Attacks Against BLE Devices by Co-located Mobile Applications

Pallavi Sivakumaran
Centre for Doctoral Training in Cyber Security
Royal Holloway University of London
Email: pallavi.sivakumaran.2012@rhul.ac.uk

Jorge Blasco
Information Security Group
Royal Holloway University of London
Email: jorge.blascoalis@rhul.ac.uk

***Abstract*—Bluetooth Low Energy (BLE) is a fast-growing wireless technology with a large number of potential use cases, particularly in the IoT domain. With many of these use cases, the BLE device stores sensitive user data or critical device controls, which may be accessed by an augmentative Android or iOS application. Uncontrolled access to such data could violate a user's privacy, cause a device to malfunction, or even endanger lives. The BLE specification aims to solve this with network layer security mechanisms such as pairing and bonding. Unfortunately, this doesn't take into account the fact that many applications may be co-located on the same mobile device, which introduces the possibility of unauthorised applications being able to access and modify sensitive data stored on a BLE device. In this paper, we present an attack in which an unauthorised Android application can access pairing-protected data from a BLE device by exploiting the bonding relationship previously triggered by an authorised application. We discuss possible mitigation strategies, and perform an analysis over 13,500+ BLE-enabled Android applications to identify how many of them implement such strategies to avoid this attack. Our results indicate that over 60% of these applications do not have mitigation strategies in place in the form of application-layer security, and that cryptography is sometimes implemented incorrectly in those that do. This implies that the corresponding BLE devices are potentially vulnerable to unauthorised data access by malicious applications.**

I. Introduction

Bluetooth is a well-known and widely-adopted technology standard for wireless data transfer, currently deployed in billions of devices worldwide [1]. A more recent addition to the Bluetooth standard is Bluetooth Low Energy (BLE), which incorporates a simplified version of the Bluetooth stack and targets low-energy, low-cost devices. BLE differs from Classic Bluetooth[1] in several ways, most notably in terms of the format of the data that can be accessed and the protocols used for doing so. Its focus on resource-constrained devices has made BLE highly suited for IoT applications [2], including personal health and fitness monitoring [3], asset tracking [4], vehicular management [5], and home automation and security [6].

At present, it is fairly common for a BLE device to be partnered with an application that runs on a BLE-capable mobile operating system, such as Android or iOS. The mobile application might read data from the BLE device and display it to the user, as with some fitness applications; or it might write data to, and thereby control the functionality of, the BLE device, as in the case of some "smart" padlocks. This data may be sensitive or critical (for example, the glucose measurement values stored by a BLE-enabled glucose meter, or the values that control a door's locking mechanism in a smart home security system), and read/write access to such data should therefore be restricted.

The Bluetooth specification provides means for restricting access to BLE data by requiring that access requests be sent over an encrypted channel. This can be achieved via *pairing* and *bonding*, which are mechanisms for authenticating the communicating devices to each other and generating keys to encrypt the transport between them. However, when multiple applications reside on a single host device and share the same transport, as is the case with mobile devices, there is potential for a malicious application to abuse the trusted relationship between the host and the BLE device that was established by an authorised application [7]. This could result in the malicious application gaining unauthorised access to data on the device. For example, it may be able to read sensitive user information, or overwrite the data (or potentially even the firmware) on the BLE device to induce unintended functionality.

In this work, we analyse the level of access to pairing-protected BLE data that is possible for an unauthorised Android application, in the presence of a co-located authorised application. We demonstrate that, due to the nature of BLE data access mechanisms and due to how BLE communication channels have been implemented within mobile operating systems, unauthorised applications may be able to easily read and write pairing-protected data on a BLE device without the user's knowledge. We also show that these unauthorised applications may be able to do so while requesting minimal permissions, thereby making them appear less invasive than even the authorised application.

We discuss protection mechanisms that might mitigate against such unauthorised data access, and perform a large-scale static analysis of 13,787 BLE-enabled Android applications, filtered down from an original dataset of over 4 million applications, to detect whether they have implemented such mechanisms. While the results vary for data reads vs. writes, overall they show that more than 60% of the tested applications do not provide cryptography-based application-layer security for BLE data. This number rises to about 85% for those applications that are categorised under "Health & Fitness" and "Medical". This information, combined with the download counts for each application, allows us to estimate a lower bound for the number of BLE devices that may be vulnerable to data access by unauthorised co-located applications.

[1]In this paper, we use the term *Classic Bluetooth* for the original version of the technology, to distinguish it from BLE.

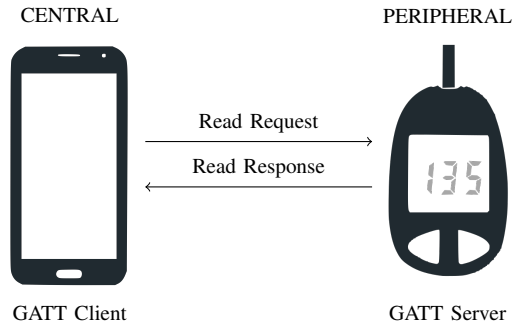


Fig. 1. GATT communications between a mobile phone and a BLE-enabled glucometer, where the phone acts as the GATT client and the glucometer as the GATT server.

The rest of this paper is structured as follows: Section II provides an overview of key BLE and Android concepts, particularly those related to data access mechanisms and restrictions. Related work is reviewed in Section III. We describe our experiments with BLE data access and explore possible mitigation strategies in Section IV. Section V details our marketplace application analysis and discusses the results. Section VI concludes the report.

II. Background

Two devices that communicate using BLE will operate in an asymmetric configuration, with the more powerful device, referred to as the *central*, taking on most of the resource-intensive work. The resource-constrained device is termed the *peripheral* and performs tasks that are designed to consume fewer resources.

Peripheral devices, such as smart locks or glucose meters, make their presence known by *advertising* on dedicated channels. Advertising packets contain, among other information, details about the peripheral's MAC address. Central devices, such as mobile phones, *scan* the advertising channels and send connection messages to peripherals that they wish to communicate with. Connection messages will contain the central device's MAC address. The exchanged MAC addresses are used by the two devices to identify each other during subsequent reconnections.

A. Data Access on BLE Devices

BLE, unlike Classic Bluetooth, can only handle discrete data known as *attributes*, and a BLE device stores multiple such attributes within an attribute database. Attributes are stored and accessed according to rules specified by the *Attribute Protocol* (ATT) and the *Generic Attribute Profile* (GATT), both of which are defined in the Bluetooth standard. There are different types of attributes, of which *characteristics* are the most relevant for our analysis, as they hold the actual data of interest [8].

When one BLE-enabled device wants to access attributes on another BLE device, the device that initiates the exchange will take on the role of GATT *client* and the other will act as the GATT *server*. In this paper, we focus on the scenario where the BLE peripheral (for example, a glucose meter), acts as the server, and the mobile phone acts as the client, as shown in Figure 1.

BLE attribute permissions: Every attribute has associated permissions that control how the attribute may be accessed. The Bluetooth specification defines the following three permission types: (1) *Access permissions* define whether an attribute can be read and/or written. (2) *Authentication permissions* indicate the level of authentication and encryption that needs to be applied to the transport between the two devices before the attribute can be accessed. (3) *Authorisation permissions* specify whether end user authorisation is required for access.

When a GATT client sends a read or write request for an attribute to a GATT server, the server will check the request against the permissions for that attribute, to determine whether the requested access is possible, and whether the client is authenticated and/or authorised, if required. Access permissions are straightforward, in that an attribute is only readable or writable if its access permissions specify it to be so. In the case of authentication permissions, if the attribute requires an authenticated or encrypted link before it can be accessed[2], and if such a link is not present when the access request is made, then the server responds with an `Insufficient Authentication/Encryption` message. At this point, the client can initiate the pairing process in order to authenticate itself and establish keys for encrypting the transport. Provided the attributes' authentication requirements are met[3], subsequent requests made by the client, over the encrypted link, will be fulfilled by the server. This procedure for handling authentication requirements is well-defined in the Bluetooth specification. Authorisation requirements, on the other hand, are implementation-specific and largely left up to developers.

Once two devices complete the pairing process, they typically go through an additional *bonding* process, during which long-term keys are established. This prevents the need for going through the pairing process again if they disconnect and subsequently reconnect, provided they retain the long-term keys. Upon reconnection, the link encryption process will be initiated, usually immediately, using the stored keys.

B. BLE on Android

From version 4.3 (API Level 18) onwards, Android offers built-in support for BLE functionality, with the Android device assuming the role of the BLE central [9]. Android applications can utilise the functionality provided by the Bluetooth APIs to scan for and connect to BLE devices, and to access data on them. For this, the application needs a reference to the Android device's Bluetooth adapter (that is, the hardware radio). This is termed the `BluetoothAdapter` object, and is accessed via a management component known as the `BluetoothManagerService`.

The `BluetoothAdapter` object represents the single adapter available to the entire system. It can be used to instantiate a scanner to scan for BLE peripherals in the vicinity, to get a list of Bluetooth devices that the Android device has bonded with, and to get a reference to a specific BLE device. Once a `BluetoothDevice` object is created, the application can invoke methods to connect to the GATT server on the BLE device and thereafter read and write characteristics.

[2]We refer to such attributes/characteristics as "pairing-protected" attributes/characteristics in this paper.

[3]Different attributes can require different strengths of authentication.

Note that an Android application does not have direct access to the BLE stack on the Android device. It can only make use of API calls, which in turn call intermediate methods that invoke functionality within the BLE stack.

Android application permissions: Applications accessing data from a BLE GATT server require `BLUETOOTH` permissions. If the application also scans for and pairs with BLE devices, then an additional `BLUETOOTH_ADMIN` permission is required. These are both considered to be "normal" permissions and are granted automatically by the operating system after installation, without any need for user interaction.

From Android version 6.0 onwards, applications also need to request `LOCATION` permissions if they intend to invoke the BLE scanner without a filter[4]. Because location information involves a user's privacy, these permissions are classed as "dangerous" and will prompt the system to display a dialog box the first time the application is executed. The user will have to grant the requested permission in order for the application to be able to use the BLE scanning function.

Permissions are important from the perspective of the user. A long list of permissions specified in an application manifest might cause concern among cautious users, but many users will probably not even be aware of it and will therefore install the application anyway [11]. However, more users will likely take notice if, when they first open an application, a system dialog pops up, requesting access to sensitive information, such as their location.

III. Related Work

User privacy has received particular attention in the BLE research community because several widely-used BLE devices, such as fitness trackers and continuous glucose monitors, are intended to always be on the user's person, thereby potentially leaking information about the user's whereabouts at all times. Some of the research has focused on the threats to privacy based on user location tracking [12], [13], while others explored the possibility of obtaining personal user data from fitness applications or devices [14], [15].

While our research *is* concerned with data access and user privacy, we focus more on the impact on privacy and security due to how the BLE standard has been implemented in mobile device architectures, as well as how it is applied by application developers, rather than due to individual BLE firmware design.

The work that is most closely related to ours is research by Naveed, et al. from 2014, which specifically explored the implications of shared communication channels on Android devices [7]. In their paper, the authors discussed the issue of Classic Bluetooth and Near Field Communication (NFC) channels being shared by multiple applications on the same device. They then demonstrated attacks against (Classic) Bluetooth-enabled medical devices, by developing an unauthorised Android application to read sensitive data from and write random data to the devices. The authors also performed an analysis of 68 Bluetooth-enabled applications that handled private user data, and concluded that the majority of them offered no protection against this attack. Finally, they proposed an operating-system level control for mitigating the attack.

Our work specifically targets pairing-protected characteristics on BLE devices, because BLE appears to slowly be replacing Classic Bluetooth in the personal health and home security domains. BLE also has a simplified stack, which leads us to hypothesise that attacks against BLE may be even easier to execute than those against Classic Bluetooth. Further, given that Android introduced a new permissions mechanism from API Level 23 onwards, an analysis of how this may affect the user experience or impact malicious applications' capabilities seems pertinent. We also perform a much larger-scale analysis over 13,500+ Android applications, to determine how prevalent application-layer security is among applications that access data on BLE peripherals.

IV. BLE Co-Located Application Attack

Section II-A outlined the mechanism that is used to access pairing-protected characteristics on a BLE peripheral. In this section, we describe our experiments for testing shared access by multiple Android applications to pairing-protected characteristics on a BLE device.

A. Experimental Set-up

For our experiments, we developed a basic BLE peripheral using the Nordic nRF51 Development Kit. The peripheral had a protected characteristic which was readable and writable, and which specified one of the strongest authentication requirements: LE Secure Connections with Numeric Comparison, which uses Elliptic-Curve Diffie-Hellman for key generation.

Two Android applications were also developed: one that functioned as the official or authorised application ("OfficialApp"), which complements the BLE peripheral and which is expected to be able to access the data on it; and the other as an application from a different developer ("AttackerApp"), which is not expected to be able to access pairing-protected data on the BLE device. Each application was signed by a unique key to emulate different development sources. This mimics a real-world scenario, where an authorised application resides on an Android device alongside multiple other applications from various sources.

We conducted our experiments on an Alcatel Pixi 4 mobile phone, running Android 6.0, and on a Google Pixel XL, running Android 8.1.0. These particular versions of the Android platform were selected because they are the most widely-deployed release[5] and the latest stable release (as of July 2018), respectively.

B. Attack 1: Reuse of Pairing Credentials

This attack demonstrates that the BLE credentials that are stored on an Android device are implicitly available to *all* applications on the device, rather than just the application that originally triggered the pairing.

We begin by launching the OfficialApp (emulating a typical scenario where the user first configures the BLE device via the companion application). The OfficialApp scans for and

[4] A filter enables an application to specify certain criteria, such as the BLE device's name or MAC address, which is used to filter the scan results [10].

[5] 23.5% of all deployments, according to Android Dashboards [16].

connects to the Nordic BLE peripheral, and then attempts to read the value of the protected characteristic on the peripheral. As there is no previous pairing history between the Nordic and the Android, this results in the peripheral responding with an `Insufficient Authentication` message, which prompts the Android operating system to initiate the pairing process. As part of this, it brings up the pairing system dialog, thereby making the user aware of the fact that the OfficialApp is attempting to access data from their BLE device.

Once the pairing has completed, we force-close the OfficialApp. This is not a necessary condition for the attack itself, but is used to demonstrate that the long-term keys are reused after a connection is closed. The AttackerApp is subsequently launched (in a real world scenario, this would generally happen because a user has been tricked, using social engineering, into downloading and opening the application. This is a fairly common method used by malware in mobile scenarios [17]). The AttackerApp then scans for[6] and connects to the Nordic. We found that, once connected, the AttackerApp was able to read from and write to the protected characteristic with no further user intervention being required, i.e., without the need to pair. This was despite the fact that the AttackerApp had never paired with the Nordic before.

Another interesting observation here is that, not only is the unauthorised AttackerApp able to access pairing-protected information from the BLE device, but also the user is likely to be unaware of the fact that this data access is taking place, as there is no indication during link re-encryption and subsequent attribute access.

Analysing BLE traffic logs from the Android device, we observed that when the AttackerApp prompts the Android OS to establish a connection with the BLE device, the OS completes the connection process and then automatically initiates link encryption, presumably with the keys that were generated and stored during the previous bonding process, which was triggered by the OfficialApp. This has been depicted in the illustrative message exchange in Figure 2.

The traffic logs also reveal that all requests of a particular type (e.g., read requests) that are made by the two Android applications are identically crafted and sent as if from the Android system, with the same source MAC address. This suggests that the BLE peripheral will be unaware that its data is accessed by two different entities.

Attack limitations: The main limitation for the AttackerApp here is that it requires the `BLUETOOTH` and `BLUETOOTH_ADMIN` permissions in its manifest, and also needs to explicitly request `LOCATION` permissions at first runtime in order to be able to invoke the BLE scanner. While, as mentioned in Section II-B, users may not notice the Bluetooth permissions within the manifest, they may be more suspicious when the system displays a dialog box requesting access to their location. This limitation is only present from Android API Level 23 onwards, and it is of course possible that unsuspecting users may grant the permissions anyway. Nevertheless, our next experiment shows that, subject to certain conditions, unauthorised applications may be able to access information from a BLE peripheral even without some of these permissions.

[6]In this paper, we only consider the scenario where the applications scan without a filter.

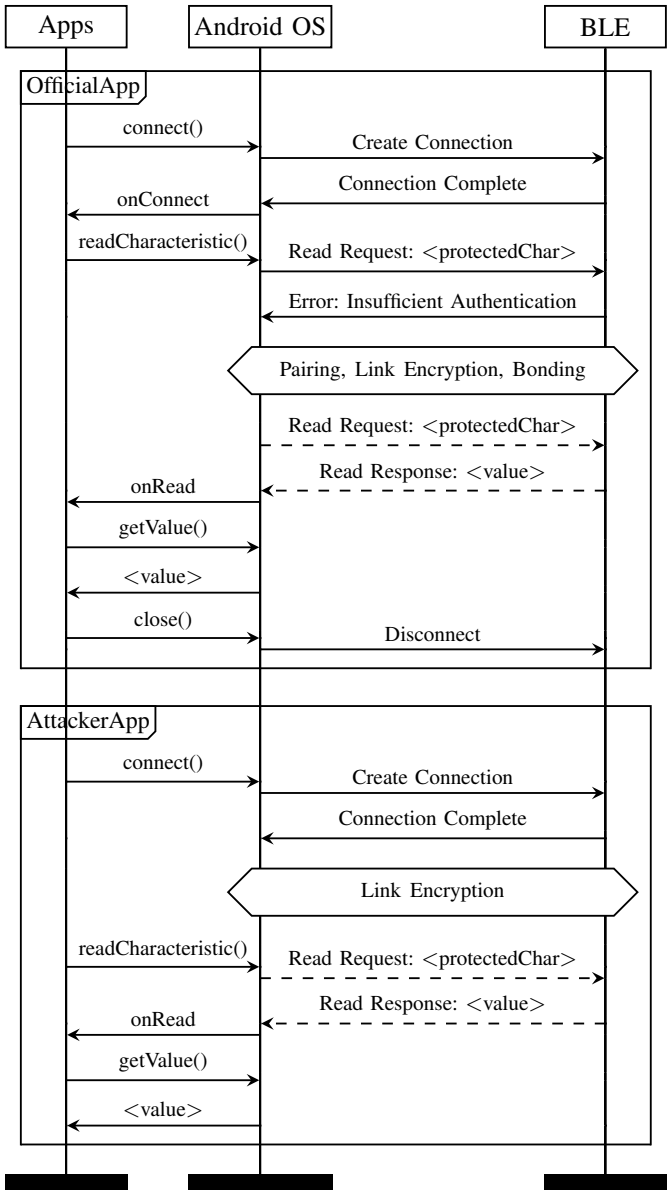


Fig. 2. Illustrative message exchange when co-located applications access pairing-protected characteristics. *Note 1:* The "Apps" run on top of the "Android OS" layer within the Android device. *Note 2:* Scanning has not been depicted, but can be assumed to take place prior to the initial `connect()` message for both applications. Scanning returns a list of BLE peripherals in the vicinity, from which the application selects one to connect to. *Note 3:* Dashed lines indicate encrypted traffic.

C. Attack 2: Reuse of Connection

Android is designed in such a way that a BLE peripheral can, at any given time, be used by multiple applications residing on the same Android device [18]. This design suggests that, when one Android application has established and is participating in a connection with a BLE peripheral, other applications on the Android device will also be able to access

information on the peripheral at the same time. Our second experiment demonstrates this, both for the general case and for pairing-protected characteristics.

We start with the same experimental set-up as for Attack 1, but with some of the AttackerApp's functionality modified. As before, when the OfficialApp is launched, it scans for and connects to the Nordic peripheral and triggers the pairing process. Once pairing completes, and while the connection is still active, the AttackerApp is invoked (this could be done by the user, or automatically by a timer previously set by the AttackerApp if it has been run at least once). The AttackerApp searches for connected BLE devices using the `BluetoothManager.getConnectedDevices()` API call, with `BluetoothProfile.GATT` as the argument. This reuses the existing `BluetoothAdapter` object and returns a list containing references to all the devices that are currently in a connected state. The AttackerApp is then able to directly connect to the GATT server on the Nordic and read and write to the characteristics on it, without the need for creating a new connection to the peripheral.

Our tests show that the extent of the access allowed to the unauthorised application is not confined to unprotected characteristics. As long as the Android device has previously paired with the BLE peripheral, the AttackerApp is able to read and write *all* applicable characteristics[7]. An example message flow where the AttackerApp writes to the protected characteristic on the Nordic peripheral (which the OfficialApp subsequently reads) has been depicted in Figure 3.

From BLE traffic logs, we observed that again, the fact that it receives requests from two different entities is transparent to the BLE peripheral. That is, the GATT Read/Write requests from different applications are sent from within the same connection and are identical to each other, which means the BLE device sees all requests as if from a single source.

The most interesting observation from this experiment is that, while the OfficialApp has to scan for the BLE device before it can connect to it, the AttackerApp merely has to query the Android OS for a list of already connected devices. As mentioned in Section II-B, this means that the OfficialApp needs to request the "dangerous" `LOCATION` permissions, in addition to the "normal" `BLUETOOTH` and `BLUETOOTH_ADMIN` permissions. In contrast, the AttackerApp only requires the declaration of the normal `BLUETOOTH` permission. This makes the AttackerApp appear to be less invasive in the eyes of a user, since it does not request any permission that involves user privacy. This could play a part in determining the volume of downloads for a malicious application. For example, a malicious application that masquerades as a gaming application, and which does not request any dangerous permissions, may be more likely to be downloaded by end users as opposed to one that requests location permissions.

Attack limitations: In this scenario, the obvious limitation for the AttackerApp that requests only the `BLUETOOTH` permission is that the application will only be able to access data from the BLE peripheral when the peripheral is already in a connection with (another application on) the Android device. That is, data access will have to be opportunistic. This can be achieved, for example, by periodically polling for a list of connected devices.

[7]We term a characteristic as applicable if it has the required access permission. That is, an applicable characteristic for a write operation is one that has Write permissions.

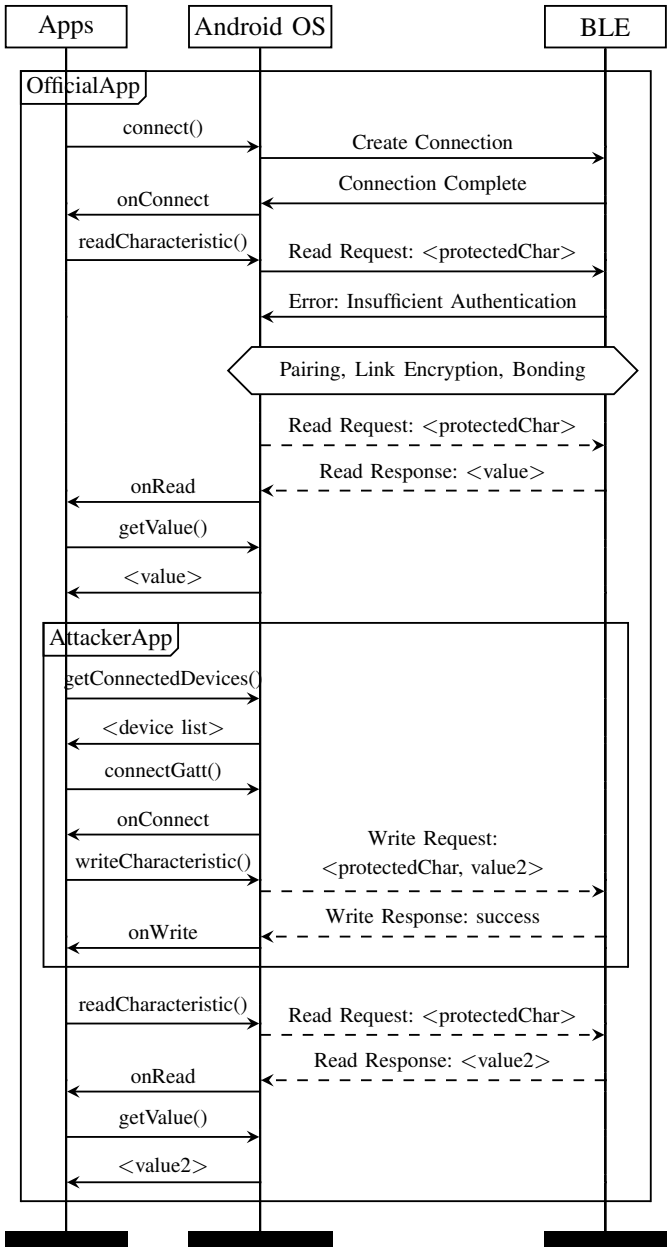


Fig. 3. Illustrative message exchange when accessing pairing-protected data by reusing an existing connection. *Note 1:* Scanning has not been depicted, but can be assumed to take place prior to the initial `connect()` message for the OfficialApp.

D. Discussion

To summarise the findings from our experiments: A malicious application that requests the `BLUETOOTH`, `BLUETOOTH_ADMIN` and `LOCATION` permissions will be able to scan for and connect to a BLE device (or reuse an existing connection with the device) and access pairing-

protected data on it. A malicious application that requests only the `BLUETOOTH` permission will only be able to access pairing-protected data on a BLE device by reusing an existing connection. In both cases, we assume that an authorised application has previously paired/bonded with the BLE device.

Possible attack scenarios: In both our experiments, the AttackerApp was able to read and write pairing-protected data from the BLE device. The simplest form of attack would then be for a malicious application to perform unauthorised reads of (for example) personal user data and perhaps relay this to a remote server. Even more serious would be for a malicious application to overwrite values on the BLE device, such that the written data either causes unexpected behaviour on the device, or is read back by the legitimate application, thereby giving the user an incorrect view of the data on the peripheral[8]. If, for example, the characteristic holding a user's glucose measurement readings was overwritten (assuming that the characteristic was write-enabled), the consequences could be severe. Further, with some BLE chipsets, it may be possible to update the peripheral's firmware via GATT writes. If this mechanism is not suitably protected, then a malicious application could potentially install malicious firmware onto the BLE device, as we demonstrate in Section V-E.

Comparison with Classic Bluetooth: In their experiments with Classic Bluetooth, Naveed et al. [7] consider a scenario where the Bluetooth on Android is only turned on by the user for brief intervals, such as when they want to use their Bluetooth device with the official application, with a view to conserving the Android device's battery. They found that, with Classic Bluetooth, an unauthorised Android application would not be able to obtain data from a Bluetooth device if the authorised application had already established a socket connection with the device, as only one application could be in communication with the device at one time. Therefore, a malicious application would either require some side-channel information in order to determine the correct moment for data download, or would need to interfere with the existing connection, thereby potentially alerting the user. This limits the attack window for the malicious application.

Our experiments show that this is not the case with BLE communication channels. With BLE, if the official application has established a connection with the BLE device, then this connection can be utilised by any application that is running on the Android device. That is, a malicious application does not have to wait for the authorised application to disconnect before it can read or write data.

E. Possible Mitigation Strategies and Recommendations

In this section, we discuss potential methods for preventing the attacks detailed in Sections IV-B and IV-C, and identify the pros and cons of each.

Operating system changes: Allowing all applications on an Android device to share BLE communication channels and long-term keys may well be by design, particularly since the BLE standard does not provide explicit mechanisms for selectively allowing or denying access to data based on the source *application*. This model may work in some situations, for example on a platform where all applications originate from the same trusted source. However, the Android ecosystem is such that, many of the applications on a device are from different and potentially untrusted sources. In this scenario, providing all applications with access to a common BLE transport opens up possibilities for attack, as we have demonstrated.

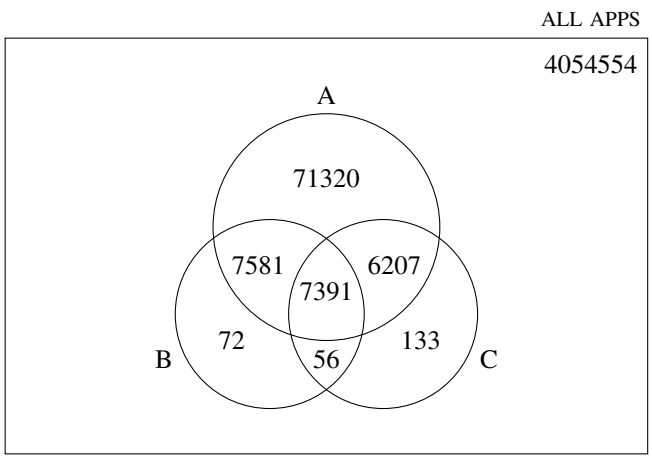
The attacks we have described are possible because of how the mobile operating system handles BLE communication channels. Therefore, modifying the Android operating system (as well as all other similarly functioning operating systems[9]) to allow some form of association between BLE credentials and the application that invokes the pairing/bonding process, and to check each data access request against the permissions associated with the requesting application, will eliminate the problem. This approach is favoured by Naveed et al. and they propose a rearchitected Android framework which will create a *bonding policy* when an application triggers pairing with a Bluetooth device [7]. This strategy has the advantage that Bluetooth devices will be protected by default from unauthorised access to their data. Further, assuming a suitably strong pairing mechanism is used, a minimum level of security will also be guaranteed. However, not only will the operating systems need to be modified, but also a mechanism is needed for ensuring that all users' mobile devices are updated. Otherwise, it is fairly likely that this measure will result in a fragmented ecosystem, with some devices running the modified operating system with protection mechanisms, and others running older versions of the OS with no protection.

Application-layer security: At present, the only mechanism available for mitigating unauthorised data access by co-located applications is the implementation of application-layer security. That is, rather than relying solely on the pairing provided by the underlying operating system, developers can implement end-to-end security from their Android application to the BLE peripheral firmware. Most BLE chipsets allow for this to be accomplished by specifying authorisation permissions for the relevant characteristics. This usually results in the BLE device intercepting read/write requests to the specified characteristics, performing developer-specified validation, and then either completing the read/write or returning an error. This requires pre-shared symmetric keys or the implementation of a secure key generation mechanism both on the BLE device and within the Android application.

One advantage of this method is that it gives the developer complete control over the strength of protection that is applied to BLE device data, as well as over the timings of security updates. However, leaving the implementation of security to the developer runs the risk of cryptography being applied improperly, thereby leaving the data vulnerable [19]. For existing developments, retrofitting application-layer security would mean that both an update for the Android application

[8]In some cases, the correct data may be stored within the peripheral's internal storage, but copies may be stored as GATT attributes, and these could be overwritten by a malicious application. We observed such behaviour during the testing of a fitness tracker, where we were able to overwrite the Device Name characteristic. When the device was rebooted, it reverted to the factory-set name. However, in the interim, it advertised the incorrect name.

[9]Preliminary tests on iOS indicate that similar behaviour can be expected, subject to restrictions on the application running in the background and App Store approval.



A - APKs that request the BLUETOOTH_ADMIN permission.

B - APKs that declare the android.hardware.bluetooth_le feature.

C - APKs that call android.bluetooth.BluetoothDevice.connectGatt.

Fig. 4. Number of APKs that declare or call permissions, features and methods related to BLE. All APKs within A,B,C request the BLUETOOTH permission.

TABLE I. NUMBER OF BLE APPLICATIONS AND DOWNLOAD COUNTS PER GOOGLE PLAY CATEGORY

Category	# APKs [unique apps]	Download count (millions)
Health & Fitness	2125 [646]	274.47
Business	1065 [609]	33.27
Lifestyle	925 [550]	46.97
Sports	930 [457]	16.05
Tools	787 [375]	5201.34
Travel & Local	630 [319]	13.00
Entertainment	291 [158]	122.84
Shopping	315 [143]	122.89
Productivity	302 [134]	31.23
Education	188 [134]	2.44
Music & Audio	257 [130]	22.25
Maps & Navigation	243 [108]	24.70
Medical	284 [87]	5.01
Finance	201 [84]	77.38
Communication	306 [80]	743.79
Books & Reference	69 [52]	0.19
Social	160 [45]	162.90
Other	594 [256]	175.19

[a] Sorted by the number of unique applications per category.
[b] Some APKs within the dataset are no longer available on Google Play and hence, have no corresponding category. These have not been included.
[c] The assumption has been made that all versions of a particular application fall under the same category.

and a firmware update for the BLE device would be required, and there is a risk that the BLE firmware update procedure itself may not be secure [20]. It is also possible that no protection is applied at all, due to an assumption that protection will be handled by pairing. In the next section, we explore the current state of application-layer security deployments via a large-scale analysis of BLE-enabled Android applications.

V. MARKETPLACE APPLICATION ANALYSIS

As evidenced from our experiments, it is fairly straightforward for any Android application to connect to and write or obtain information from a BLE device. As discussed in Section IV-E, the only mitigation strategy at present is for developers to implement application-layer security, typically in the form of cryptographic protection, between the Android application and the application layer within the BLE firmware. In this section, we explore how many existing applications and devices actually implement such security mechanisms or, more accurately, how many do *not* and are therefore potentially vulnerable to the types of attack demonstrated in Section IV.

To identify the presence of such application-layer security, there are two possible targets for analysis: BLE peripheral firmware or Android applications. BLE firmware needs to be obtained from the peripheral itself. This would necessitate the purchase of a large number of devices, which would not be financially viable. Further, BLE firmware may not be easy to reverse and analyse, as the firmware image is usually a .hex file, which can typically only be converted to binary or assembly. Android applications, on the other hand, are easier to obtain, and a number of decompilers exist that allow for conversion to human-readable format.

We therefore target Android applications for our analysis and perform the following: (1) obtain a substantial dataset of BLE-enabled Android applications, (2) determine the BLE method calls and the cryptography libraries of interest, and (3) determine whether BLE reads and writes make use of cryptographically processed data.

A. Application Dataset

Our dataset was obtained from the AndroZoo project [21]. This is an online repository that has been made available for research purposes and which contains APKs from several different application marketplaces. We focus on only those APKs that were retrieved from the official Google Play store, which nevertheless resulted in a fairly sizeable dataset of over 4 million applications. This dataset (dated 20 Feb 2018) includes multiple versions for each application, as well as applications that may no longer be available on the marketplace, which indicates that the dataset of unique and current applications is smaller. We chose to perform our analysis over the entire dataset, rather than focusing on only those applications that are currently available from the marketplace. This was in part because older versions of an application may still be residing on users' devices, and in part to be able to identify trends in application-layer security deployments over time.

Figure 4 depicts the number of applications within the dataset that declare or call permissions, features and methods relating to BLE. As we are only interested in those applications that perform BLE data access, and because such access always requires communicating with the GATT server on the BLE peripheral, the APKs were filtered by calls to the `connectGatt` method, which is called prior to performing any data reads or writes. 13,787 APKs (region C in Figure 4) from the original set of 4,000,000+ applications invoke calls to this method, and these formed our final dataset.

Application categories: Applications are categorised in Google Play according to their primary function, such as "Productivity" or "Entertainment", and it may be possible to gauge the sensitivity of the BLE data handled by an application based on the category it falls under. For example, "Health and

Fitness" applications are probably more likely to hold personal user data than "Entertainment" applications.

The number of APKs per category has been listed in Table I for our dataset. Approximately 25% of the applications (17% of unique apps) fall under the categories of "Health and Fitness" and "Medical", with a cumulative download count of over 275 million.

B. Identification of BLE Methods and Crypto-Libraries

We perform our analysis against specific BLE methods and crypto-libraries. With BLE methods, we are primarily interested in methods involving data writes and reads. For data writes, the `BluetoothGattCharacteristic` class within the `android.bluetooth` package has `setValue` methods that set the locally-stored value of a characteristic, which is then written out to the BLE peripheral. For reads, the same class has `getValue` methods, which return data that is read from the BLE device. The methods that handle BLE data values have been listed in Table II, and function as the starting point for our analysis. In a few APKs that we analysed, BLE methods were also called from within other, vendor-specific packages. However, we do not include these in our analysis as they are now obsolete.

For cryptography, Android builds on the Java Cryptography Architecture [22] to provide a number of APIs, contained within the `java.security` and `javax.crypto` packages, for integrating security into applications. While it is possible for developers to implement their own algorithms, Android recommends against this [23]. Therefore, we look for evidence of calls to these two packages as an indication of application-layer security.

C. Determining the Presence of Application-Layer Security

Identification of cryptographically-processed BLE data is in essence a taint-analysis problem. For instance, a call to an encryption method will taint the output variable that may later be written to a BLE device. There are a number of tools available for performing such analyses, for example, Flowdroid [24] and Amandroid [25], [26]. Both tools allow for specifying custom data sources and sinks, and attempt to determine a path from a source to a sink. After some initial testing, we opted to use Amandroid.

We ran our dataset of APKs through Amandroid, with sources and sinks reflecting methods from the BLE and crypto-libraries mentioned in Section V-B[10]. Amandroid can be a fairly resource-intensive process, with a single analysis sometimes utilising over 10GB of RAM and taking several hours to complete. As our dataset consisted of 13,787 APKs, and as two sets of analyses had to be performed for each APK (one each for BLE Reads and Writes), Amandroid was deployed on multiple servers and executed in multiple parallel threads within a server, to speed up the overall execution time. Even so, a time limit of 30 minutes was necessary, to prevent some APKs from unduly tying up resources. Despite the high processing power of the servers, a large proportion of analyses (approximately 45%) did time out without completing. We

[10]The source-sink files that were used are available at https://github.com/projectbtle/BLECryptracer.

```
1  .method private
     a(Landroid/bluetooth/BluetoothGatt;[B...)V
2    .locals 10
3
4    .prologue
5    const/4 v9, 0x2
6    const/4 v8, 0x3
7    const/4 v7, 0x1
8    ...
9    invoke-virtual {v0, v3},
       Landroid/bluetooth/BluetoothGattService;->
       getCharacteristic(Ljava/util/UUID;)
       Landroid/bluetooth/BluetoothGattCharacteristic;
10
11   move-result-object v3
12   ...
13   invoke-virtual {v3, p2},
       Landroid/bluetooth/BluetoothGattCharacteristic;
       ->setValue([B)Z
14   invoke-virtual {v1, v3},
       Landroid/bluetooth/BluetoothGatt;
       ->writeCharacteristic(Landroid/bluetooth/
       BluetoothGattCharacteristic;)Z
```

Fig. 5. Sample smali code for BLE attribute write.

believe this may be because Amandroid maps out the entire call graph before it performs taint analysis, and also attempts to identify every possible path from a source to a sink, whereas our goal is simply to determine whether there exists *any* such path (i.e., to identify if an application is using any kind of cryptography-based application-layer security).

The Amandroid results indicated that only 0.25% of the applications (that did not time out) used cryptographically-processed data for BLE writes and only 0.06% applied cryptographic processing for data read from BLE devices. However, when we performed a manual analysis of some applications, to verify the obtained results, we found that Amandroid did not identify cryptographically-processed data used in some libraries included with an application. This may have been due to there being no straightforward call to the library or due to exceeding the per-component analysis time limit.

Because BLE functionality is incorporated solely via external libraries for some applications, an analysis that does not identify them will return an inaccurate result. We therefore developed a custom analysis tool, called *BLECryptracer*, using Python, to analyse *all* calls to BLE `setValue` and `getValue` methods within an APK. We evaluate our tool against Amandroid and present a comparison in Table III.

BLECryptracer is developed on top of Androguard[11], an open-source reverse-engineering tool, which decompiles an APK and enables analysis of its components. Our tool has two separate components: one to analyse characteristic writes (i.e., `setValue`) and the other to analyse reads (i.e., `getValue`), as the direction of tracing is different in the two cases.

Backtracing BLE writes: BLE writes use one of the `setValue` methods in Table II to first set the value that is to be written, before calling the method for performing the actual write. BLECryptracer identifies all calls to these methods, and then traces the origins of the data held in the registers that

[11]https://github.com/androguard/androguard.

TABLE II. BLE DATA HANDLING METHODS

Access	Method [in the form <return> <method> (<args>)]	#Apps [#Unique Apps]	% of Total [% of Total Unique]
Read	byte[] getValue ()	12995 [6831]	94% [96%]
	Integer getIntValue (int formatType, int offset)	6006 [3447]	44% [48%]
	String getStringValue (int offset)	1576 [897]	11% [13%]
	Float getFloatValue (int formatType, int offset)	446 [245]	3% [3%]
Write	boolean setValue (byte[] value)	11454 [6147]	83% [86%]
	boolean setValue (int value, int formatType, int offset)	4180 [2805]	30% [39%]
	boolean setValue (String value)	390 [173]	3% [2%]
	boolean setValue (int mantissa, int exponent, int formatType, int offset)	303 [194]	2% [3%]

All methods are from the class `android.bluetooth.BluetoothGattCharacteristic`.

are passed as input to the methods. This technique of tracing register values is sometimes referred to as "slicing" and has been utilised in several static code analyses [19], [27], [28].

Considering the smali[12] code in Figure 5 as an example, `setValue` is invoked at Line 13 and is passed two registers as input. As `setValue` is an instance method, the first input, local register `v3`, holds the `BluetoothGattCharacteristic` object that the method is invoked on. The second input, parameter register `p2`, holds the data that is to be written to the BLE device, and is the second argument that is passed to the method `a` (Line 1).

BLECryptracer identifies `p2` as the register that holds the data of interest, and traces backward to determine if this data is the result of some cryptographic processing. To achieve this, the method(s) within the APK that invoke method `a` are identified, and the second input to each such method is traced. If the BLE data had come from a local register, rather than a parameter register, BLECryptracer would trace back *within* method `a`'s instructions, to determine the origin of the data. This backtracing is performed until either a crypto-library is referenced, or a `const-<>` or `new-array` declaration is encountered (which would indicate that no cryptography is used). Note that calls to any method within the crypto-libraries mentioned in Section V-B are accepted as evidence of the use of cryptography with BLE data. The tool stops processing an APK at the first instance where such a method call is identified.

During execution, the BLECryptracer maintains a list of registers (set within the context of a method) to be traced, for each `setValue` method call within the application code. This initially contains a single entry, which is the input to the `setValue` method. A new register is added to the list if it appears to have tainted the value of any of the registers already in the list. This could be due to simple operations such as `aget`, `aput` or `move-<>` (apart from `move-result` variants), or it could be as a result of a comparison, arithmetic or logic operation (in which case, the register holding the operand on which the operation is performed is added to the trace list). Similarly, if a register obtains a value from an instance field (via `sget` or `iget`), then all instances where that field is assigned a value are analysed. However, the script does not analyse the order in which the field is assigned values, as this would require activity life-cycle awareness.

Where a register is assigned a value that is output from a method invocation via `move-result`, if the method is not an external method, then the instructions within that method

```
1  .method public
     onCharacteristicRead(Landroid/bluetooth/
     BluetoothGatt;Landroid/bluetooth/
     BluetoothGattCharacteristic;I)V
2    ...
3    invoke-virtual {p2}, Landroid/bluetooth/
       BluetoothGattCharacteristic;->getValue()[B
4    move-result-object v0
5    new-instance v2, Ljava/lang/StringBuilder;
6    invoke-direct {v2},
       Ljava/lang/StringBuilder;-><init>()V
7    const-string v3, "Read value: "
8    invoke-virtual {v2, v3},
       Ljava/lang/StringBuilder;->append(Ljava/lang/
       String;)Ljava/lang/StringBuilder;
9    move-result-object v2
10   invoke-static {v0},
       Ljava/util/Arrays;->toString([B)Ljava/lang/
       String;
11   move-result-object v3
12   invoke-virtual {v2, v3},
       Ljava/lang/StringBuilder;->append(Ljava/lang/
       String;)Ljava/lang/StringBuilder;
13   move-result-object v2
14   ...
```

Fig. 6. Sample smali code for BLE attribute read.

are analysed, beginning with the return value and tracing backwards. In some instances, the actual source of a register's value is obfuscated due to the use of intermediate formatting functions. In an attempt to overcome this, BLECryptracer traces the inputs to called methods as well. Further, if a register is used as input to a method, then all other registers that are inputs to the method are also added to the trace list. While this captures some indirect value assignments, it runs the risk of false positives. For this reason, we have included the concept of *Confidence Levels* for the code output.

If, for an APK, the input to the `setValue` method can be backtraced directly, via only register value transfers and as immediate results of method invocations, and is found to be the result of some form of cryptographic processing, then a confidence level of "High" is assigned to the result. If a register cannot be traced back directly to a cryptographic output, but if an indirect trace identifies the use of a cryptography library, then a confidence level of "Medium" is assigned. Finally, in the event that no cryptography use is identified at High or Medium confidence levels, the script performs a less stringent search through all the instructions of the methods that it previously analysed. This risks including instances of cryptography use with functions unrelated to BLE and is therefore assigned a "Low" confidence level.

[12]Android applications are typically written in Java and converted into Dalvik bytecode. The smali format can be considered an "intermediate" step between the high-level Java source and the bytecode.

TABLE III. TESTS AGAINST BENCHMARKING APPS

		BLECryptracer		Amandroid	
Test Case	DroidBench Source	Write	Read	Write	Read
Direct use	N/A	✓	✓	✓	✓
Aliasing	Merge1	✓	✓	✓	✓
Format change	N/A	✓	✓		
Reflection	Reflection1	✓	✓	✓	✓
Inter-Activity	ActivityCommunication1	✓	✓	✓	✓
Intents	ActivityCommunication2	✓	✓	✓	✓
Threading	AsyncTask1			✓	✓

[a] The "Direct use" test case passes the input/output byte array of a cryptographic operation directly to/from a BLE method.
[b] The "Format change" test application applies intermediate formatting functions to the data (e.g., converting to string and then to byte array) before passing it between a cryptographic function and a BLE operation.

Forward-tracing BLE reads: With BLE reads, a `getValue` variant is invoked and the output, i.e., the value that is read, is moved to a register. To trace this value, BLECryptracer identifies all calls to `getValue` variants, then traces the output registers and all registers they taint until either a crypto-library is referenced or the register value changes. Such value changes can occur due to `new-array`, `new-instance` and `const` declarations, as well as by being assigned the output of various operations (such as method invocations or arithmetic/logic operations).

With forward-tracing, the register holding the BLE data is considered to taint another if, for example, the source register is used in a method invocation, or comparison/arithmetic/logic operation, whose result is assigned to the destination register. The destination register is then added to the trace list. When a register is used as input to a method, then along with the output of that method, the use of the register *within* the method is also analysed.

This method of analysis tends to result in a "tree" of traces. As an example, considering the smali code in Figure 6, the byte array output from the BLE read is stored in register `v0` (Line 4). This taints register `v3` via a format conversion function (Lines 10 and 11), which in turn taints `v2` via a `java.lang.StringBuilder` function (Lines 12 and 13). At this point, all three registers are tainted and will be traced until their values change.

The forward-tracing mode also assigns one of three confidence levels to its output. "High" is assigned when cryptographically-processed data is identified via the tracing mechanism above; "Medium" is when the use of cryptography is identified by tracing classes that implement interfaces. "Low" is similar to the backtracing case, and is assigned when a less stringent search through all instructions (of all encountered methods) results in identification of a reference to a cryptography library.

Intent handling: When analysing BLE writes, if a register of interest obtains its value from one of the `getExtra` variants within the `android.content.Intent` class[13], then the corresponding `putExtra` method is identified and its input is traced. This is subject to the constraint that the string identifier for the Intent is initialised as a `const-string` (opcode `0x1A`) within the method that invokes `getExtra`/`putExtra`. With BLE reads, if the register value is included as an Extra within an Intent, then the corresponding `getExtra` invocation is identified (subject to the same constraints) and the register that is assigned the value is added to the trace list.

[13] We include only those `getExtra` variants that accept a string identifier as an argument.

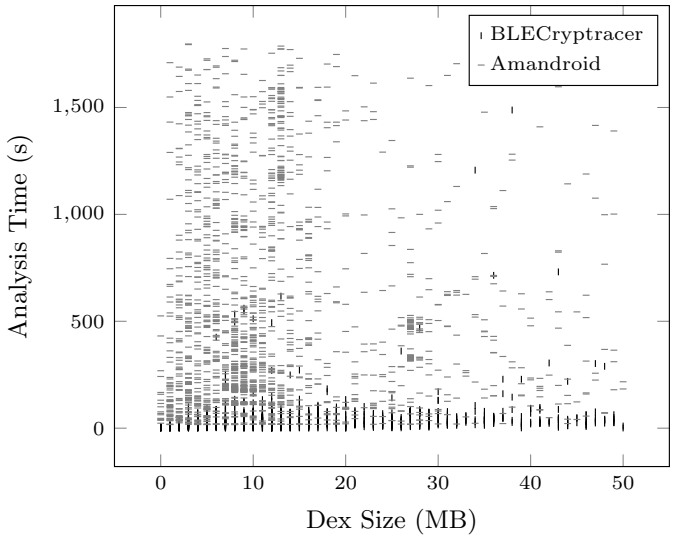


Fig. 7. Comparison of time taken to execute Amandroid and BLECryptracer (both when analysing BLE writes) against a random subset of approximately 2000 APKs (for readability), plotted against the size of the .dex file.

Testing: In order to test our tool against different data transfer or modification mechanisms, we developed a set of sample applications which transport data between the relevant source and sink (i.e., from the output of a cryptography method invocation to `setValue`; or from `getValue` to the input of a cryptography method invocation) in a variety of ways. Most of the test apps were based on the DroidBench[14] benchmarking applications, but were modified for the BLE use case. Two apps were developed per data transfer mechanism: one for BLE Reads and the other for BLE Writes. The same applications were also tested against Amandroid for comparison.

Table III depicts the results of testing the sample applications against our tool and Amandroid. While BLECryptracer does not identify instances where data processing is handled in a separate thread, it fares better than Amandroid when the data goes through multiple formatting functions. Also, we believe that, for efficiency and in the context of BLE, most applications will process data travelling between a source and sink within the same thread, thereby making that use case less likely.

Source code: The source code for BLECryptracer, as well as the application code for the test applications, is available at https://github.com/projectbtle/BLECryptracer.

Execution environment: We executed BLECryptracer on a virtualised server running Ubuntu 16.04, with 32GB RAM and 2.10GHz octa-core processor. To take advantage of the multiple processor cores, BLECryptracer employs multi-processing to parallelise analyses. As with Amandroid, a maximum analysis time of 30 minutes was imposed per APK. We found that this limit was exceeded for 26 applications out of 13,787. Figure 7 plots the time taken to execute BLECryptracer (when

[14] https://github.com/secure-software-engineering/DroidBench.

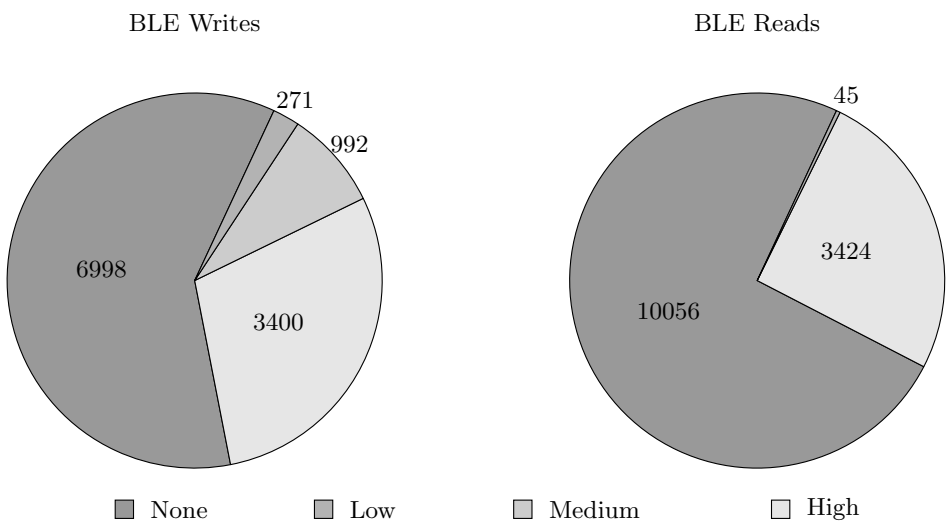


Fig. 8. Analysis results showing usage of cryptographically-processed data with BLE Writes and Reads, with breakdown according to Confidence Level. Only those APKs that invoke calls to the relevant methods (setValue or getValue) have been included in each individual pie chart.

analysing BLE writes) against the size of the application's dex file. The figure shows that analysis times were, for the most part, around 3-4 minutes per application regardless of the size of the dex file. APKs that took longer to process have confidence levels of "Medium" and "Low", which indicates that the longer analysis times may simply have been because of having to first go through the most stringent analysis (at the highest confidence level). For comparison, the same figure contains the analysis times for Amandroid as well, for the same set of applications. The graph shows that, for Amandroid, the execution times vary to a greater extent than with our script, due to the difference in the mechanisms employed for performing the analysis.

D. Results & Discussion

Figure 8 summarises the results obtained by BLECryptracer. From our analyses, we make the following observations.

Prevalence of application-layer security with BLE: Our analysis shows that approximately 95% of BLE-enabled applications call the `javax.crypto` and `java.security` cryptography libraries somewhere within their code. This suggests that about 500 applications do not implement any form of standard cryptographic-based protection anywhere within their code. Further, while 95% appears to be a fairly large proportion of applications with cryptography, the results also indicate that a much smaller percentage use cryptographically processed data with BLE reads and writes (approximately 26% and 40%, respectively).

Influence of third-party libraries: We found that many BLE-enabled APKs actually use external libraries for incorporating BLE functionality. In particular, variants of a single library for creating BLE beacon-enabled[15] Android applications were responsible for more than 70% of all identified instances of application-layer security with BLE writes and 96% of all identified cryptographically-processed BLE reads. An analysis of this library suggested that cryptography is being used to authenticate requests to modify settings on the beacon. Only 1380 APKs that were identified as having cryptographically-processed data did *not* make use of this library, and of these, only 164 were identified with a High confidence level (for either reads or writes). Of these, 6 use the public URIBeacon library from Google (https://github.com/google/uribeacon). This library allows beacons to broadcast URLs, allowing users to access the Web through physical interactions. We did not find any obvious security issues in either of these beacon libraries.

Of the remaining APKs, we found a set of about 20 parking applications from a single developer, which leak the keys used for encryption to the device logs. In addition, we also found a group of ten applications which share a BLE library that has an Over The Air (OTA) update mechanism with hardcoded keys. These applications are designed for use with a glucose meter, a pollution meter and smart lightning devices. Surprisingly, we also saw a UV control (tanning) application that makes use of the library, although the user interface doesn't make any reference to the use of BLE enabled devices.

Application-specific implementations: Our results showed that, of the applications that were identified as having cryptographically-processed data, very few rely on application-specific BLE implementations, i.e., without the use of external libraries. Analysing the use of cryptography within such applications (from the group of 164 APKs), we found that four applications use hardcoded keys to encrypt and decrypt data from the BLE device. These included a fitness tracker, a beacon-based social network, a laser distance meter and a fleet tracking application. We also found four applications with no obvious mistakes. This group included a printer companion, an electronic cigarette controller, a reader for a smart pH meter and a smart lock application.

Protection for writes vs. reads: Our results show that, of the 11,515 applications that called both BLE read and write functions, about 60% don't appear to be using cryptographically-processed data with either type of data access. They also show that, on average, it is more likely for

[15]A BLE beacon is a one-way broadcasting device that enables a receiving application to perform location-based actions.

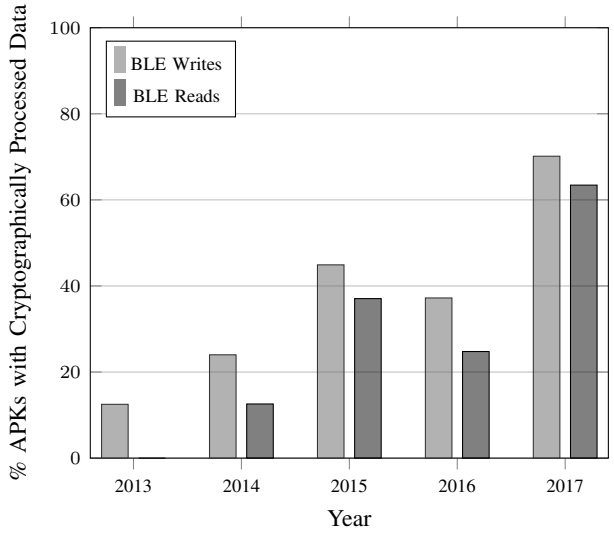


Fig. 9. Application-layer security trends over time. *Note 1:* The year of an APK is obtained from the dex date field within the AndroZoo source list. Some dates may be invalid or manipulated [Ref: https://androzoo.uni.lu/lists]. Dates prior to 2012 are likely invalid, as in-built Android platform support for BLE was only introduced that year. Such APKs have therefore been excluded.

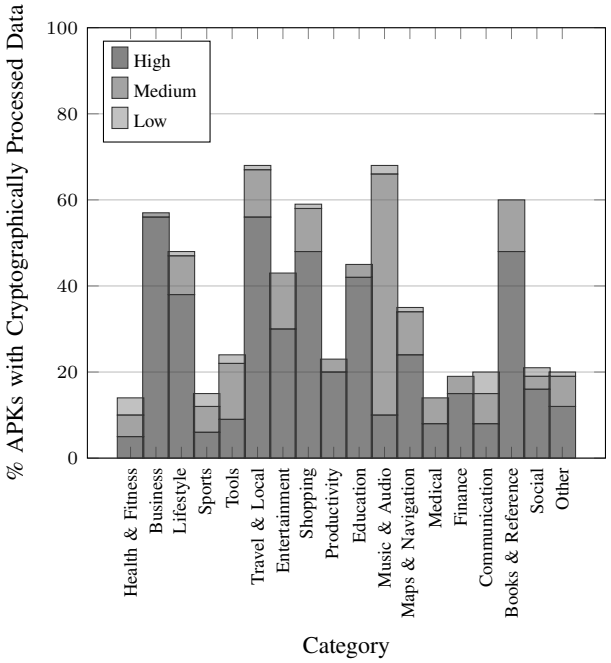


Fig. 10. Presence of application-layer security in different categories of applications, averaged over BLE data reads and writes and broken down by confidence level. All APKs, rather than just unique applications, have been taken into consideration, but applications that don't currently have a presence on Google Play have been excluded, as their category cannot be identified.

cryptography to be used with data writes (40% of all APKs and 47% of unique applications that call `setValue`) when compared with reads (26% of all APKs and 34% of unique applications that call `getValue`). The result is unsurprising, because while privacy and the ability for unauthorised applications to read BLE data is a concern, the integrity of the data is paramount. Also, in some cases, the data that is read from a BLE device may not be sensitive, e.g., a device's battery level, whereas data that is written is generally more so. Further, given that cryptographic processing results in overheads in terms of transmitted data and processing power, both of which will have an adverse impact on a resource-constrained BLE peripheral, prioritising the protection of data writes may have been considered an acceptable trade-off between security and resource conservation.

Trends over time: Figure 9 shows the trend of application-layer security over time. The results seem positive, as there is an overall upward trend for both reads and writes. However, older applications, which do not employ application-layer security, may still be operational on users' devices. Further, more than 30% of the applications that were released in 2017 (some with install counts exceeding several million) still do not appear to implement application-layer security.

Application-layer security by category: The percentage of applications that use cryptographically processed data from each major application category has been graphed in Figure 10. While it would be reasonable to expect that most "Medical" and "Health & Fitness" applications would implement some level of application-layer security, the results show that less than 15% of applications under these categories actually have such protection mechanisms. Perhaps surprisingly, applications that are categorised under "Business", "Shopping", "Music & Audio", "Books & Reference" and "Travel & Local" appear to be the most likely to incorporate application-layer security, with over 50% of all such applications being identified as having cryptographically processed BLE data. However, a closer inspection revealed that more than 75% of such occurrences were due to the external beacon library identified previously.

Comparison with Amandroid: When comparing our result against those obtained from Amandroid, we found that BLECryptracer identifies more instances of cryptographically-processed BLE data than Amandroid, for both reads and writes. A large percentage of these were due to the beacon library mentioned in Section V-C, which we manually verified as having cryptography-based application-layer security. Of the applications that did not use this library, manual verification was used to confirm all results that indicated a High confidence level with BLE writes. When analysing a result that was assigned a Medium confidence level by BLECryptracer and which was not identified by Amandroid, we found that the data underwent multiple formatting changes between being output from a crypto-function and prior to being used in the `setValue` method. We believe this to be the reason for it not being identified by Amandroid, which is in accordance with our findings with the test applications (Table III).

There were two applications for which Amandroid identified application-layer security with BLE writes when BLECryptracer did not. However, manual verification showed that one such result was a false positive, as only hardcoded strings were being used as the write values. The other application applied cryptography within one branch of a conditional statement and not the other. This use case is not currently identified by BLECryptracer, as explained in Section V-F. Amandroid did not discover any instances of application-layer security with BLE reads that were not also identified by BLECryptracer.

Impact analysis: While 13,787 BLE-enabled applications may seem like a relatively small number of applications when compared with the initial dataset of over 4 million, especially given that the number of unique packages is even smaller, it should be noted that the nature of BLE is such that a single application may correspond to multiple BLE devices, sometimes even millions of devices as in the case of fitness trackers[16]. This makes the attack surface much larger.

E. Case Study: Firmware Update over BLE

During our analysis, we found that one of the applications flagged as not having application-layer security was used to control a device that was part of our test device set. The test device was a low-cost fitness tracker, which, based on the install count (1,000,000+) on Google Play, appears to be widely used. An analysis of the APK suggested that the device could be put into a Device Firmware Update (DFU) mode. In particular, this device uses a legacy DFU mode which does not require the firmware to be signed. To exploit this, we developed an application that, in accordance with the attacks described in Section IV, connects to the device, sends commands to place it in DFU mode, and then writes a new modified firmware to the device without user intervention. The updated firmware in this case was a simple, innocuous modification of the original firmware. However, given that the device can be configured to receive notifications from other applications, a malicious firmware could be developed in such a way that all notifications (including second-factor authentication SMS messages or end-to-end encrypted messages) end up going back to the malicious application that installed the firmware. For devices without access to sensitive information, an attacker could even install a firmware that blocks the device until a ransom is paid.

While our attack was crafted for a specific device, it does demonstrate that attacks against these types of devices are relatively easy. An attacker could easily embed within a single application several firmwares to target a range of vulnerable devices, thereby increasing the chances of succeeding.

F. Limitations

In this section, we outline some limitations, either in our script or due to the inherent nature of our experiments, that may have impacted our results.

Application dataset: The APK dataset we used in our experiments was from 20 Feb 2018 and contained only applications from Google Play. This of course means that there will be a number of newer applications that were not included in our analysis. BLECryptracer is open-source and available on Github, and can be used to reproduce our experiments with newly published APKs or datasets from other markets.

Lack of ground truth: At present, there is no dataset of real-world APKs with known use of cryptographically-processed BLE data, i.e., ground truth. However, APKs flagged by Amandroid were also flagged by BLECryptracer (except for the two previously-mentioned cases). Furthermore, BLECryptracer was able to identify additional APKs using cryptographically processed data for BLE transfers. APKs flagged by BLECryptracer with High confidence were manually verified and, in some cases, even resulted in the identification of issues on the usage of cryptography. Our analysis has resulted in a dataset that we believe can be used as a basis for ground truth regarding the use (but not correctness) of cryptography with BLE for future research. A list of these APKs is available on our GitHub repository. We leave the detailed correctness analysis of cryptographic implementations as future work.

External data transfers: It is possible that an application obtains the data to be written to a BLE device from, or forwards the data read from a BLE device to, another entity, such as a server that is accessible via a network. That is, the Android application could merely act as a "shuttle" for the data, which means that an analysis of the APK would not show evidence of any usage of cryptography libraries. However, the transfer of data to and from a remote server does not in itself indicate cryptographically-processed data, as plain-text values can also be transmitted in the same manner. We therefore do not analyse instances of data transfers to external entities. Further, BLECryptracer does not analyse data that is written out to file, handled by a separate thread, or communicated out to a different application, because we consider these mechanisms to be less likely to be used when handling BLE data.

Conditional statements with backtracing: As mentioned in Section V-C, when backtracing a register, the script stops when it encounters a constant value assignment. However, it is possible that this value assignment occurs within one branch of a conditional jump, which means that another possible value could be contained within another branch further up the instruction list. To identify this, the script would have to first trace forward within the instruction list, identify all possible conditional jumps, and then trace back from the register of interest for all branches. This would need to be performed for every method that is analysed and could result in a much longer processing time per APK file, as well as potentially unnecessary overheads. In our analysis of 13,787 APKs we identified a single APK matching this condition, which was highlighted by Amandroid.

VI. Conclusions

In this paper, we present the risks posed to data on Bluetooth Low Energy devices from co-located Android applications. We show that, once an authorised Android application pairs and bonds with a BLE peripheral, any co-located Android application will also be able to access potentially sensitive, pairing-protected data from the peripheral. In some cases, an unauthorised application may be able to access such protected data with fewer permissions required of it than would be required of an authorised application, and without the user being aware of the data access. We describe the attacks and the conditions required for them, and discuss mitigation strategies. We also present the results of a large-scale automated analysis, using custom-built tools, of 13,787 BLE-enabled Android APKs, filtered from a dataset of 4 million APKs.

At present, our results suggest that approximately 500 BLE-enabled applications make no reference to standard cryptography libraries anywhere within their code. Further,

[16]https://www.idc.com/getdoc.jsp?containerId=prUS43260217 [accessed 16 Feb 2018].

over 60% of all applications, and about 85% of "Medical" and "Health & Fitness" applications, do not implement cryptography-based application-layer security for BLE data. We also found that some applications that do use cryptographically processed data for BLE, do it in the wrong way, leaking passwords to the smartphone logs or hardcoding the key values. We believe that if this situation does not change, then as more and more sensitive use cases are proposed for BLE, the amount of private or critical data that may be vulnerable to unauthorised access can only increase. We hope that our work increases awareness of this issue and prompts changes by application developers and operating system vendors, to lead to improved protection for BLE data.

REFERENCES

[1] M. Ryan, "Bluetooth: With low energy comes low security," in *7th USENIX Workshop on Offensive Technologies, WOOT '13, Washington, D.C., USA, August 13, 2013*, 2013.

[2] M. Elkhodr, S. Shahrestani, and H. Cheung, "Emerging wireless technologies in the Internet of Things: A comparative study," *International Journal of Wireless & Mobile Networks (IJWMN)*, vol. 8, no. 5, pp. 67–82, Oct 2016.

[3] C. Gomez, J. Oller, and J. Paradells, "Overview and evaluation of Bluetooth Low Energy: An emerging low-power wireless technology," *Sensors (Basel, Switzerland)*, vol. 12, no. 9, pp. 11 734–11 753, 2012.

[4] I. Bisio, A. Sciarrone, and S. Zappatore, "A new asset tracking architecture integrating RFID, Bluetooth Low Energy tags and ad hoc smartphone applications," *Pervasive and Mobile Computing*, vol. 31, pp. 79–93, 2016.

[5] W. Bronzi, R. Frank, G. Castignani, and T. Engel, "Bluetooth Low Energy performance and robustness analysis for inter-vehicular communications," *Ad Hoc Netw.*, vol. 37, no. P1, pp. 76–86, Feb 2016.

[6] R. Karani, S. Dhote, N. Khanduri, A. Srinivasan, R. Sawant, G. Gore, and J. Joshi, "Implementation and Design Issues for Using Bluetooth Low Energy in Passive Keyless Entry Systems," in *India Conference (INDICON), 2016 IEEE Annual*. IEEE, 2016, pp. 1–6.

[7] 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 *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*, 2014.

[8] "Bluetooth core specification," Bluetooth Special Interest Group, Dec 2016, ver 5.

[9] "Bluetooth low energy overview," Android, Apr 2018, [Online]. Available: https://developer.android.com/guide/topics/connectivity/bluetooth-le. [Accessed: 18 July 2018].

[10] "ScanFilter," Android, June 2018, [Online]. Available: https://developer.android.com/reference/android/bluetooth/le/ScanFilter. [Accessed: 18 July 2018].

[11] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner, "Android permissions: User attention, comprehension, and behavior," in *Proceedings of the eighth symposium on usable privacy and security*. ACM, 2012, p. 3.

[12] 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.

[13] K. Fawaz, K.-H. Kim, and K. G. Shin, "Protecting privacy of BLE device users," in *USENIX Security Symposium*, 2016, pp. 1205–1221.

[14] A. Korolova and V. Sharma, "Cross-app tracking via nearby Bluetooth Low Energy devices," in *PrivacyCon 2017*. Federal Trade Commission, 2017.

[15] B. Cyr, W. Horn, D. Miao, and M. Specter, "Security analysis of wearable fitness devices (Fitbit)," *Massachusetts Institute of Technology*, p. 1, 2014.

[16] "Distribution dashboard," Android, [Online]. Available: https://developer.android.com/about/dashboards/. [Accessed: 06 Aug 2018].

[17] G. Suarez-Tangil, J. E. Tapiador, P. Peris-Lopez, and A. Ribagorda, "Evolution, detection and analysis of malware for smart devices," *IEEE Communications Surveys Tutorials*, vol. 16, no. 2, pp. 961–987, Second 2014.

[18] Nordic Semiconductor. BLE on Android v1.0.1. [Online]. Available: https://devzone.nordicsemi.com/attachment/bdd561ff56924e10ea78057b91c5c642. [Accessed: 05 Feb 2018].

[19] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, "An empirical study of cryptographic misuse in Android applications," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013, pp. 73–84.

[20] "Firmware Over the Air," ARM Ltd, 2016, [Online]. Available: https://docs.mbed.com/docs/ble-intros/en/master/Advanced/FOTA/. [Accessed: 21 July 2018].

[21] 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 13th International Conference on Mining Software Repositories*. ACM, 2016, pp. 468–471.

[22] "Java Cryptography Architecture (JCA) Reference Guide," Oracle, [Online]. Available: https://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html. [Accessed: 18 July 2018].

[23] "Security tips," Android, June 2018, [Online]. Available: https://developer.android.com/training/articles/security-tips. [Accessed: 18 July 2018].

[24] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps," *Acm Sigplan Notices*, vol. 49, no. 6, pp. 259–269, 2014.

[25] F. Wei, S. Roy, X. Ou *et al.*, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of Android apps," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014, pp. 1329–1341.

[26] ——, "Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps," *ACM Transactions on Privacy and Security (TOPS)*, vol. 21, no. 3, p. 14, 2018.

[27] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna, "Execute this! analyzing unsafe and malicious dynamic code loading in Android applications." in *NDSS*, vol. 14, 2014, pp. 23–26.

[28] J. Hoffmann, M. Ussath, T. Holz, and M. Spreitzenbarth, "Slicing Droids: Program slicing for smali code," in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*. ACM, 2013, pp. 1844–1851.