines for secure *AdBP* development.

Excluding Identifiable Information. This study has identified a large number of *AdBP* companion apps carrying either device or personal identifiable information, or both. In any case, it has been demonstrated to result in privacy leakages. In fact, fields in advertising packets containing such sensitive identifiable information are entirely customized by developers. Therefore, the guideline for *AdBP* secure development is to exclude any identifiable information in advertising packets, especially being in cleartext.

Assigning Appropriate Permissions. Another common mistake made in the process of development that is identified in this empirical study is that the vast majority of GATT attributes have been assigned no protection and a small portion of the rest attributes has been assigned with encryption permission. Unfortunately, either of these two protection exposes pieces of sensitive data (if they were) stored in an attribute to attackers. As such, another guideline in secure *AdBP* development is that always assign the highest level of security permission (*i.e.*, authentication permission in Android) to GATT attributes that might store sensitive pieces of data. Further to this, developers should be careful when assigning read and write permissions to attributes, as these may not always be necessary for the required functionality.

Enforcing Secure Pairing Policies. While it eases the development of an *AdBP* that the mobile operating system handles the pairing and bonding procedure including the exchange of I/O information to negotiate paring protocol, priori work has demonstrated that this mode is subject to the downgrade attack [51]. This attack not only shows the ineffectiveness of encryption permission in certain circumstances but also indicates an additional requirement for secure development: enforcing secure pairing policy before any access to any GATT attribute, if at least one attribute might contain sensitive information.

7 RELATED WORK

BLE and IoT Security. There has been a large body of works that analyze BLE vulnerabilities in IoT devices. Some of them focus on vulnerabilities in the protocol, such as leaking credentials in the pairing procedure [16] and the unencrypted channel [50], eavesdropping vulnerability in the passkey pairing protocol in both BLE 4.0 and 4.1 [30], bruce-force attacks on the long term key [49], reusable passkey to break the Passkey Entry paring procedure [35, 37], enforceable insecure pairing methods [51], and the consequent privacy violations [12, 20] . In addition, some other works also focus on vulnerabilities in BLE-enabled IoT devices with conclusions that most wearable devices are subject to privacy disclosure [15], IoT devices are subject to the misconfiguration of privileges [21, 24, 45], unchanged MAC addresses [9, 17, 45], and fingerprintable static UUIDs [11, 45, 53]. In addition, many efforts also focus on other attack surfaces may also compromise Bluetooth in smartphones, such as AT commands [25, 40], reconnecting between two paired devices [47], and interactions between OS [48] and co-located apps [36]. Unlike works analyzing security in BLE protocols and IoT devices, our study focuses on the security and privacy in app-define Bluetooth peripherals.

Mobile Apps Analysis. There is a large body of works that analyze mobile apps to identify vulnerabilities using static or dynamic analysis. For example, TaintDroid [19] uses taint analysis dynamically to identify user privacy data leakages. With respect to the static analysis, Flowdroid [6] and Amandroid [44] are designed to track security-related data flows to identify data leakage, WARDroid [27] and Extractocol [14] target at network relevant data flows to identify related issues, and PlayDrone [43] is capable of extracting static secret keys to analyze vulnerabilities in cloud-based services. Complementary to these works, we use static analysis to uncover link layer configurations of a Bluetooth peripheral defined by mobile apps.

Contact Tracing Analysis. Contact tracing apps can turn a smartphone into a Bluetooth peripheral, and there are several works [26, 34, 38, 46, 52] that present empirical studies inspecting the privacy issues in these apps. Additionally, some efforts also provide in-depth studies on specific apps. For instance, Cho *et al.* [13] focus on TraceTogether and Veale [42] targets the NHS COVID-19 App. Compared to previous works that primarily analyze privacy problems in a relatively small set of apps, our study focuses on the security and privacy practice in app-defined Bluetooth peripherals at scale.

8 CONCLUSION

This paper presents PeriScope, an automated tool that unveils the security and privacy vulnerabilities at the link layer of mobile app-defined Bluetooth peripherals. Specifically, it introduces a series of automated program analysis techniques in precise recognition of companion apps, accurate program value resolution to uncover their link layer functionality implementations, and effective vulnerabilities identification. PeriScope has recognized 1,160 Bluetooth peripheral apps from Google Play, and identified 69.13% of them that broadcast device or personal identifiable information in cleartext and 95% pieces of data stored in all recognized apps that can be accessed by any connected devices without authentication, which leads to sensitive data leakages. In addition, it also discovered a severe manufacturer ID abuse in current ecosystem violating the associated usage policy. Finally, a set of guidelines for secure app-defined Bluetooth peripheral development is provided with the hope of preventing the potential severe security impacts at the early stage.

9 ACKNOWLEDGEMENT

We thank anonymous reviewers for their insightful comments. This research was partially supported by NSF Awards 1834215 and 1834216.

REFERENCES

[1] Mohammad Afaneh. 2020. Bluetooth GATT: How to Design Custom Services & Characteristics. https://www.novelbits.io/bluetooth-gatt-services-characteristics/.

[2] Nadeem Ahmed, Regio A Michelin, Wanli Xue, Sushmita Ruj, Robert Malaney, Salil S Kanhere, Aruna Seneviratne, Wen Hu, Helge Janicke, and Sanjay K Jha. 2020. A survey of covid-19 contact tracing apps. *IEEE Access* 8 (2020), 134577–134601.

[3] Kevin Allix, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2016. AndroZoo: Collecting Millions of Android Apps for the Research Community. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR '16)*. ACM, New York, NY, USA, 468–471. https://doi.org/10.1145/2901739.2903508

[4] Apktool. 2022. A tool for reverse engineering 3rd party, closed, binary Android apps. https://ibotpeaches.github.io/Apktool/.

[5] Apple. 2022. iBeacon. https://developer.apple.com/ibeacon/.

[6] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices* 49, 6 (2014), 259–269.

[7] Asilentkingdom. 2014. BLE pairing vs. bonding. https://piratecomm.wordpress.com/2014/01/19/ble-pairing-vs-bonding/.

[8] Johannes K Becker, David Li, and David Starobinski. 2019. Tracking anonymized bluetooth devices. *Proceedings on Privacy Enhancing Technologies* 2019, 3 (2019), 50–65.

[9] 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.

[10] Yinzhi Cao, Yanick Fratantonio, Antonio Bianchi, Manuel Egele, Christopher Kruegel, Giovanni Vigna, and Yan Chen. 2015. EdgeMiner: Automatically Detecting Implicit Control Flow Transitions through the Android Framework.. In *NDSS*.

[11] Guillaume Celosia and Mathieu Cunche. 2019. Fingerprinting Bluetooth-Low-Energy Devices Based on the Generic Attribute Profile. In *Proceedings of the 2nd International ACM Workshop on Security and Privacy for the Internet-of-Things*.

[12] Guillaume Celosia and Mathieu Cunche. 2019. Saving private addresses: An analysis of privacy issues in the bluetooth-low-energy advertising mechanism. In *Proceedings of the 16th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*.

[13] Hyunghoon Cho, Daphne Ippolito, and Yun William Yu. 2020. Contact tracing mobile apps for COVID-19: Privacy considerations and related trade-offs. *arXiv preprint arXiv:2003.11511* (2020).

[14] Hyunwoo Choi, Jeongmin Kim, Hyunwook Hong, Yongdae Kim, Jonghyup Lee, and Dongsu Han. 2015. Extractocol: Automatic Extraction of Application-level Protocol Behaviors for Android Applications. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM '15)*. ACM, New York, NY, USA, 593–594. https://doi.org/10.1145/2785956.2790003

[15] Brian Cusack, Bryce Antony, Gerard Ward, and Shaunak Mody. 2017. Assessment of security vulnerabilities in wearable devices. (2017).

[16] Britt Cyr, Webb Horn, Daniela Miao, and Michael Specter. 2014. Security analysis of wearable fitness devices (fitbit). *Massachusets Institute of Technology* (2014).

[17] 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.

[18] Dex2jar. 2021. Tools to work with android .dex and java .class files. https://github.com/pxb1988/dex2jar.

[19] W. Enck, P. Gilbert, B.G. Chun, L.P. Cox, J. Jung, P. McDaniel, and A.N. Sheth. 2010. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI*.

[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] Daniel Filizzola, S. Fraser, and Nikita Samsonau. 2018. Security Analysis of Bluetooth Technology.

[23] Google. 2021. BluetoothGattDescriptor. https://developer.android.com/reference/android/bluetooth/BluetoothGattDescriptor.

[24] 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.

[25] Imtiaz Karim, Fabrizio Cicala, Syed Rafiul Hussain, Omar Chowdhury, and Elisa Bertino. 2019. Opening Pandora's box through ATFuzzer: dynamic analysis of AT interface for Android smartphones. In *Proceedings of the 35th Annual Computer Security Applications Conference*. 529–543.

[26] Jinfeng Li and Xinyi Guo. 2020. COVID-19 Contact-tracing Apps: A Survey on the Global Deployment and Challenges. *arXiv preprint arXiv:2005.03599* (2020).

[27] 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)*.

[28] Yuhong Nan, Zhemin Yang, Xiaofeng Wang, Yuan Zhang, Donglai Zhu, and Min Yang. 2018. Finding Clues for Your Secrets: Semantics-Driven, Learning-Based Privacy Discovery in Mobile Apps. In *Proceedings of the 2018 Network and Distributed System Security Symposium*.

[29] O'Reilly. 2022. Getting Started with Bluetooth Low Energy. https://www.oreilly.com/library/view/getting-started-with/9781491900550/ch04.html.

[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] Bluetooth SIG. 2022. Bluetooth Pairing Part 4. https://www.bluetooth.com/blog/bluetooth-pairing-part-4/.

[32] Bluetooth SIG. 2022. Company Identifiers. https://www.bluetooth.com/specifications/assigned-numbers/company-identifiers/

[33] Bluetooth SIG. 2022. A Developer's Guide To Bluetooth. https://www.bluetooth.com/blog/a-developers-guide-to-bluetooth/.

[34] Lucy Simko, Ryan Calo, Franziska Roesner, and Tadayoshi Kohno. 2020. COVID-19 Contact Tracing and Privacy: Studying Opinion and Preferences. *arXiv preprint arXiv:2005.06056* (2020).

[35] Pallavi Sivakumaran and Jorge Blasco. 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.

[36] Pallavi Sivakumaran and Jorge Blasco. 2019. A study of the feasibility of co-located app attacks against BLE and a large-scale analysis of the current application-layer security landscape. In *28th USENIX Security Symposium*.

[37] 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.

[38] Qiang Tang. 2020. Privacy-preserving contact tracing: current solutions and open questions. *arXiv preprint arXiv:2004.06818* (2020).

[39] Bluetooth® Technology. 2020. Blood Pressure Service. https://www.bluetooth.com/xml-viewer/?src=https://www.bluetooth.com/wp-content/uploads/Sitecore-Media-Library/Gatt/Xml/Services/org.bluetooth.service.blood_pressure.xml.

[40] Dave Jing Tian, Grant Hernandez, Joseph I Choi, Vanessa Frost, Christie Raules, Patrick Traynor, Hayawardh Vijayakumar, Lee Harrison, Amir Rahmati, Michael Grace, et al. 2018. Attention spanned: Comprehensive vulnerability analysis of {AT} commands within the android ecosystem. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 273–290.

[41] Kevin Townsend. 2014. Introduction to Bluetooth Low Energy. https://learn.adafruit.com/introduction-to-bluetooth-low-energy/gap.

[42] Michael Veale. 2020. Analysis of the NHSX Contact Tracing App 'isle of Wight'Data Protection Impact Assessment. (2020).

[43] Nicolas Viennot, Edward Garcia, and Jason Nieh. 2014. A measurement study of google play. In *ACM SIGMETRICS / International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '14, Austin, TX, USA - June 16 - 20, 2014*. 221–233. https://doi.org/10.1145/2591971.2592003

[44] 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.

[45] Haohuang Wen, Zhiqiang Lin, and Yinqian Zhang. 2020. FirmXRay: Detecting Bluetooth Link Layer Vulnerabilities from Bare-Metal Firmware. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*.

[46] Haohuang Wen, Qingchuan Zhao, Zhiqiang Lin, Dong Xuan, and Ness Shroff. 2020. A Study of the Privacy of COVID-19 Contact Tracing Apps. In *International Conference on Security and Privacy in Communication Networks*.

[47] Jianliang Wu, Yuhong Nan, Vireshwar Kumar, Dave Jing Tian, Antonio Bianchi, Mathias Payer, and Dongyan Xu. 2020. {BLESA}: Spoofing Attacks against Reconnections in Bluetooth Low Energy. In *14th {USENIX} Workshop on Offensive Technologies ({WOOT} 20)*.

[48] Fenghao Xu, Wenrui Diao, Zhou Li, Jiongyi Chen, and Kehuan Zhang. 2019. BadBluetooth: Breaking Android Security Mechanisms via Malicious Bluetooth Peripherals.. In *NDSS*.

[49] Wondimu K Zegeye. 2015. Exploiting Bluetooth low energy pairing vulnerability in telemedicine. In *International Telemetering Conference Proceedings*. International Foundation for Telemetering.
[50] 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.
[51] Yue Zhang, Jian Weng, Rajib Dey, Yier Jin, Zhiqiang Lin, and Xinwen Fu. 2020. Breaking Secure Pairing of Bluetooth Low Energy Using Downgrade Attacks. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association. https://www.usenix.org/conference/usenixsecurity20/presentation/zhang-yue
[52] Qingchuan Zhao, Haohuang Wen, Zhiqiang Lin, Dong Xuan, and Ness Shroff. 2020. On the Accuracy of Measured Proximity of Bluetooth-based Contact Tracing Apps. In *International Conference on Security and Privacy in Communication Networks*.
[53] Chaoshun Zuo, Haohuang Wen, Zhiqiang Lin, and Yinqian Zhang. 2019. Automatic Fingerprinting of Vulnerable BLE IoT Devices with Static UUIDs from Mobile Apps. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*.

A APPENDIX

Class Name	Method Name	Parameters
AdvertiseSettings .Builder	setAdvertiseMode	int advertiseMode
	setConnectable	boolean connectable
	setTimeout	int timeoutMillis
	setTxPowerLevel	int txPowerLevel
AdvertiseData .Builder	addManufacturerData	int manufacturerId, byte[] manufacturerSpecificData
	addServiceData	ParcelUuid serviceDataUuid, byte[] serviceData
	addServiceUuid	ParcelUuid serviceUuid
	setIncludeDeviceName	boolean includeDeviceName
	setIncludeTxPowerLevel	boolean includeTxPowerLevel

Table 6: List of System APIs to Configure BLE Advertising

Class Name	Method Name	Parameters
BluetoothGattServer	addService	BluetoothGattService service
BluetoothGattService	<init>	UUID uuid, int serviceType
	addCharacteristic	BluetoothGattCharacteristic ch
	addService	BluetoothGattService service
BluetoothGattCharacteristic	<init>	UUID uuid, int properties, int permissions
	addDescriptor	BluetoothGattDescriptor desc
	setValue	String value
	setValue	int value, int formatType, int offset
	setValue	byte[] value
	setValue	int mantissa, int exponent, int formatType, int offset
BluetoothGattDescriptor	<init>	UUID uuid, int permissions
	setValue	byte[] value

Table 7: List of System APIs to Configure GATT Server

A.1 Android System APIs Configuring *AdBP*

Android provides a set of system APIs for developers to configure an *AdBP*. In particular, an *AdBP* and other types of BLE peripherals can work in two modes to provide services. One is to work as a non-connectable device (*i.e.*, beacon) and the other one is to function as a connectable device. Similar to normal BLE peripherals, an *AdBP* should be able to work in two modes as well. In this regard, as shown in Table 6, Android system APIs allow Android developers to configure (*a*) broadcasting behaviors (for beacons) and (*b*) the GATT server (for connectable peripherals). In addition, as mentioned in §4.3, an *AdBP* also uses system APIs to construct the hierarchy. The full list of these APIs are shown in Table 7.

Attri.	UUID	Semantic
service UUID	00001802-0000-1000-8000-00805F9B34FB	Immediate Alert
	00001805-0000-1000-8000-00805F9B34FB	Current Time Service
	0000180A-0000-1000-8000-00805F9B34FB	Device Information
	0000180F-0000-1000-8000-00805F9B34FB	Battery Service
	00001814-0000-1000-8000-00805F9B34FB	Running Speed and Cadence
	0000180D-0000-1000-8000-00805F9B34FB	Heart Rate
	00001800-0000-1000-8000-00805F9B34FB	Generic Access
	00001812-0000-1000-8000-00805F9B34FB	Human Interface Device
	00001816-0000-1000-8000-00805F9B34FB	Cycling Speed and Cadence
	00001818-0000-1000-8000-00805F9B34FB	Cycling Power
	00001826-0000-1000-8000-00805F9B34FB	Fitness Machine
characteristic UUID	00002A28-0000-1000-8000-00805F9B34FB	Software Revision String
	00002A29-0000-1000-8000-00805F9B34FB	Manufacturer Name String
	00002A2B-0000-1000-8000-00805F9B34FB	Current Time
	00002A0F-0000-1000-8000-00805F9B34FB	Local Time Information
	00002A24-0000-1000-8000-00805F9B34FB	Model Number String
	00002A25-0000-1000-8000-00805F9B34FB	Serial Number String
	00002A26-0000-1000-8000-00805F9B34FB	Firmware Revision String
	00002A06-0000-1000-8000-00805F9B34FB	Alert Level
	00002A19-0000-1000-8000-00805F9B34FB	Battery Level
	00002A23-0000-1000-8000-00805F9B34FB	System ID
	00002A53-0000-1000-8000-00805F9B34FB	RSC Measurement
	00002A54-0000-1000-8000-00805F9B34FB	RSC Feature
	00002A7A-0000-1000-8000-00805F9B34FB	Heart Rate Measurement
	00002A2A-0000-1000-8000-00805F9B34FB	IEEE 11073-20601
	00002A4A-0000-1000-8000-00805F9B34FB	HID Information
	00002A4B-0000-1000-8000-00805F9B34FB	Report Map
	00002A4C-0000-1000-8000-00805F9B34FB	HID Control Point
	00002A4D-0000-1000-8000-00805F9B34FB	Report
	00002A5D-0000-1000-8000-00805F9B34FB	Sensor Location
	00002A63-0000-1000-8000-00805F9B34FB	Cycling Power Measurement
	00002A65-0000-1000-8000-00805F9B34FB	Cycling Power Feature
	00002A76-0000-1000-8000-00805F9B34FB	UV Index
	00002A00-0000-1000-8000-00805F9B34FB	Device Name
	00002A01-0000-1000-8000-00805F9B34FB	Appearance
	00002A02-0000-1000-8000-00805F9B34FB	Peripheral Privacy Flag
	00002A03-0000-1000-8000-00805F9B34FB	Reconnection Address
	00002A04-0000-1000-8000-00805F9B34FB	Peripheral Pref. Conn. Para.
	00002A05-0000-1000-8000-00805F9B34FB	Service Changed
	00002A08-0000-1000-8000-00805F9B34FB	Date Time
	00002A09-0000-1000-8000-00805F9B34FB	Day of Week
	00002A0A-0000-1000-8000-00805F9B34FB	Day Date Time
	00002A0C-0000-1000-8000-00805F9B34FB	Exact Time 256
	00002A31-0000-1000-8000-00805F9B34FB	Scan Refresh
	00002A38-0000-1000-8000-00805F9B34FB	Body Sensor Location
	00002A4E-0000-1000-8000-00805F9B34FB	Protocol Mode
	00002A4F-0000-1000-8000-00805F9B34FB	Scan Interval Window
	00002A55-0000-1000-8000-00805F9B34FB	SC Control Point
	00002A5B-0000-1000-8000-00805F9B34FB	CSC Measurement
	00002A5C-0000-1000-8000-00805F9B34FB	CSC Feature
decriptor UUID	00002902-0000-1000-8000-00805F9B34FB	Client Char. Configuration
	00002901-0000-1000-8000-00805F9B34FB	Char. User Description
	00002900-0000-1000-8000-00805F9B34FB	Char. Extended Properties
	00002904-0000-1000-8000-00805F9B34FB	Char. Presentation Format
	00002908-0000-1000-8000-00805F9B34FB	Report Reference
	00002907-0000-1000-8000-00805F9B34FB	External Report Reference

Table 8: Uncovered Standard UUIDs.

A.2 Uncovered Standard UUIDs in *AdBP*

This paper has discovered 11 service, 30 characteristic, and 6 descriptor that are labelled by standard UUIDs. These UUIDs as well as their semantics are listed in Table 8.