Breaking Secure Pairing of Bluetooth Low Energy Using Downgrade Attacks

Yue Zhang $^{\Delta,\gamma}$, Jian Weng$^{\Delta}$, Rajib Dey$^{\gamma}$, Yier Jin*, Zhiqiang Lin‡, and Xinwen Fu$^{\gamma}$

$^{\Delta}$College of Information Science and Technology, Jinan University
$^{\gamma}$Department of Computer Science, University of Central Florida
*Department of Electrical and Computer Engineering, University of Florida
‡Department of Computer Science and Engineering, Ohio State University

Abstract

To defeat security threats such as man-in-the-middle (MITM) attacks, Bluetooth Low Energy (BLE) 4.2 and 5.x introduced a Secure Connections Only (SCO) mode, under which a BLE device can *only* accept secure pairing such as Passkey Entry and Numeric Comparison from an initiator, e.g., an Android mobile. However, the BLE specification does not require the SCO mode for the initiator, and does not specify how the BLE programming framework should implement this mode. In this paper we show that the BLE programming framework of the initiator must properly handle SCO initiation, status management, error handling, and bond management; otherwise severe flaws can be exploited to perform downgrade attacks, forcing the BLE pairing protocols to run in an insecure mode without user's awareness. To validate our findings, we have tested 18 popular BLE commercial products with 5 Android phones. Our experimental results proved that MITM attacks (caused by downgrading) are possible to all these products. More importantly, due to such system flaws from the BLE programming framework, all BLE apps in Android are subject to our downgrade attacks. To defend against our attacks, we have built a prototype for the SCO mode on Android 8 atop Android Open Source Project (AOSP). Finally, in addition to Android, we also find all major OSes including iOS, macOS, Windows, and Linux do not support the SCO mode properly. We have reported the identified BLE pairing vulnerabilities to Bluetooth Special Interest Group, Google, Apple, Texas Instruments, and Microsoft.

1 Introduction

Bluetooth Low Energy (BLE) is a widely adopted wireless communication technology and is broadly used in many IoT applications such as retail (e.g., beacons), healthcare (e.g., blood pressure monitor), and wearables (e.g., smart watches). BLE has two salient features: (*i*) low energy consumption, increasing the lifetime of battery-powered BLE devices, and (*ii*) Generic Attribute Profile (GATT) based data transmission, allowing mobile, tablet and PC applications for arbitrary data transmission with peer BLE devices.

Being a wireless communication technology, BLE relies on pairing, under which two paired devices authenticate each other and negotiate a secret key, to encrypt the communication channel and ensure the secure communication. Latest versions of the specification ([1, 2]) introduced four association methods: (*i*) Just Works, (*ii*) Passkey Entry, (*iii*) Numeric Comparison, and (*iv*) Out Of Band (*OOB*). However, Just Works uses a plain Elliptic-curve Diffie–Hellman key exchange protocol without authentication of exchanged public keys and it is therefore subject to the Man-in-the-Middle (MITM) attack [3]. Out of Band (OOB) requires a non-Bluetooth channel such as Near Field Communication (NFC) for key exchanging to defeat passive eavesdropping and MITM attacks. It is rarely used due to the requirement of an extra non-Bluetooth channel [4]. Consequently, Passkey Entry and Numeric Comparison are actually the two practical secure association methods.

In addition to these four association methods, the latest BLE 4.2 [1] and 5.x [2] added a new Secure Connections Only (SCO) mode for BLE enabled devices to address vulnerabilities found in the previous generations of Bluetooth. For example, in Bluetooth Classic 2.1 and 3.0, Bluetooth Secure Simple Pairing (SSP) is used [5]. Under SSP, two Bluetooth devices use only input/output (I/O) capabilities (such as display and keyboard) to determine the association method. However, an attacker can falsely declare their I/O capabilities and conduct an MITM attack [5]. Therefore, with BLE 4.2 and 5.x, if a BLE device supports the SCO mode, it can be forced to authenticate the user/mobile device with secure association methods, which are expected to defeat the MITM attacks.

However, we discover that in the BLE specification, the SCO mode only specifies that a BLE device needs to authenticate the mobile device (typically the BLE connection initiator), but the mobile device is not required to authenticate the BLE device. Therefore, an attacker can spoof a victim BLE device's MAC address and other characteristics to create a fake BLE device and attack the initiators. We further discover that a proper implementation of the SCO mode is in fact quite challenging for the BLE programming framework. That is, at least four capabilities are required: (*i*) Initiation: An application shall have the capability of instructing the BLE stack the specific secure association method to enforce; (*ii*) Status management: The BLE stack shall memorize the specified secure association method, enforce it at the right time and notify the corresponding result; (*iii*) Error handling: When errors occur during communications, the BLE stack and application shall coordinate to handle these errors and enforce the specified secure association method; and (*iv*) Bond management: The application shall have the capability of removing its broken bond caused by errors in order to initiate the enforcement process again.

The lack of the above capabilities in the BLE programming framework leads to security flaws, as demonstrated in this paper. Specifically, we show that the lack of proper enforcement and handling of the SCO mode in the BLE programming framework for the mobile device can lead to a variety of attacks by a fake BLE device, including (*i*) exposure of secret data from mobile apps, e.g., a user's password for device access, and from mobiles, e.g., a mobile device's Identity Resolving Key (IRK) and MAC address; (*ii*) injection of false data to affect the mobile app data integrity. As a concrete example, even if an Android mobile was paired with a peer BLE device through secure pairing using secure association methods, a fake device can downgrade the association method into insecure ones, i.e., Just Works or even communicating in plaintext. These attacks go beyond mobiles. For instance, by stealing an Android mobile's IRK and MAC address with a fake device, an attacker can pretend to be the legitimate mobile to bypass a peer device's whitelist if there is any. Not only the BLE programming framework in Android has these security flaws, but also all other major OSes including iOS, macOS, Windows, and Linux contain them as well, as shown in our experiment.

Contributions. Our major contributions are summarized as follows.

- **Novel Discovery.** We are the first to discover that in the SCO mode, the BLE programming framework at the mobile device side must properly handle initiation, status management, error handling, and bound management during the life cycle of a BLE pairing process; any flaws among them will allow a fake device to steal secrets or tamper with sensitive data to mobile devices.
- **Practical Attacks.** We demonstrate with attacks on 18 commercial BLE devices to show the specific design flaws in the BLE programming framework of Android. These attacks also apply to all of the 18,929 BLE Android apps we examined. Our extensive experiments also confirm that the design flaws exist in all major OSes including Android, iOS, macOS, Windows and Linux while these flaws may vary in particular OSes. The attack against mobiles and peer devices may be deployed from tens of meters with off-the-shelf devices.
- **Countermeasures.** Security defenses are also proposed and prototyped to enhance the SCO mode for Android by enforcing secure association methods in Android Open Source Project (AOSP) [6]. Our security analysis with BLE keyboards further shows that Numerical Comparison is more secure than Passkey Entry when both the mobile and the peer device enforce secure pairing.

Responsible Disclosures: We have reported our findings to Bluetooth Special Interest Group (SIG), Google Android Security Team, Apple, Windows, and Texas Instruments (TI) Product Security Incident Response Team (PSIRT). Googled rated the identified Android vulnerabilities as **High** severity and released a patch in December 2019 Android Security Bulletin, which fixes part of the issue. TI patched its BLE stack [7]. Progress with Apple can be tracked through CVE-2020-9770. The Microsoft Security Response Center (MSRC) assigned a vulnerability tracking number (VULN-012119) to the raised issues.

2 Background

2.1 BLE Protocol Stack

BLE is a short-range wireless communications technology. Figure 1 shows its protocol stack using a BLE-equipped blood pressure monitor as an example. As illustrated, there are two apps involved: one running in the blood pressure monitor, and the other running in the mobile device such as Android. These two apps use the *BLE core system* for communication, which consists of two building blocks: LE controller and host. The LE controller uses the link layer and physical layer to create a connection for sending/receiving data. The physical layer uses frequency hopping for communication, where data is exchanged over a sequence of hopping frequencies, which is negotiated between two devices. The host implements multiple protocols including the Security Manager Protocol (SMP) and Attribute Protocol (ATT) for secure communication. The Host Controller Interface (HCI) moves data, e.g., blood pressure measurements or SMP control commands, from the host to the LE controller through a physical interface, a function call or other venues depending on specific implementations.

2.2 BLE Workflow

The typical workflow between a BLE master (e.g., the mobile device) and slave (e.g., the blood pressure monitor) is illustrated in Figure 2. In total, there are 11 steps within three stages: (*i*) Connection, (*ii*) Pairing (which is optional), and (*iii*) Communication. A typical BLE connection setup process goes through steps 1 to 4. After the two BLE devices

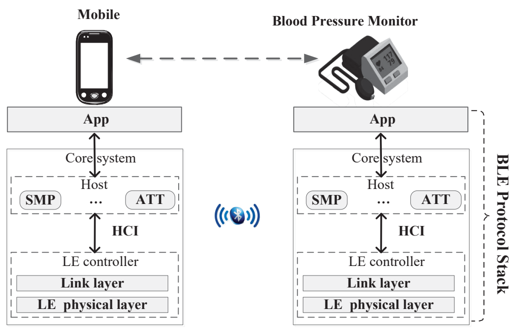

Figure 1: BLE protocol stack

establish the connection, if no device explicitly requests pairing, the communication continues in plaintext. Otherwise, pairing is started to negotiate keys and encrypt the communication. Steps 5 to 9 in Figure 2 illustrate a typical pairing process. Afterwards, the two devices start to exchange data and communicate via the ATT protocol as demonstrated in Step 10 and 11. We will present the workflow in detail between a mobile and a blood pressure monitor as follows.

2.2.1 Connection Stage

In Step 1, when the blood pressure monitor tries to establish a connection, it first broadcasts advertising packets, indicating its availability. In Step 2, when the mobile app is launched, it receives the advertisements and then sends a scan request to the monitor. In Step 3, the blood pressure monitor responds with a scan response packet. During this connection stage, the mobile app uses advertising and scanning to collect information about the blood pressure monitor such as the monitor's name, MAC address, and primary services. In Step 4, the mobile initiates a connection with the blood pressure monitor of interest. Here the mobile device is called the master/initiator for its role of initiating the connection. The peer BLE device such as the blood pressure monitor is called the slave/responder.

2.2.2 Pairing Stage

A mobile app and the system `Settings` app on most OSes such as Android can initiate a pairing process through SMP shown in Figure 1. As a slave device, the blood pressure monitor may send a security request and ask the mobile device (i.e., the master) to initiate the pairing process, which can be divided into the following three phases.

Phase 1 – Pairing feature exchange In Step 5, the two devices announce their pairing features as follows to negotiate a common association method. 1. *Authentication requirements* – Authentication requirements include *bonding* and *MITM protection*. Bonding means that the keys generated during the pairing process will be saved for later use to reduce delay caused by a future pairing process. MITM protection indicates the preference of defense against MITM attacks. If two devices explicitly set *MITM protection* as false, Just Works is selected as the association method. If one device sets *MITM protection* as true, then there will be two potential cases: (*i*) Passkey Entry or Numeric Comparison is chosen if I/O capabilities of both devices support the association method; or (*ii*) Just Works is used. 2. *I/O capabilities* – The exchanged I/O capabilities determine a association method with authentication requirements. Different association methods require different I/O capabilities as introduced in §2.3.1. 3. *BLE version* – BLE version is indicated in the *Secure Connections (SC)* bit. If the mobile and peer device set the SC bit, BLE 4.2 and above will be adopted. Otherwise, the BLE legacy pairing protocol is used.

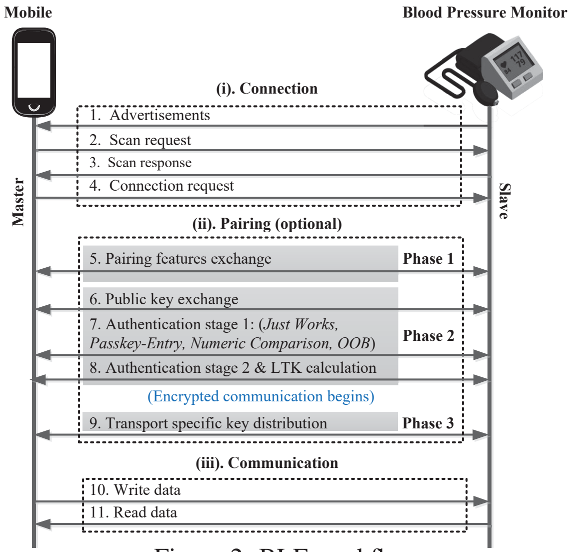

Figure 2: BLE workflow

Phase 2 – Key exchange and authentication This phase includes three steps (Steps 6, 7, and 8) as follows. 1. *Public key exchange*: In Step 6, the master and slave use the Elliptic-Curve Diffie–Hellman (ECDH) key exchange protocol to obtain each other's public key and generate a symmetric key, known as the Diffie–Hellman Key (DHKey). 2. *Authentication stage 1*: In Step 7, depending on the exchanged I/O capabilities and authentication requirements of the two devices, one of the following four association methods is adopted, including "Just Works", "Passkey Entry", "Numeric Comparison" and "Out of Band (OOB)". 3. *Authentication stage 2 and LTK calculation*: In Step 8, the two pairing devices use previously exchanged authentication information including DHKey to generate MacKey and Long Term Key (LTK). MacKey is used to ensure both devices generate the same LTK. If the pairing feature *bonding* is required, LTK is saved for future *SessionKey* generation and link encryption.

Phase 3 – Transport specific key distribution In Step 9, the communication after Phase 2 will be encrypted with a *SessionKey* generated from LTK. In this phase, the Identity Resolution Key (IRK) may be distributed from one device (either the master or the slave) to the other and is used for privacy preserving.

2.2.3 Communication Stage

The ATT protocol is a server/client protocol with the slave as the server and the master as the client. For example, the app on the mobile device is a client and the blood pressure monitor is a server in Figure 2. A server maintains services in the format of attributes. The client accesses the values of attributes from the server. An attribute has four properties: an attribute handle, a universally unique identifier (UUID), a value, and a set of permissions. To access an attribute at the server in Steps 10 and 11, a client can issue a read/write request to the server with the attribute handle, which uniquely identifies the attribute. The UUID refers to the data type. The permission protects attributes on a device and specifies the security levels required to access attributes.

2.3 BLE Security and Privacy

2.3.1 Association Methods in BLE

Passkey Entry: During the pairing process, one device such as a mobile needs to display a 6-digit pin, and the user inputs the pin on the other device using a keypad/keyboard. The authentication stage 1 (i.e., Step 7) in Figure 2 fails if the attacker does not know the pin.

Numeric Comparison: This association method is applicable when both devices have displays and confirmation buttons. A function converts the exchanged public keys and nonces into a six-digit number. Each device displays the number [1] and the user confirms that these two displayed numbers match by pressing a "Yes" button on each device to proceed the pairing process. The fact that both displayed numbers are the same ensures that the exchanged two pubic keys are from the two intended pairing devices.

Out of Band (OOB): In *OOB*, a secret is shared through an out-of-band venue such as near-field communication (NFC) and the LTK is derived from this secret. If the OOB venue is secure, the MITM attack can be defeated.

Just Works: It is designed for devices without I/O capabilities [1] and is unfortunately subject to MITM attacks. Just Works has almost the same pairing process as Numeric Comparison except that the generated number is not displayed and the user is not involved to ensure the exchanged pubic keys are the same.

2.3.2 Attribute Permission

The client (master) may access the attributes at the server (slave). The permission specifies the security level required to access attributes and may be read/write, encrypted read-/write, authenticated read/write, or authorized read/write. Authorized read/write is unspecified in the BLE specification yet while the first three security levels correspond to the adopted association methods. Different association methods result in different types of keys, and a specific type of key may have access to an attribute with a particular permission.

Specifically, BLE defines two types of keys: unauthenticated-and-no-MITM-protection keys corresponding to Just Works and authenticated-and-MITM-protection keys corresponding to Passkey Entry, Numeric Comparison and OOB. A read/write attribute can be accessed with no restriction. An encrypted read/write attribute can be accessed with an unauthenticated-and-no-MITM-protection key or authenticated-and-MITM-protection key. An authenticated read/write attribute can only be accessed when the link is encrypted with an authenticated-and-MITM-protection key. If the attribute such as the keyboard input is sensitive, a high security level like authenticated read/write shall be used so that secure pairing protocols are required to counter eavesdropping and MITM attacks, and prevent keystroke leaking. We find that the permission is often misused in practice, causes security issues, and will discuss the misuse in §4.4.2.

2.3.3 Identity Privacy

Identity Resolving Key (IRK) shall be shared during pairing for device identity privacy. A BLE device such as a mobile can be tracked if its MAC address is used in advertisement and later communication. BLE addresses this privacy issue by IRK and a suite of protocols. In particular, IRK is used to generate resolvable private addresses in advertisement and communication. Only a device with privacy requirements needs to distribute its IRK and real MAC address to its peer device. For example, if a mobile needs to protect its MAC address, it distributes its IRK and real MAC address to its peer device first. Then, the mobile uses this IRK to generate a resolvable private address for its packets and the peer device uses the mobile's IRK to resolve the private address. If the mobile's peer device needs to protect its MAC address, it sends its own IRK and MAC address to the mobile for private address generation and resolution although this practice is rare.

2.4 BLE Profiles

A Bluetooth profile specifies functionalities and features of all layers in Figure 1 for a particular class of applications. For example, the Human Interface Device Profile (HID) defines rules that allow a HID device, such as a keyboard, to accept inputs from humans and shows the output to humans through Bluetooth. A profile may contain other profiles and protocols as its building blocks. The Generic Access Profile (GAP) defines the basic requirements of a Bluetooth device and all Bluetooth devices implement GAP. For example, GAP performs advertising and scanning.

A smart device can implement the Generic Attribute Profile (GATT), which is built upon the ATT protocol, to exchange arbitrary data in the format of attributes with its peer devices. GATT organizes attributes into services. A service contains zero or more characteristics, which are also attributes and user data containers. A characteristic contains zero or more descriptors, which provide more metadata. A primary service provides the primary functionality of the device. A secondary service can work as a building block and should be included in the primary service.

3 SCO Mode Design Flaws

In this section, we first discuss specification deficiency and introduce four key capabilities required to support the SCO mode at initiators such as mobile devices. Next, we show the design flaws in the Android BLE programming framework due to the lack of these capabilities while similar issues in other OSes are presented in §7.

3.1 Specification Deficiency

For a slave device such as a blood pressure monitor in Figure 2 that provides services, the BLE specification defines the SCO mode. This mode provides the highest BLE security level (Mode 1, Level 4 [8]), in which only the three secure association methods, Passkey Entry, Numeric Comparison and secure OOB, can be used and the BLE Legacy is not allowed. In this mode, if secure pairing is not used, the device shall send *Pairing Failed* packets with the error code "Authentication Requirements". According to Page 373, Vol 3, Part C of the BLE specification [8], when a device is in the SCO mode, "*The device shall only accept new outgoing and incoming service level connections for services that require Security Mode 1, Level 4 when the remote device supports LE Secure Connections and authenticated pairing is used.*", where the service level connection refers to the application layer connection.

It can be observed that although BLE specifies the SCO mode for a slave that provides services, it does not explicitly define (or require) the SCO mode for a master, which is also the airing initiator such as the mobile in Figure 2. Unfortunately, without such a requirement at the initiator, an attacker can spoof a victim BLE device (e.g., using a fake blood pressure monitor) and connect to the initiator to launch various attacks as shown in this paper.

In our analysis, we find that the following four stages are critical to implement the SCO mode at the initiator, which includes initiation, status management, error handling, and bond management. Correspondingly, we propose four required capabilities at the initiator as follows:

- **Initiation** – A mobile application/app shall have the capability of instructing the OS, i.e., the BLE stack, a secure association method to enforce.
- **Status management** – The OS shall memorize the specified secure association method, enforce it at the right time and notify the application of the result. The right time is between Step 5 and Step 6 in Figure 2 when the peer device sends its I/O capabilities and the initiator determines the association method correspondingly.
- **Error handling** – When errors happen during communication, the OS and application shall coordinate to handle these errors and enforce the specified secure association method.
- **Bond management** – The app shall have the capability of removing a broken bond caused by errors in order to initiate the enforcement process again.

Pairing stage	Design flaws
Initiation	**Flaw 1** – No mechanism to specify a association method
Status management	**Flaw 2** – No mechanism to enforce a specified association method or for an app to obtain the negotiated association method in time
Error handling	**Flaw 3** – No mechanism for an app to handle errors while the BLE stack mishandles pairing errors
Bond management	**Flaw 4** – No mechanism to programmatically remove a suspicious/broken bond and start re-pairing.

Table 1: Design flaws that an OS may have

Table 1 lists four design flaws that an OS may have corresponding to the four capabilities.

3.2 Design Flaws in Android

We now show how the BLE specification shortcoming leads to security issues in Android. We focus on Android because of its prevalence and rich set of BLE applications. We later also show that security issues in Android endanger peer BLE devices in §6.3 and similar issues exist in non-Android OSes in §7. Android has all the four design flaws in Table 1 as follows.

Flaw 1 – No mechanism to specify a association method. The function `createBond()` in Listing 1 is the only function an Android app can use to start a pairing process with a peer BLE device. It does not accept any input parameter and the app cannot specify any particular association method even if it knows its peer BLE device's I/O capabilities. The return value of this function, `true` or `false`, indicates if the pairing process has been successfully started. `createBond()` also checks if the mobile device has an LTK in the device. If yes, `createBond()` returns `false` and will not re-pair with the peer device since the mobile device was paired with the device. In addition, `createBond()` is an asynchronous call and does not wait for the pairing process to complete.

```
boolean createBond() {
    ...
    DeviceProperties deviceProp = mRemoteDevices.
     getDeviceProperties(device);
    //if already paired, return false
    if (deviceProp != null && deviceProp.
     getBondState() != BluetoothDevice.BOND_NONE) {
        return false;
    }
    ...
    //put a create bond message into the message
     processing queue
    Message msg = obtainMessage(BondStateMachine.
     CREATE_BOND);
    sendMessage(msg);
    return true;
}
```

Listing 1: The function *createBond()* (Android 9.0)

Flaw 2 – No mechanism to enforce a specified association method or for an app to obtain the negotiated association method in time. From source code, we find Android only relies on exchanged I/O features to determine the association method. An app may use the following asynchronous mechanisms to obtain the status of a pairing process after pairing is completed. Through the intent `ACTION_BOND_STATE_CHANGED`, the app knows pairing status including pairing in progress (`BOND_BONDING`), pairing failure (`BOND_NONE`), or pairing succeeded (`BOND_BONDED`). Through the intent `ACTION_PAIRING_REQUEST`, the app knows either Passkey Entry or Numeric Comparison is adopted. By registering both intents `ACTION_BOND_STATE_CHANGED` and `ACTION_PAIRING_REQUEST`, an app knows the adopted association method, Passkey Entry, Numeric Comparison, Just Works or plaintext communication only after the pairing process is completed. Therefore, an app cannot use Listing 2 to enforce a specified association method in time. This flaw can be exploited to steal a mobile's MAC address and IRK, as shown in §4.3.

Flaw 3 – No mechanism for an app to handle errors while the BLE stack mishandles pairing errors. The Android BLE programming framework does not memorize a negotiated association method. Further, Android does not provide APIs for apps to properly process pairing errors. Pairing errors of interest are introduced below.

"`Pin or Key Missing (0x06)`": When an Android mobile and its peer BLE device are paired, their communication link is encrypted with the negotiated keys including the LTK. If a peer BLE device's LTK is intentionally removed, the device will send an error code "`Pin or Key Missing (0x06)`" to the mobile. However, the Android mobile does not notify the user of this error. Instead, it automatically communicates with the peer device in plaintext. Moreover, there are no APIs or mechanisms for an Android App to detect the `0x06` error. An app cannot use the Android reflection technique [9] to call a system level function `isEncrypted()` and check if the communication is in plaintext since it is prohibited [10]. We also find when this error occurs, Android does not remove the corresponding LTK. It should have removed the LTK since the communication is in plaintext and the LTK is supposed to encrypt the communication.

"`Insufficient Authentication (0x05)`" or "`Insufficient Encryption (0x0f)`": When an initiator tries to access an attribute with the "`encrypted read/write`" or "`authenticated read/write`" permission at its peer device, if the link is not encrypted, the peer device may send either an "`Insufficient Authentication (0x05)`" or "`Insufficient Encryption (0x0f)`" error code. If the attribute's permission is "`authenticated read/write`" and the link is only encrypted with an unauthenticated-and-no-MITM-protection key as introduced in §2.3.2, the peer device sends the `0x05` error code. When an Android mobile's Bluetooth service receives either `0x05` or `0x0f` error code, it automatically starts re-pairing, ignoring the previously adopted association method. Although the app can learn if the `0x05` or `0x0f` error occurs via a callback function `onCharacteristicRead()`, the app cannot stop the re-pairing process in this callback function. Therefore, an attacker may spoof a paired device, utilize this error to start a pairing process with an Android mobile, and obtain the Android mobile's MAC address and IRK.

```
boolean numericcomparison=false;
boolean passkey=false;
boolean justworks=false;
boolean plaintext=true;
// Activity starts; register intents
public void OnCreate(){
  IntentFilter pairingRequestFilter = new
     IntentFilter();
  pairingRequestFilter.addAction(BluetoothDevice.
     ACTION_BOND_STATE_CHANGED);
  pairingRequestFilter.addAction(BluetoothDevice.
     ACTION_PAIRING_REQUEST);
  registerReceiver(mPairingRequestRecevier,
     pairingRequestFilter);
  }
 //Once connected call createBond()
  device.createBond();
 //Process intents and determine association method
public void onReceive(Context context, Intent intent
     ) {
  if (BluetoothDevice.ACTION_PAIRING_REQUEST.equals(
     intent.getAction())){ // either numeric
     comparison or passkey is used
    int pairingtype = intent.getIntExtra(
     BluetoothDevice.EXTRA_PAIRING_VARIANT,
     BluetoothDevice.ERROR);
    if(pairingtype==BluetoothDevice.
     PAIRING_VARIANT_PASSKEY_CONFIRMATION){
      numericcomparison=true;
      plaintext=false;
    }
    if(pairingtype==BluetoothDevice.
     PAIRING_VARIANT_PIN){
      Passkey=true;
      plaintext=false;
    }
  }
  if (BluetoothDevice.ACTION_BOND_STATE_CHANGED.
     equals(intent.getAction())) { // Bonding,
     bonded, or bonding none (failure)?
    int bondstate = intent.getIntExtra(
     BluetoothDevice.EXTRA_BOND_STATE,
     BluetoothDevice.ERROR);
    if(bondstate==BluetoothDevice.BOND_BONDED){
    if(!numericcomparison && !passkey){
      justworks=true;
      plaintext=false;
    }
  }
 }
}
```

Listing 2: Android determining association method after bonding

Flaw 4 – No mechanism to programmatically remove a suspicious/broken bond and start re-pairing. A third-party Android app cannot remove a bond from the mobile's list of bonded devices although the user can manually remove a bond with the system settings app. The app cannot use the prohibited reflection technique to call the system level API `removeBond()` and delete an LTK, i.e., a bond. Even if the app is able to tear down an insecure connection that uses Just Works, breaking the connection does not remove the bond. The app cannot start a new secure pairing process with a bonded device using `createBond()` either since the LTK/bond still exists.

4 Downgrade Attacks

In this section, we present the threat model, attack overview, and detailed downgrade attacks against Android mobiles and ensuing attacks against their peer devices.

4.1 Threat Model

Threat model for Android mobiles. Our attacks against Android mobiles take the following assumptions. (*i*) An attacker can obtain the same type of victim devices to explore the applications and communication protocols. (*ii*) The attacker cannot physically access the mobile. (*iii*) Our attacks do not need malicious apps installed on the mobile while many other attacks require malicious apps for Bluetooth exploits [11–13]. (*iv*) Before the attack, the Android mobile and its peer device are paired using secure association methods such as Passkey Entry and Numerical Comparison. This assumption presents a more reasonable and harder scenario for attackers. Note that all attacks introduced in this paper can also be deployed if the Android and its peer device have not paired or paired with Just Works.

Threat model for peer devices. The threat model for the attacks against peer BLE devices is different from the threat model for attacks against mobiles, and it has following assumptions: (*i*) Before the attack, the Android mobile and its peer device are paired using secure association methods. This assumption is the same as the one for attacks against mobiles. (*ii*) We also assume that the attacker cannot touch or unlock victim mobiles, but the attacker may have physical access to BLE devices, which could be true in various scenarios. For example, IoT products such as smart lights may be placed outside a house. Few people physically lock away their BLE keyboards and attackers may press keys of those BLE keyboards. Regardless, we consider the following two attack scenarios against peer BLE devices of mobiles: (*a*) The attacker can physically access victim BLE devices briefly, for example, for a few minutes or even seconds; (*b*) The attacker cannot physically access the BLE device. Our defense in §5 will defeat attacks even if the attacker can physically access victim BLE devices.

4.2 Attack Overview

Our attacks against mobiles involve four adversarial parties: sniffer, fake BLE device, fake mobile, and blocker. The sniffer sniffs BLE communication and collects basic information such as the device's MAC address and name from advertising packets and scan response packets. The fake BLE device and fake mobile are full-fledged BLE devices and also called the spoofing device and spoofing mobile. A fake device emulates a victim device. The attacker uses a sniffer to obtain the MAC address and name of a BLE device. A fake device is then configured to have the same MAC address and name as the victim BLE device. It can forge advertising and scan response packets that contain the same device name and service description as those of the victim device. The fake device can implement the same attributes of the victim device and manipulate the permissions of these attributes. A fake mobile emulates a victim mobile. This requires that the fake mobile know the victim mobile's MAC address and IRK which is proved possible and will be demonstrated later in this section.

A blocker can launch a Denial of Service (DoS) attack and block a victim BLE device from connecting to a victim mobile so that a fake/spoofing device can connect to the victim mobile. The blocker can be implemented as follows. (*i*) A blocker can be a customized initiator. The number of connections to a victim device is often limited to one. Therefore, when a blocker connects to the victim BLE device, other mobiles cannot connect to the victim device any more. If the victim device allows multiple connections, multiple blockers can be used [1]. (*ii*) If a whitelist is used by the victim device, a blocker may fail to connect to it, since the victim device only accepts an initiator that has paired with it before. To subvert such a defense, a fake/spoofing BLE device can increase its advertising frequency and will have a better chance connecting to the victim mobile than the victim device with the same MAC address. Our experiments in §6.3 have validated this approach. (*iii*) A jammer can also work as a blocker although we do not use it in this paper.

The four adversarial parties collaborate to deploy attacks against victim mobiles and peer devices as shown in Figure 3. For example, to attack a victim mobile, a blocker can be used to block a victim device so that a fake device can connect to the victim mobile. The fake device can then manipulate the BLE protocol such as device I/O capabilities and intentionally create errors to poke the mobile. With the stolen IRK and MAC address of the victim mobile through attacks against mobiles, the fake mobile can connect to the victim device, which can work with the fake device to perform attacks such as MITM attacks.

4.3 Attacks against Android Mobiles

Figure 3 gives steps of each attack and the relationship between different attacks. One attack can be a building block of other attacks. The name of an attack indicates its goal.

Attack I – False data injection via Design Flaw 3. The fake device intentionally creates an error code `Pin or Key Missing (0x06)`. The communication between the Android mobile and the fake device is downgraded to plaintext as discussed in Design Flaw 3. We configure the permission of the attributes of the fake device as read/write so that access to the attributes does not require any pairing. The fake device can then inject false data to the mobile. This attack cannot be easily detected since the Android mobile does not delete the original LTK. Therefore, even if the user checks the list of bonded devices at the Android mobile's system settings, the list will not show any aberrations.

Attack II – Spoofing attack on sensitive information via Design Flaw 3. By using Design Flaw 3, the attacker downgrades the communication between the fake device and the Android mobile to plaintext. The fake device is positioned to receive any sensitive information from the Android mobile. We find that many IoT applications implement an application layer password mechanism for user authentication. When a user inputs the password, the fake device can collect this password.

Attack III – Stealing Android mobile's IRK and MAC address via Design Flaws 1, 2 and 3. To prevent the MAC address from leakage, an Android mobile with API 23 or above uses IRK by default [14]. According to our experiments, the IRK is generated when the mobile is configured for the first time starting from the factory settings. It will not change until the mobile is reset to the factory settings. Any peer BLE device paired with the mobile will receive the same IRK and MAC address of the mobile.

To obtain the IRK and MAC address of a victim Android mobile, the fake device can intentionally create a "`Pin or Key Missing (0x06)`" error so that the communication between the mobile and fake device is downgraded to plaintext. The attacker also configures the attribute permission of the fake device as "`encrypted read/write`". When the Android app tries to access these attributes, the fake device sends an "`Insufficient Authentication (0x05)`" or "`Insufficient Encryption (0x0f)`" error to the victim mobile, which starts a re-pairing process according to Design Flaw 3. The fake device is configured to have no I/O capabilities so that the victim mobile and fake device pair with Just Works because of Design Flaws 1 and 2. The mobile then distributes the IRK and MAC address to the fake device in Step 9 in Figure 2. With the IRK, the attacker can perform the private address resolution and trace the identity of the Mobile every time the mobile uses BLE. This attack defeats the purpose of IRK, which is used to prevent an Android mobile from being tracked.

Attack IV – Denial of Service (DoS) via Design Flaws 1, 2, 3 and 4. To perform Attack IV, the attacker first performs Attack III stealing the mobile's MAC and IRK, in which an attacker can pair a fake device with a victim Android mobile using Just Works. This pairing process creates a new LTK for the mobile. The attacker then turns off the fake device and blocker. The victim mobile will try to communicate with the victim device. However, since the LTK on the mobile and the LTK on the victim device are now different, we find that Android cannot detect the inconsistency and the communication enters into a deadlock. However, as mentioned in Design Flaw 4, there is no public API for an app to remove a bond on the mobile. The app cannot remove the bond or start re-pairing. The deadlock can only be resolved by manually removing the bond in the Android system setting.

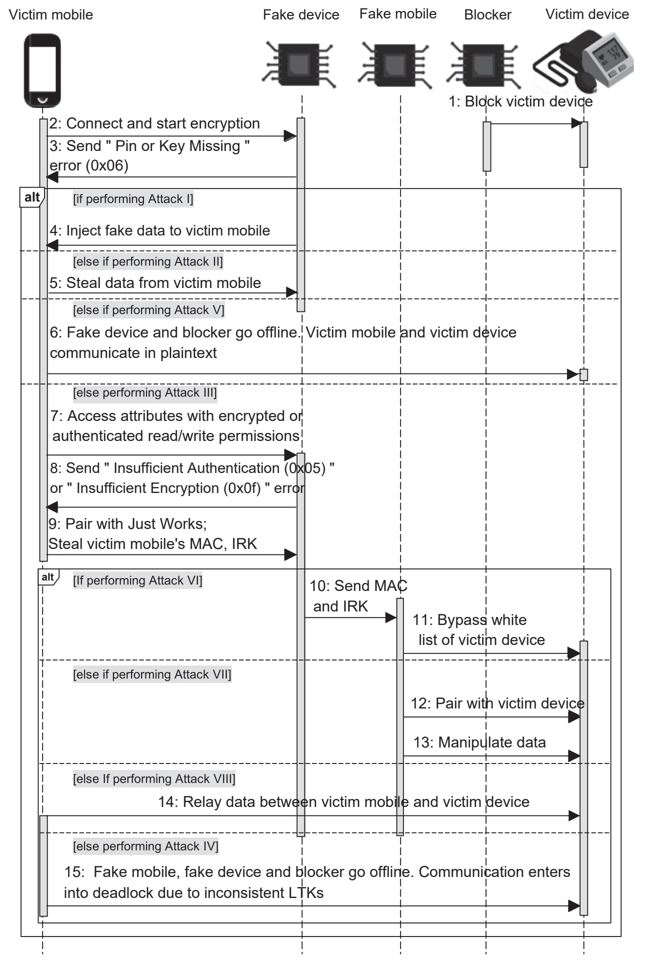
Victim mobile
Fake device
Fake mobile
Blocker
Victim device
1: Block victim device
2: Connect and start encryption
3: Send " Pin or Key Missing " error (0x06)
alt
[if performing Attack I]
4: Inject fake data to victim mobile
[else if performing Attack II]
5: Steal data from victim mobile
[else if performing Attack V]
6: Fake device and blocker go offline. Victim mobile and victim device communicate in plaintext
[else performing Attack III]
7: Access attributes with encrypted or authenticated read/write permissions
8: Send " Insufficient Authentication (0x05) " or " Insufficient Encryption (0x0f) " error
9: Pair with Just Works; Steal victim mobile's MAC, IRK
alt
[If performing Attack VI]
10: Send MAC and IRK
11: Bypass white list of victim device
[else if performing Attack VII]
12: Pair with victim device
13: Manipulate data
[else If performing Attack VIII]
14: Relay data between victim mobile and victim device
[else performing Attack IV]
15: Fake mobile, fake device and blocker go offline. Communication enters into deadlock due to inconsistent LTKs

Figure 3: Sequence diagram of downgrade attacks w/o physical access in Unified Modeling Language (UML). The *alt* frame is the alternative combined fragment, modeling the if-then-else logic. *Steps of Attacks against victim mobile*: (I) Fake data Injection Attack (1–4); (II) Sensitive Information Stealing Attack (1–3 and 5); (III) Stealing IRK and MAC Address Attack (1–3 and 7–9); (IV) DoS Attack (1–3, 7–9 and 15). *Steps of Attacks against victim device*: (V) Eavesdropping Attack (1–3 and 6); (VI) Whitelist Bypassing Attack (1–3 and 7–11); (VII) Data Manipulation Attack (1-3 and 7-13); (VIII) Man-in-the-Middle Attack; (1–3 and 7–14)

4.4 Attacks against Peer Devices

Attacks against an Android mobile will affect its bonded peer BLE device. We now discuss the attacks beyond mobiles, i.e., Attacks V–VIII in Figure 3. The fake mobile that obtains the victim mobile's MAC address and IRK can connect to the victim device and deploy different attacks under the two different threat models.

4.4.1 Attacks with Brief Access to Victim Device

Given that a mobile cannot enforce secure pairing, a fake device connects to the victim mobile using the scheme in Attack I (false data injection attack). Since an attacker can touch a victim peer device, the attacker can always pair a fake mobile with the victim device even if the victim device enforces the SCO mode. Now the fake mobile and fake device can launch an MITM attack.

4.4.2 Attacks without Access to Victim Device

In §2, we show that two secure measures can be adopted to protect sensitive data on a device, namely pairing and attribute permissions. While secure pairing protects the communication and attribute permissions limit access to attributes based on adopted association methods, we find that attribute permissions are often misused and the misused permissions will cause security issues.

Attack V – Passive eavesdropping attack. This attack works when the victim device has only *read/write* attributes. We assume that before the attack, the mobile pairs with the peer device that uses the SCO mode. To launch this attack, the attacker first blocks the victim device. A fake device then performs the "`Pin or Key Missing (0x06)`" error attack so that the communication between the fake device and the victim mobile is downgraded to plaintext. The fake device then goes offline and the blocker is turned off. We find that the victim mobile then communicates with the victim peer BLE device in plaintext and can access the peer device's *read/write* attributes. Since the communication is in plaintext, the attacker can eavesdrop on the communication and retrieve sensitive information using a sniffer. Similar to the false data injection attack, even if the user checks the bonded devices list at the mobile's system settings, no abnormalities will be observed.

Attack VI – Bypassing the whitelist. A BLE device may use a whitelist of MAC address and IRK, and allow connections only from already paired mobiles. Since an attacker can steal a victim mobile's MAC address and IRK, a fake mobile with the same MAC address and IRK can bypass the whitelist and connect to the victim peer BLE device. We will use this attack to bypass a keyboard's whitelist and perform further attacks.

Attack VII – Data manipulation. The fake mobile may attempt to access sensitive services once it connects to the victim device. If the permission of the attributes of the BLE device is encrypted read/write or authenticated read/write, the fake mobile has to pair with the peer device first. If the BLE device enforces the SCO mode or the attribute permission is authenticated read/write, the fake mobile has to perform secure pairing with the peer BLE device and may not be able to perform the attack. Recall that an authenticated read/write attribute requires secure pairing from the mobile.

Attack VIII – MITM attacks. If the data manipulation attack is possible on a peer device, the MITM attack can then be deployed. To this end, a fake device connects to the Android mobile using the fake data injection attack and a fake mobile sets up another connection with the peer device using the data manipulation attack. The fake device and the fake mobile can now communicate with each other, and work as the MITM to relay or manipulate the messages between the victim device and mobile.

5 Countermeasures

In this section, we address the design flaws discussed in §3 and present countermeasures to enforce secure pairing within Android. For compatibility, we implement the SCO mode as a configurable option for the BLE programming framework, allowing apps to defeat the presented attacks. If the option is not used, BLE on an Android mobile follows the current BLE specification to support legacy devices. We have implemented a prototype on Android 8 based on the Android Open Source Project (AOSP) [6]. Please note the issue of multiple apps (including malware) using the same peer BLE device with one connection has been addressed in co-located attacks [11–13]. Our defense measure still works if we ignore the danger of co-located attacks and allow multiple apps per connection. For example, all apps connecting to the same peer device shall follow our defense measure mechanism to enforce the SCO mode and deal with errors. Other implementations are possible too and will be up to the policy using the peer device. Our defense measure can also be directly applied in the scenario that one app may connect to multiple devices. The detailed discussion of dealing with these two cases is out of the scope of this work.

5.1 Overview

For a mission critical application, the app knows the peer device's I/O capabilities, which should support secure pairing. With the SCO mode enabled at the mobile, the user has to physically authenticate the BLE device. If the negotiated association method between the mobile and its peer device is not the specified one, the communication shall be rejected and a critical security warning shall be directed to the user.

The principle of the proposed defense measures is also applicable to system wide devices such as keyboards managed by a system settings app. The system settings app manages BLE profiles. A profile specifies aspects of a class of BLE devices. For example, keyboards follow the HID profile specification, which recommends association methods for keyboards as part of the specification. Therefore, a profile

Design flaw	Pairing stage	Defense
Flaw 1	Initiation	Specifying a secure association method
Flaw 2	Status management	Enforcing a specified association method and notifying the app of the association method in time
Flaw 3	Error handling	Allowing apps to handle errors; Enforcing specified association method through stack when errors occur
Flaw 4	Bond management	Removing suspicious bond and starting secure re-pairing.

Table 2: Enforcing Secure Pairing on Android

can be updated if the class of BLE devices requires the SCO mode and the systems setting app will also be updated to enforce the SCO mode.

Our solution will not affect user experience much as it takes effect only when there are errors caused by attacks. A mobile app using our proposed solution works no different than a traditional one when there are no errors or attacks. Prompting users under attacks is apparently very necessary and improves security. For apps that do not have security concerns, they can just communicate with no pairing, but in plaintext. In this case, the proposed solution will not prompt users and affect user experience. Our proposed solution has the flexibility of dealing with different use cases while those cases with no security are not the focus of this paper.

5.2 Enabling the SCO mode

Table 2 summarizes how we address the four design flaws listed in Table 1 in the four stages of a pairing process respectively. We present the detailed defense measures as follows.

Addressing Design Flaw 1: Specifying a secure association method. An Android mobile can enforce a secure association method after the mobile and peer device have determined the association method through the exchanged I/O capabilities between Step 5 and Step 6 in Figure 2. If the negotiated association method is not the specified one, Android should reject further actions and give the user a security warning. The Android BLE stack shall cache the specified secure association method in memory and save it in a configuration file on its nonvolatile storage if bonding is requested.

To address Design Flaw 1, an app can use our function `specifyPairing()` to store the specified association method in a configuration file `scm.conf` through the Java Native Interface (JNI). Our `specifyPairing()` is a system API. It can programmatically obtain the app's package name. File `scm.conf` is located in the system folder `/data/misc/bluedroid/` and stores the app's package name and metadata including the specified association method. An app cannot manipulate metadata of another app.

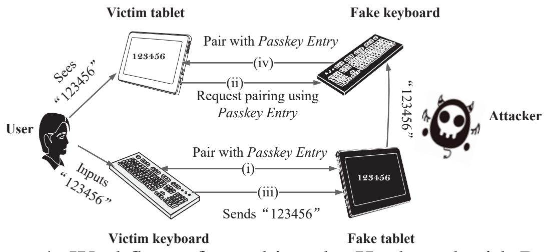

Figure 4: Workflow of attacking the Keyboard with Passkey Entry enforced

Addressing Design Flaw 2: Enforcing a specified association method; Notifying the app association method in time. When the pairing process starts, Android uses the system function `smp_proc_pair_cmd()` to exchange pairing features with the peer device. The bits in an integer `peer_io_caps` are used to indicate the peer device's I/O capabilities. Therefore, `smp_proc_pair_cmd()` can know the negotiated association method through announced I/O capabilities. In `smp_proc_pair_cmd()`, we read the configuration file `scm.conf` and obtain the app's specified association method. If the specified association method and negotiated association method do not match, `smp_proc_pair_cmd()` sends the error code `SMP_PAIR_AUTH_FAIL` to the peer BLE device, halts the pairing process, breaks the connection and sends warning messages to the user. Note that `smp_proc_pair_cmd()` can obtain the negotiated association method at the earliest possible time. This also addresses Design Flaw 2. An app knows its specified association method will be enforced. If it cannot be enforced, the user will receive a security warning.

Addressing Design Flaw 3: Allowing an app to handle errors; Enforcing the specified association method through the stack when errors occur. The "`Pin or Key Missing`" error occurs because the fake device does not have the LTK. The "`Insufficient Authentication/Encryption`" error occurs because the BLE connection does not have the permission to access the attributes on the fake device. Android does not notify the user these errors and starts a vulnerable association method. We address the design flaw as follows. If there is any such pairing related error, the Android BLE stack shall notify the user and ask the user whether to pair with the peer device. If the app has a specified association method in the configuration file and the user chooses to pair with the peer device, Android will enforce the specified association method, but give the user a security warning if it cannot be enforced.

Addressing Design Flaw 4: Removing a suspicious bond for secure re-pairing. An app shall be able to remove its own bonded devices whenever needed. We make the system API `removeBond()` available to applications. `removeBond()` is redesigned so that **an app can only remove its own bond**, not bonds of other apps. Therefore,

Association method	Brief physical access	
	Yes	No
Passkey Entry (Enforced)	✓	✗
Numerical Comparison (Enforced)	✗	✗

Table 3: Security of enforced secure association methods. ✓: vulnerable (e.g., to MITM attacks although not necessarily all attacks). Note: not meaningful to enforce Just Works.

a bond shall maintain metadata including the app's package name. `removeBond()` will obtain the calling app's package name and can remove only its own bond.

5.3 Security Analysis

We now discuss BLE pairing security if Android addresses the design flaws and enforces secure pairing, and the peer BLE device also enforces secure pairing. Under the assumption that an attacker cannot physically access the mobile or peer BLE device, the attacks in §4 will fail since secure pairing requires the attacker (operating the fake mobile and fake device) to see and work on the victim device and mobile.

Unfortunately, when an attacker can physically touch a BLE keyboard that uses the Passkey Entry association method, even if both the keyboard and mobile enforce Passkey Entry, the attacker can still perform the MITM attack as follows. Passkey Entry is secure only if the attacker cannot obtain the passkey. However, the BLE keyboard is a human input device, which sends keystrokes to a mobile device as long as the mobile device is paired with the keyboards. As shown in Figure 4, (i) if the attacker has brief physical access to the keyboard, the attacker can pair a fake mobile with the keyboard by entering a chosen passkey when the user is away from the device. (ii) The fake keyboard later pretends to be the real one and starts a pairing process with the victim mobile. The victim mobile enforces Passkey Entry and requires the user to enter a passkey displayed on the victim mobile. (iii) However, when the user enters the passkey on the victim keyboard, the fake mobile receives the user entered passkey. (iv) The fake mobile then sends the passkey to the fake keyboard, which can then use the passkey to connect to the victim mobile. The attacker can now perform the MITM attack.

The MITM attack above will fail when the victim mobile and keyboard enforce the Numeric Comparison association method even under the assumption that the attacker can physically access the keyboard. To implement Numerical Comparison, the keyboard must have a display. The attacker's fake mobile can still be paired with the victim keyboard because of the assumption of physical access. However, when the user pairs the victim keyboard with the victim mobile, the user has to compare the two numbers displayed on the victim keyboard and the victim mobile. With the underlying Numerical Comparison protocol, if the attacker performs the MITM attack with a fake mobile and a fake keyboard in the middle, the two numbers on the victim keyboard and the victim mobile will be different. The MITM attack will be detected and fail.

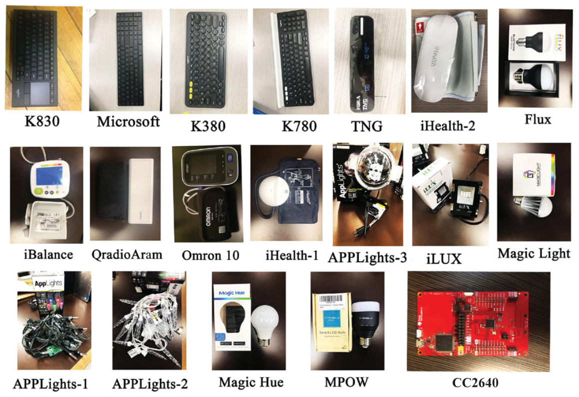

Figure 5: The Tested BLE devices

Based on the analysis above, it can be observed that under the assumption that the attacker can physically access the keyboard, Numerical Comparison is more secure than Passkey Entry. When we enforce secure pairing, Numerical Comparison provides the strongest pairing security. The BLE specification treats Passkey Entry and Numeric Comparison equally and these two secure association methods have the same security level - authenticated-and-MITM-protection. In the specification, if either of the two protocols is applied, the connection is considered as authenticated. This term is not accurate based on our analyses. Table 3 summarizes the security of enforced association method.

6 Evaluation

In this section, we first present experiment setup, and then evaluate the presented attacks and countermeasures.

6.1 Experiment Setup

We use Adafruit Bluefruit LE Sniffer [15] to sniff BLE communication and collect basic information such as a device's MAC address and name from advertising packets and scan response packets. We use Texas Instruments (TI) CC2640 [16] development boards to emulate the blocker, fake BLE device, and fake mobile.

To measure the generality of our attacks against different mobile devices and apps (§6.2), we used five mobiles from mainstream Android versions from 7.0 to 9.0 as listed in Table 4 in our experiment, along with 18,929 Android BLE apps, which were also used in [12], from the Androzoo database [17]. The cumulative user installation of these BLE apps including those in categories of health & fitness, business, medical and finance reaches about 9 billions [12]. To evaluate the attacks beyond the mobile devices (§6.3), we selected 18 popular commercial BLE products and three CC2640 development boards, which are presented in Figure 5 from various vendors to demonstrate our findings.

6.2 Attacks against Mobiles

Generality of the attacks against different Android mobiles. We tested all design flaws on mainstream Android versions, from 7.0 to 9.0 as shown Table 4 and find that all our attacks work with no adjustments. Recall that a fake device may use the "`Insufficient Authentication (0x05)`" error or "`Insufficient Encryption (0x0f)`" error in §3 to stealthily pair with the victim mobile through Just Works. This approach works under all versions of Android we tested. On Android 7.0, a fake device can also send a security request to stealthily pair with the victim mobile while the security request on higher versions of Android will raise a pairing request dialog window asking the user for permission. Such a dialog Window may alert the user.

Brand	Version
Samsung Galaxy S8+	Samsung Official Android 7.0
Google Pixel 2	AOSP Android 8.0
Samsung Tablet	Samsung Official Android 8.1
Samsung Note 8	Samsung Official Android 8.1
Google Pixel 2	AOSP Android 9.0

Table 4: Tested Android mobiles

Generality of the attacks against BLE apps. In §3, we show that the Android BLE programming framework has four design flaws. Intuitively, all Android BLE apps using the framework are vulnerable to attacks presented in this paper. We also want to find if apps use any pairing intents (presented in Listing 2) to determine the association method after pairing, and thus detect the MAC address and IRK stealing attack for the purpose of intrusion detection. Recall Listing 2 cannot prevent the MAC address and IRK stealing attack and other attacks as discussed in Flaw 2 in §3.

We build a tool named *BLE pairing scanner (BLEPS)* based on *soot* [18] to statically enumerate functions used in an app, construct call graphs and then determine how the app performs pairing and uses intents. Table 5 shows among all the BLE apps, 6282 apps use pairing related functions and intents. 2581 apps use *createBond()* to explicitly start a pairing process. 6117 apps use *getBondState()* to determine if the mobile is bonded with the peer device before data transmission. 2005 apps use only the `ACTION_BOND_STATE_CHANGED` intent to check if the mobile is bonded with the intended device. 239 apps use both `ACTION_BOND_STATE_CHANGED` and `ACTION_PAIRING_REQUEST`. 152 out of the 239 apps use intents to determine if Passkey Entry or Numeric Comparison is used. These apps then automatically input a fixed passkey for Passkey Entry via `setPin()` or programmatically "click" the confirmation button via `setPairingConirmation()` when Numeric Comparison is used as the association method. These strategies make Passkey Entry and Numeric Comparison useless. 87 of the 239 apps register intents for

BLE apps	Quantity	Radio
All apps	18929	100%
Apps using pairing related functionalities/intents	6282	33.10%
Apps using `createBond()` for pairing	2581	13.60%
Apps using `getBondState()` for pairing status	6117	32.31%
Apps using `ACTION_BOND_STATE_CHANGED` intent for pairing status	2005	10.59%
Apps using intents for automatic pairing	152	0.80%
Apps using intents for debuging	87	0.45%
Apps using intents for intrusion detection	0	0

Table 5: BLE apps using pairing related functions and intents

Device Name	Type	Permission	Attacks							
			I	II	III	IV	V	VI	VII	VIII
APPLights-1	Light	read/write	✓	✓	✓	✓	✓	✗	✓	✓
APPLights-2	Light	read/write	✓	✓	✓	✓	✓	✗	✓	✓
APPLights-3	Light	read/write	✓	✓	✓	✓	✓	✗	✓	✓
Magic Hue	Light	read/write	✓	✓	✓	✓	✓	✗	✓	✓
Magc Light	Light	read/write	✓	✓	✓	✓	✓	✗	✓	✓
Flux	Light	read/write	✓	✓	✓	✓	✓	✗	✓	✓
NPoW	Light	read/write	✓	✓	✓	✓	✓	✗	✓	✓
iLux	Light	read/write	✓	✓	✓	✗	✓	✗	✓	✓
FORA TNG	Medical	read/write	✓	✓	✓	✓	✓	✗	✓	✓
iHealth-1	Medical	read/write	✓	✓	✓	✓	✓	✗	✓	✓
iHealth-2	Medical	read/write	✓	✓	✓	✓	✓	✗	✓	✓
iBalance	Medical	read/write	✓	✓	✓	✓	✓	✗	✓	✓
Omron 10	Medical	read/write	✓	✓	✓	✓	✓	✗	✓	✓
Qradio Aram	Medical	encrypted read/write	✓	✓	✓	✓	✗	✗	✓	✓
Logitech K830	Keyboard	encrypted read/write	✓	✓	✓	✓	✗	✓	✓	✓
Logitech K380	Keyboard	encrypted read/write	✓	✓	✓	✓	✗	✓	✓	✓
Logitech K780	Keyboard	encrypted read/write	✓	✓	✓	✓	✗	✓	✓	✓
Microsoft Designer	Keyboard	encrypted read/write	✓	✓	✓	✓	✗	✓	✓	✓
CC26XX with SCO mode enabled	Development board	authenticated read/write	✓	✓	✓	✓	✗	✗	✓	✓

Table 6: Attacks against commercial products. ✓ means vulnerable. ✗ means not vulnerable. All the attacks are launched without physical access to mobiles and peer devices.

debugging purposes by printing pairing status via `Log.d(.)`. We also perform manual analysis of these apps that involve both the two intents and find that none of the apps implements Listing 2 for intrusion detection.

Attacks against the mobiles and apps of the tested BLE devices. We have also successfully deployed Attacks I-IV in Figure 3 against mobiles installed with the apps of all the 18 commercial BLE products in Figure 5 and the results are presented in Table 6. Example attacks are presented as follows. (i) Attack I. We can deploy the downgrade attack and inject false measurements into the mobile app of iBalance Blood pressure monitors. (ii) Attack II. We can steal the APPLights app's passwords that are used for application layer user authentication. (iii) Attack III. A fake keyboard has the same MAC address and name as a victim Logitech 780 BLE keyboard and utilizes a higher advertising frequency so that it has a better chance to connect to a victim tablet than the victim keyboard. Here a blocker is not used to block the victim BLE keyboard because a BLE keyboard often implements a whitelist and accepts only a previously paired mobile. Once paired with the victim tablet, the fake keyboard can obtain the IRK and MAC address of the victim tablet. (iv) Attack IV. We are able to deploy the DoS attack against a smartphone installed with the smart light app from Flux so that the communication between the mobile and the real light fails since the mobile's LTK is manipulated and LTKs on the two sides are different. Note that the light from iLux does not support pairing at all. Therefore, an attacker can not create an LTK on the light, and the DoS attack fails.

6.3 Attacks beyond Mobiles

Attacks against BLE devices. Table 6 also shows the results of the attacks against the 18 commercial BLE products and the CC2640 development board. In particular, we have identified various vulnerabilities on the peer BLE devices, which may exist in other BLE products too: (i) Lack of SCO mode. All 18 commercial BLE devices do not enable the SCO mode, and an attacker can pair with these devices using Just Works without physical access. (ii) Misused permissions. 13 devices configure their attributes as read/write, and these attributes can be accessed without pairing. The current BLE Human Interface Device (HID) profile [19] does not enforce the SCO mode and requires only the encrypted (not authenticated) read/write permission for keyboard services. Therefore, the attacker may pair a fake tablet with a victim keyboard **remotely** using Just Works. Intuitively, all keyboards should be subject to our MITM attack given it is an HID specification flaw. (iii) Incorrect implementation of the SCO mode. Although TI's SDK allows an application to set an SCO mode flag, it only checks if the incoming pairing request enables the *Secure Connections (SC)* bit and does not check if the negotiated association method is Passkey Entry or Numerical Comparison. (iv) Incorrect implementation of attribute permission. An LTK can be an unauthenticated-and-no-MITM-protection key created by Just Works or an authenticated-and-MITM-protection key created by Passkey Entry, Numeric Comparison and OOB. Assume that a victim mobile has used secure pairing to pair with a victim BLE device based on TI chips and generated an authenticated-and-MITM-protection LTK. We find when a fake mobile with the victim mobile's MAC address uses Just Works and pairs with the victim device, TI's BLE stack does not update the key property, the generated LTK is still an authenticated-and-MITM-protection key, and the fake mobile can access attributes with the authenticated read/write permission. We have tested and proved the vulnerabilities on TI's CC2640, CC2640R2F, and CC2650, and reported the identified vulnerabilities to TI and a patched SDK was released recently [7].

We present example attacks beyond mobiles against the 18 commercial BLE products as follows. (i) Attack V. The passive eavesdropping attack requires the victim device have read/write attributes. It fails if the peer device has attributes of encrypted read/write or authenticated read/write as shown in Table 6. For example, with Attack V, an attacker can sniff blood pressure readings sent from an iBalance blood pressure monitor, breaching user privacy. (ii) Attack VI. The attack bypassing the whitelist works against BLE devices with a whitelist enabled such as the K780 keyboard. (iii) Attack VII. The data manipulation attack works against all BLE devices. For example, we can access and manipulate attributes with authenticated read/write permission of any device based upon TI CC26XX chips, even if the SCO mode is enabled. (iv) Attack VIII. The MITM attack works against all devices. For example, we have implemented the MITM attack against the k780 BLE keyboard and a tablet with two TI CC2640 development boards hosted in a case. One board works as a fake tablet connecting to the victim keyboard, and the other works as the fake BLE keyboard connecting to the victim tablet with the stolen IRK and MAC address.

Maximal attack distance Although BLE is designed for short-range communication, the attack distance against BLE devices depends on factors such as antenna gain and transmission power of involved devices. The attacker can use a large antenna to increase the attack distance. We use the CC2640R2F chips as the attacking fake devices and fake mobiles and find these off-the-shelf chips can achieve a reasonable long *maximal attack distance*, which is measured with a Bosch GLR825 laser distance measurer as the farthest distance at which the attacking device and target can be paired together. Figure 6 gives the cumulative distribution function (CDF) of the maximal attack distance against 20 different Android mobiles including Google Pixel 4, Samsung S10 and HUAWEI P30 Pro and the 18 devices in Figure 5. The maximal attack distance mean and maximum are 77.2 meters (m) and 94.0m against mobiles, and 46.5m and 77.1m against devices.

Keyboard connection competition As discussed earlier, when both a victim keyboard and a fake keyboard try to connect to a victim mobile, the one with a higher advertising frequency has a better chance. We now present the impact of the advertising frequency on the success rate of the fake keyboard connecting to the victim mobile. In our experiments, the victim keyboard is put close to an Android mobile as in a normal use scenario, while the fake keyboard is 10 meters away from the keyboard. For each advertising frequency, we perform the connection competition game 20 times. The success rate is the number of successful connections by our fake keyboard over 20. Figure 7 shows the success rate versus the advertising frequency. The success rate is 50% when the advertising frequency of the fake keyboard is 30HZ. The BLE specification sets the highest advertising frequency as 50 HZ, at which the success rate by the fake keyboard is 75%. We use CC2640 for the fake keyboard, which does not work when the advertising frequency is beyond 50HZ.

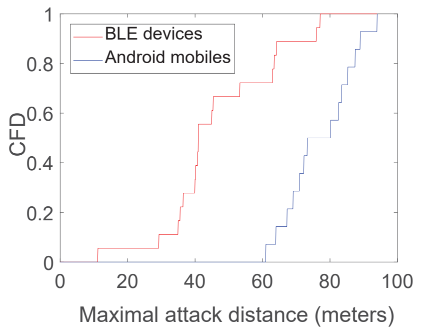

Figure 6: Maximal attack distance against mobiles and devices

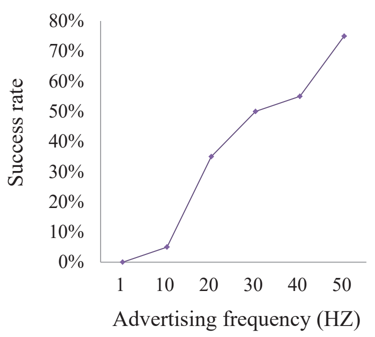

Figure 7: Success rate v.s. Advertising frequency

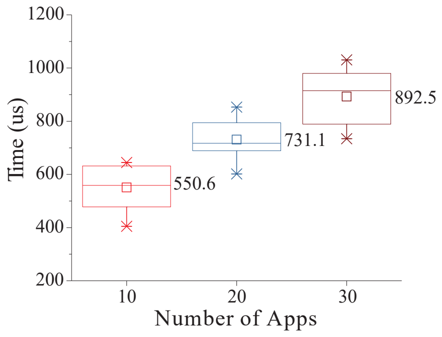

Figure 8: Defense performance

6.4 Countermeasure Evaluation

We have implemented our proposed defenses on a Google Pixel 2 mobile through the AOSP. We launched all our attacks and confirmed that they failed under the patched Android system. For example, in the case of the BLE keyboard, when Numerical Comparison is enforced, the user finds that the two numbers displayed on the victim mobile and keyboard (emulated by a CC2640) are different when the MITM attack is deployed. The user should reject the pairing and investigate the possibility of attacks.

We also evaluated the performance of our secure pairing, i.e., the overhead caused by the query of the configuration file `scm.conf` for a specific app's metadata such as the specified association method. We tested three cases: 10, 20 and 30 BLE apps using our defense mechanisms on the security enhanced Android mobile. The app of interest is always set as the last one in `scm.conf`. That is, we consider the worst case of time needed to find the metadata of the app of interest. We run the test for each case 10 times and derive the average time. Figure 8 shows the average delay is from 550.6μs to 892.5μs and is feasible for typical use of BLE apps in a mobile [20].

7 Flaws and Attacks in Other OSes

While we have demonstrated the flaws and attacks in Android systems, we also discover that these issues also exist in other major OSes including iOS, macOS, Windows, and Linux. This gives more evidence that no SCO mode at initiators is not an implementation issue but rather a BLE specification flaw. We argue all operating systems shall provide an option of enforcing the SCO mode in a similar way to the programming framework we have proposed for Android. In this section, we present the detailed discovery with these OSes.

In particular, Table 7 compares the design flaws and attacks against different OSes of latest releases and peer devices. Unless explicitly stated otherwise, we use the same threat model in §4.1: Before the attack, the initiators and their peer devices are paired using secure association methods. The attacker does not have physical access to either the initiator or responder. We summarize the differences between OSes regarding the four design flaws and attacks as follows: (i) A specific OS may not have all the four flaws. (ii) Some OSes such as Android may know the adopted association method after pairing, while others such as iOS does not know it at all. (iii) An OS may not have Flaw 3, but allows an app to handle errors. However, given that all OSes have Flaws 1 and 2, the app handles the errors in various vulnerable ways. (iv) Personal computer operating systems such as macOS, Windows and Linux do not use IRK by default as initiators, while a Linux device may programmatically adopt IRK [21]. Without the protection from IRK, an attacker may sniff a BLE connection, obtain the MAC address of a macOS/Windows/Linux device, and deploy attacks against peer devices as introduced in §4.4.2.

7.1 iOS and macOS

According to design guidelines for Apple devices [22] and our experiments, iOS and macOS use the same SDK *Core-Bluetooth* [23] to handle BLE communication. CoreBluetooth does not provide functions for BLE pairing although a function `IOBluetoothDevicePair.start()` is provided by the programming framework IOBluetooth for Bluetooth Classic on macOS [24]. In iOS and macOS, when an initiator tries to access an attribute that requires pairing at a peer device, the peer device sends error codes to the initiator, which then starts pairing exclusively through its BLE stack in the kernel. Therefore, Apple devices have four similar design flaws to Android: (i) Flaw 1. Apple devices can not specify a secure association method. (ii) Flaw 2. There is no mechanism to enforce a specified association method or for an app to obtain the negotiated association method. (iii) Flaw 3. There is no mechanism for an app to handle errors while the BLE stack mishandles pairing errors. An Apple app can learn whether *Insufficient Authentication/Encryption errors* occur by checking the *CBATTError* object, and the *Pin or Key Missing* error is not defined by the Apple programming framework. (iv) Flaw 4. There is no public

OS	Programming framework flaws				Attack against the Initiator				Attack against the peer device			
	Flaw 1	Flaw 2	Flaw 3	Flaw 4	Attack I	Attack II	Attack III	Attack IV	Attack V	Attack VI	Attack VII	Attack VIII
Android	✓	✓ˣ	✓	✓	Flaw 3	Flaw 3	Flaws1,2,3	Flaws 1,2,3,4	Flaws 3	Flaws 1,2,3	Flaws 1,2,3	Flaws 1,2,3
iOS	✓	✓	✓	✓	Flaw 3	Flaw 3	Flaws 1,2,3	Flaws 1,2,3,4	✗	Flaws 1,2,3	Flaws 1,2,3	Flaws 1,2,3
macOS	✓	✓	✓	✓	Flaw 3	Flaw 3	No IRK	Flaws 1,2,3,4	✗	These attacks can be deployed without stealing IRK;		
Windows	✓	✓	✗	✗	Flaw 1,2	Flaw 1,2	No IRK	✗	✗	These attacks can be deployed without stealing IRK;		
Linux	✓	✓ˣ	✗	✗	✗	✗	Flaws 1,2	✗	✗	Flaws 1,2	Flaws 1,2	Flaws 1,2

Table 7: Flaws and downgrade attacks across OSes with the assumption of no physical access to both initiators and peer devices. ✓ˣ means the app may know the adopted association method *after* pairing. * means the attack fails when the peer device enforces the SCO mode; ✓ means yes, ✗ means no, and their specific meaning should be clear in the context.

API for Apple devices to remove a bond or initiate a new pairing process even when errors occur.

Next, we discuss attacks against Apple devices as initiators of BLE communication. (i) The false data injection and spoofing attack for sensitive information are the same against iOS and macOS. When the "`Pin or Key Missing` (*0x06*)" error occurs, iOS and macOS do not notify the app, and will communicate with its peer device in plaintext. (ii) The downgrade attack stealing the IRK and MAC address works against iOS devices as follows. We find that an iOS device does not respond to the " `Insufficient Authentication` (*0x05*)" error, but initiates pairing if its peer device sends a security request and the "`Insufficient Encryption` (*0x0f*)" error code. Therefore, a fake device can utilize the *0x0f* error, pair with the victim iOS device using Just Works and steal its IRK and Mac address. Since macOS does not use IRK, an attacker can just sniff BLE communication to obtain the MAC address and spoof the initiator. (iii) The DoS attack can be deployed with no change on iOS and macOS because Apple devices can not resolve the issue of inconsistent LTK.

We now present attacks against the peer BLE device of an Apple device. Without physical access to the peer device of an iOS device, the attacker may use the whitelist bypass attack and deploy the data manipulation attack as introduced in §4.4.2). However, the passive eavesdropping attack does not work against iOS devices for the following reason. When a victim BLE device reconnects to an iOS device, the iOS device encrypts the connection rather than communicates with the peer device in plaintext as an Android device does. For macOS devices, since IRK is not used, the initiator can be easily spoofed and the spoofing device can then deploy all the attacks except passive eavesdropping.

7.2 Windows

Windows 10's SDK supports the Universal Windows Platform (UWP) [25], which provides a common platform for various devices, including laptops, desktops, and embedded devices including BLE devices. With UWP, the same source code can be compiled to run on different platforms. We find Windows 10 has the following flaws: (i) Flaw 1. UWP provide a function `PairAsync(ProtectionLevel)`, which has a parameter `ProtectionLevel` specifying the minimal protection level of a pairing process, including `None` (Plaintext), `Encryption` (Just Works), or `EncryptionAndAuthentication` (Secure association methods). However, according to our experiments on latest Windows 10, whatever the protection level is, Windows 10 always determines the association method based on the I/O capabilities of the peer device. The parameter `ProtectionLevel` does not work on personal computer Windows OS while it may be designed for particular embedded devices with a BLE stack supporting `ProtectionLevel`. (ii) Flaw 2. It can be observed from Flaw 1 that personal computer Windows OS cannot enforce secure pairing. There is also no mechanism for an application to obtain the negotiated association method. `PairAsync(.)` returns an instance of `DevicePairingResult`. DevicePairingResult has a member variable `ProtectionLevelUsed`, which shall return the pairing protection level. In our experiments, `ProtectionLevelUsed` is always set to `None`, no matter what association method is adopted. (iii) Windows does not have Flaw 3. When errors occur, Windows always tears down the connection and returns the error codes to the application through an instance of `ProtocolError`. The application can determine how to process the errors on its own.(iv) Windows has the function `UnpairAsync()` to remove a bond and does not have Flaw 4.

We now discuss possible attacks against a Windows 10 device as an initiator. Recall that to deploy attacks, a fake device shall spoof a victim device and connect to the victim initiator. Errors will occur as discussed in §3.2. Windows is different from Android. It tears down the connection and reports the error to the application. The application has two options: (i) removes the bond and initiates re-pairing with the fake device; (ii) does not respond to the error and stops working, i.e. halting. Since Windows can not enforce secure pairing, if the application chooses option (i), the communication is subject to false data injection and spoofing attack for sensitive information. If the application chooses option (ii), the communication is vulnerable to the DoS attack for the following reasons. First, since IRK is not used by Windows, an attacker can obtain the MAC address by sniffing and then spoof the initiator. The fake initiator pairs with the victim device and changes its LTK if the victim device does not enforce the SCO mode. Now the two LTKs on the victim initiator and victim device are different. The victim Windows initiator cannot pair with the victim device any more. Second, the LTK of the victim device can be lost due to device reset and the user will not be able to use the device because of the inconsistent LTK issue. Once the communication is stuck for the two reasons above, intuitively a user wants to continue his/her work, may manually remove the bonded device and initiate pairing again to move forward. The result of this practice is equivalent to option (i). This is why Table 7 only shows option (i). To attack the peer device of a Windows device, an attacker can easily implement the fake initiator since Windows does not use IRK. All the attacks in §4.4.2 can be deployed except eavesdropping, which does not work for the same reason under macOS.

7.3 Linux

Linux device uses BlueZ [21] as the Bluetooth stack for BLE communication. We use the official BlueZ programming framework, which is based on Python and C, to discuss its flaws: (i) Flaw 1. Linux devices can not specify a secure association method. For the purpose of pairing, an application can register a pairing agent via a Python function `RegisterAgent(agent, capability)`, where *agent* is an instance of pairing agent `org.bluez.Agent1` and *capability* is the I/O capability of the Linux initiator. Once registered, a Linux device calls a Python function `Pair()` to initiate pairing. Other than the default pairing agent, a customized agent written in C can also be programmed to handle the pairing process. Similar to other OSes, under Linux the association method is determined by the I/O capability of the peer device and the configured I/O capability of the initiator. Therefore, Linux devices are subject to Flaw 1. (ii) Flaw 2. There is no mechanism to enforce a specified association method or for an application to timely obtain the negotiated association method. If a fake device pairs with the victim Linux computer using Just Works, Linux may use a customized Linux pairing agent, modify the C function `bluez_agent_method_call(.)` and learn the adopted association method only after pairing while the default pairing agent does not provide this capability. (iii) Flaw 3. Since a Linux device tears down the connection and notifies the application when errors occur as Windows behaves, it does not have Flaw 3. (iv) Flaw 4. A Linux device is able to remove a bond via `RemoveDevice(.)` and has no Flaw 4.

We now present possible attacks against Linux devices. As discussed in §7.2, although an application has two options of processing errors, we argue they are equivalent. Therefore, when errors occur, the application chooses repairing. With the default pairing agent, a Linux initiator is subject to Attacks I, II and III. With a customized pairing agent, since the application can know the adopted association method after pairing, the application may tear down the connection if the association method is not the intended one to avoid false data injection and sensitive information stealing attacks. Since Linux cannot know *Just Works* timely, the IRK stealing attack still works if a Linux device employs IRK to prevent tracking for privacy.

To attack the peer device of a Linux device, an attacker can easily implement the fake initiator since Linux does not use IRK by default. If Linux programmatically adopts IRK, the attacker can use the IRK stealing attack to obtain the MAC address of the initiator. All the attacks in §4.4.2 can then be deployed except eavesdropping, which does not work for the same reason under macOS.

8 Lessons Learned

Standardization process. Bluetooth has been subject to varieties of attacks and a more rigid standardization process may help security and privacy of Bluetooth including BLE. During our study of the specification, we find it is often confusing and not consistent across chapters as our partner TI finds too. The confusion may lead to the fact that different vendors implement BLE protocols in quite different ways, for example, for error handling, IRK use and interaction between an application and the BLE stack. A similar standardization process to RFC (request for comments) for Internet standards would help protocol verification.

Secure framework for pairing. BLE has a suite of protocols addressing different aspects of this wireless personal area network technology. Our paper focuses on pairing. Defeating other attacks such as co-located attacks requires extra remedies [11, 12, 26]. These remedies often rely on the assumption that the communication is secure the first time the user configures the mobile and device, which can share a secret to protect later communication at the application layer. However, the assumption may not be true without proper pairing. We believe both initiators and peer devices shall have the option of the Secure Connections Only mode so that we can achieve mutual authentication between an initiator and its peer device. This SCO mode requires the support in the four stages of the pairing process. In this paper, we have carefully addressed the SCO mode at initiators. We also find some vendors do not correctly implement the SCO mode at the peer device as discussed in §6.3. Correct implementation of this mode at initiators and peer devices will be able to defeat attacks presented in this paper.

9 Related Work

Vulnerabilities in Bluetooth. Bluetooth before the Simple Secure Pairing (SSP) is not secure [27, 28] and is out of the scope of this paper. The Simple Secure Pairing is also vulnerable. For example, Haataja et al. [5] proposed MITM attacks against SSP of Bluetooth Classic in 2010. They assumed that the victim devices use only I/O capabilities to determine the association method and the attacking devices can pair with victim devices using Just Works. The latest BLE introduces the Secure Connections Only mode to defeat those attacks. Our work focuses on the Secure Connections Only mode.

Mike Ryan [29] built a BLE sniffer over Ubertooth and demonstrated that the Passkey Entry for LE legacy connections is not secure. His tool *crackle* can crack such connections and target BLE 4.0 and 4.1. Our paper addresses the latest BLE 4.2 and 5.x, which are considered secure against his attacks. The work by Rosa [30] is similar to Mike Ryan's work. Zegeye et al. cracked the BLE temporary key used in the pairing process by using a brute-force attack [31], which also extends the attack in [29]. Dazhi Sun et al. [32] proposed a method that can break Passkey Entry when the passkey is reused. The similar problem was also discussed in [4]. However, reusing a passkey is not recommended in BLE, which requires a random passkey shall be used in each pairing session with Passkey Entry. We assume a random passkey in this paper. Antonioli et al. [33] identified Bluetooth Classic specification authentication vulnerabilities and can downgrade the Secure Connections protocol into the Legacy Secure Connections protocol.

Bluetooth attacks on mobiles. Jasek et al. [34] studied possible attacks between a Bluetooth smart device and its mobile app. However, they study BLE 4.0 and 4.1, which do not have the Secure Connections Only mode for BLE. They attacked Passkey Entry with Mike Ryan's approach [29]. Many works reverse engineer particular products [35–37] and exploit faulty app protocols while we focus on the operating system level and programming framework issues. For example, Britt Cyr et al. performed a security analysis of wearable fitness devices [35]. They reverse engineered the devices, BLE communication traffic, and app, and used Mike Ryan's attacks against pairing. Zhang et al. analyzed the commands from four popular smart wristbands by sniffing packets without reverse engineering the apps [36], and presented replay and MITM attacks against those particular wristbands. BlueBorne [38] explored faulty BLE implementations. our attacks are not based on those issues. William et al. [39] and Melamed et al. [40] studied the spoofing attack and MITM attack between a Bluetooth smart device and its mobile app. They presented software based and hardware based attacks, but did not address how to attack two paired devices with a secure association method. Fawaz et al. [41] collected and analyzed the advertisement packets from 214 BLE devices and found that the poor design and implementation of BLE advertisements may lead to privacy leaks. We address pairing security in this paper. Muhammad Naveed et al. [11], Xu et al. [13] , Zhang et al. [26] and Sivakumaran et al. [12] also addressed Bluetooth security but not on pairing. Zuo et al. [42] fingerprint via UUIDs vulnerable IoT devices that use insecure pairing.

10 Conclusion

BLE 4.2 and 5.x have an SCO mode to enforce secure pairing such as Passkey Entry and Numerical Comparison for BLE devices. However, the BLE specification does not explicitly require an initiating device such as a mobile to support the SCO mode. This creates potential security vulnerabilities against both mobiles and their peer BLE devices. In this paper, we have systematically investigated Android's BLE programming framework and discovered four design flaws. We then present a suite of downgrade attacks and case studies exploiting these design flaws. To defend against these attacks, we patch Android to enforce secure pairing. We also explored other major OSes including iOS, macOS, Windows and Linux, and found all OSes have similar security issues and they all need to adopt the SCO mode at the initiators. We have performed extensive experiments to validate the identified attacks and proposed defense measures. We believe for mission critical BLE systems, the SCO mode shall be enforced on both initiators and responders.

Acknowledgements

We thank shepherd Nils Ole Tippenhauer as well as the anonymous reviewers for their insightful comments. This research was supported in part by US Natural Science Foundation (NSF) Awards 1931871 and 1915780, Semiconductor Research Corporation (2018-TS-2860), National Natural Science Foundation of China (Grant Nos. U1736203, 61877029). Any opinions, findings, conclusions, and recommendations in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] Bluetooth Special Interest Group (Bluetooth SIG), "Bluetooth Core Specification (V 4.2)," *Specification of the Bluetooth System*, 2014.

[2] ——, "Bluetooth Core Specification (V 5.1)," *Specification of the Bluetooth System*, 2019.

[3] Bluetooth Wireless Forum (SILICON LABS), "MITM Attack on 'Just Works' Pairing," 2017, Available at https://www.silabs.com/community/wireless/bluetooth/forum.topic.html/mitm_attack_on_just-OoG9 [Accessed: Apr, 2020].

[4] P. Sivakumaran and J. B. Alís, "A Low Energy Profile: Analysing Characteristic Security on BLE Peripherals," in *Proceedings of the 2018 Eighth ACM Conference on Data and Application Security and Privacy*, 2018, pp. 152–154.

[5] K. Haataja and P. J. Toivanen, "Two practical Man-in-the-Middle Attacks on Bluetooth Secure Simple Pairing and Countermeasures," *IEEE Trans. Wireless Communications*, vol. 9, no. 1, pp. 384–392, 2010.

[6] Google, "Android Open Source Project (AOSP)," 2020, Available at https://source.android.com/ [Accessed: Apr, 2020].

[7] Texas Instruments Product Security Incident Response Team (TI-PSIRT), "SIMPLELINK-CC13X2-26X2-SDK," 2019, Available at http://www.ti.com/tool/download/SIMPLELINK-CC13X2-26X2-SDK [Accessed: Apr, 2020].

[8] Bluetooth Special Interest Group (Bluetooth SIG), "Bluetooth Core Specification (V 4.2) - Secure Connections Only Mode, Vol 3, Part C, Page 373," *Specification of the Bluetooth System*, 2014.

[9] L. Li, T. F. Bissyandé, D. Octeau, and J. Klein, "Reflection-aware Static Analysis of Android Apps," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016, pp. 756–761.

[10] Android Development Guide, " Restrictions on Non-SDK Interfaces," 2020, Available at https://developer.android.com/about/versions/pie/restrictions-non-sdk-interfaces [Accessed: Apr, 2020].

[11] M. Naveed, X. Zhou, S. Demetriou, X. Wang, and C. A. Gunter, "Inside Job: Understanding and Mitigating the Threat of External Device Mis-Binding on Android," in *Proceedings of the 21st Annual Network and Distributed System Security Symposium*, 2014.

[12] P. Sivakumaran and J. Blasco, "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 *Proceedings of the 28th USENIX Security Symposium*, 2019, pp. 1–18.

[13] F. Xu, W. Diao, Z. Li, J. Chen, and K. Zhang, "BadBluetooth: Breaking Android Security Mechanisms via Malicious Bluetooth Peripherals," in *Proceedings of the 26th Annual Network and Distributed System Security Symposium*, 2019.

[14] Android Development Guide, "Android 6.0 Changes (Access to Hardware Identifier)," 2016, Available at https://developer.android.com/about/versions/marshmallow/android-6.0-changes#behavior-hardware-id [Accessed: Apr, 2020].

[15] Adafruit Inc., "Bluefruit LE sniffer," 2018, Available at https://www.adafruit.com/product/2269 [Accessed: Apr, 2020].

[16] Texas Instruments, "SimpleLink Bluetooth Low Energy CC2640R2F wireless MCU LaunchPad development kit," 2019, Available at http://www.ti.com/tool/LAUNCHXL-CC2640R2 [Accessed: Apr, 2020].

[17] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Androzoo: Collecting Millions of Android Apps for The Research Community," in *Proceedings of the IEEE/ACM 13th Working Conference on Mining Software Repositories*, 2016, pp. 468–471.

[18] R. Vallée-Rai, E. Gagnon, L. J. Hendren, P. Lam, P. Pominville, and V. Sundaresan, "Optimizing Java Bytecode Using the Soot Framework: Is It Feasible?" in *Proceedings of the 9th International Conference on Compiler Construction*, 2000, pp. 18–34.

[19] HID WG, "Input Boot Keyboard Report Characteristic Requires No Authentication Permission (Page 17)," *HID Service Specification 1.0*, 2011.

[20] S. Hopwood, "How Many Mobile Apps are Actually Used?" 2017, Available at https://www.apptentive.com/blog/2017/06/22/how-many-mobile-apps-are-actually-used/ [Accessed: Apr, 2020].

[21] BlueZ, "BLE Stack for Linux (BlueZ)," 2008, Available at https://git.kernel.org/pub/scm/bluetooth/bluez.git/tree/doc [Accessed: Apr, 2020].

[22] Apple Inc., "Accessory Design Guidelines for Apple Devices) ," 2019, Available at https://developer.apple.com/accessories/Accessory-Design-Guidelines.pdf [Accessed: Apr, 2020].

[23] ——, "Core Bluetooth (API for Bluetooth Smart)," 2020, Available at https://developer.apple.com/documentation/corebluetooth [Accessed: Apr, 2020].

[24] ——, "IOBluetoothDevicePair(Pairing API for Bluetooth Classic) ," 2020, Available at https://developer.apple.com/documentation/iobluetooth/iobluetoothdevicepair [Accessed: Apr, 2020].

[25] Microsoft Inc., "Universal Windows Platform Documentation," 2020, Available at https://docs.microsoft.com/en-us/windows/uwp/get-started/universal-application-platform-guide [Accessed: Apr, 2020].

[26] Z. Yue, W. Jian, L. Zhen, P. Bryan, and F. Xinwen, "BLESS: A BLE Application Security Scanning Framework," in *Proceedings of 2020 IEEE International Conference on Computer Communications*, 2020.

[27] A. Becker and I. C. Paar, "Bluetooth Security & Hacks," *Ruhr-Universität Bochum*, 2007.

[28] D. Kügler, "'Man-In-The-Middle' Attacks on Bluetooth," in *Proceedings of the 7th International Conference on Financial Cryptography*, 2003, pp. 149–161.

[29] M. Ryan, "Bluetooth: With Low Energy Comes Low Security," in *Proceedings of the 7th USENIX Workshop on Offensive Technologies*, 2013.

[30] T. Rosa, "Bypassing Passkey Authentication in Bluetooth Low Energy." *IACR Cryptology ePrint Archive*, vol. 2013, p. 309, 2013.

[31] W. K. Zegeye, "Exploiting Bluetooth Low Energy Pairing Vulnerability in Telemedicine," in *Proceedings of the 2015 International Telemetering Conference*, 2015.

[32] D.-Z. Sun, Y. Mu, and W. Susilo, "Man-in-the-Middle Attacks on Secure Simple Pairing in Bluetooth Standard V5.0 and Its countermeasure," *Personal and Ubiquitous Computing*, vol. 22, no. 1, pp. 55–67, 2018.

[33] D. Antonioli, N. Tippenhauer, and K. Rasmussen, "BIAS: Bluetooth Impersonation AttackS," in *Proceedings of the 2020 IEEE Symposium on Security and Privacy*, 2020, pp. 1539–1552.

[34] S. Jasek, "Gattacking Bluetooth Smart Devices," in *Proceedings of the Black Hat USA Conference*, 2016.

[35] B. Cyr, W. Horn, D. Miao, and M. Specter, "Security Analysis of Wearable Fitness Devices (Fitbit)," *Massachusets Institute of Technology*, p. 1, 2014.

[36] Q. Zhang and Z. Liang, "Security Analysis of Bluetooth Low Energy Based Smart Wristbands," in *Proceedings of the 2nd Frontiers of Sensors Technologies International Conference*, 2017, pp. 421–425.

[37] A. K. Das, P. H. Pathak, C.-N. Chuah, and P. Mohapatra, "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, 2016, pp. 99–104.

[38] ARMIS, "The Attack Vector 'BlueBorne' Exposes Almost Every Connected Device," https://armis.com/blueborne/, 2017.

[39] W. Oliff, A. Filippoupolitis, and G. Loukas, "Evaluating the Impact of Malicious Spoofing Attacks on Bluetooth Low Energy based Occupancy Detection Systems," in *Proceedings of the 2017 Software Engineering Research, Management and Applications*, 2017, pp. 379–385.

[40] T. Melamed, "An Active Man-in-the-middle Attack On Bluetooth Smart Devices," *Safety and Security Studies*, p. 15, 2018.

[41] K. Fawaz, K. Kim, and K. G. Shin, "Protecting Privacy of BLE Device Users," in *Proceedings of the 25th USENIX Security Symposium*, 2016, pp. 1205–1221.

[42] C. Zuo, H. Wen, Z. Lin, and Y. Zhang, "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 (CCS)*, 2019, p. 1469–1483.