IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2022, No. 3, pp. 330–366. DOI:10.46586/tches.v2022.i3.330-366

BreakMi: Reversing, Exploiting and Fixing Xiaomi Fitness Tracking Ecosystem

Marco Casagrande[1], Eleonora Losiouk[2], Mauro Conti[2], Mathias Payer[3] and Daniele Antonioli[1]

[1] EURECOM, Sophia Antipolis, France, {marco.casagrande,daniele.antonioli}@eurecom.fr
[2] University of Padua, Padua, Italy, {eleonora.losiouk,mauro.conti}@unipd.it
[3] EPFL, Lausanne, Switzerland,
mathias.payer@nebelwelt.net

Abstract.
Xiaomi is the leading company in the fitness tracking industry. Successful attacks on its fitness tracking ecosystem would result in severe consequences, including the loss of sensitive health and personal data. Despite these relevant risks, we know very little about the security mechanisms adopted by Xiaomi. In this work, we uncover them and show that they are insecure. In particular, Xiaomi protects its fitness tracking ecosystem with custom application-layer protocols spoken over insecure Bluetooth Low-Energy (BLE) connections (ignoring standard BLE security mechanisms already supported by their devices) and TLS connections. We identify severe vulnerabilities affecting such proprietary protocols, including unilateral and replayable authentication. Those issues are critical as they affect all Xiaomi trackers released since 2016 and up-to-date Xiaomi companion apps for Android and iOS. We show in practice how to exploit the identified vulnerabilities by presenting six impactful attacks. Four attacks enable to wirelessly impersonate any Xiaomi fitness tracker and companion app, man-in-the-middle (MitM) them, and eavesdrop on their communication. The other two attacks leverage a malicious Android application to remotely eavesdrop on data from a tracker and impersonate a Xiaomi fitness app. Overall, the attacks have a high impact as they can be used to exfiltrate and inject sensitive data from any Xiaomi tracker and compatible app. We propose five practical and low-overhead countermeasures to mitigate the presented vulnerabilities. Moreover, we present `breakmi`, a modular toolkit that we developed to automate our reverse-engineering process and attacks. `breakmi` understands Xiaomi application-layer proprietary protocols, reimplements Xiaomi security mechanisms, and automatically performs our attacks. We demonstrate that our toolkit can be generalized by extending it to be compatible with the Fitbit ecosystem. We will open-source `breakmi`.

Keywords: IoT, Reverse Engineering, Bluetooth Low Energy, Fitness Tracker

1 Introduction

Fitness tracking systems are complex and pervasive technologies used to monitor sensitive (health) data. They are composed of wearable devices connected to a mobile application acting as a gateway to cloud services. We have seen impactful security and privacy breaches affecting those systems. For example, researchers found backdoors in trackers' firmware [SJMdR16], managed to flash malicious firmware wirelessly [CWP+18], leaked private data, including health records and login credentials [RCT16], and disabled communication encryption [FCS+17]. It is not straightforward to fix these issues as vendors might not patch them at all, and fitness trackers might not support (secure) remote patching. Furthermore, vendors must distribute the software fix securely and on a large scale.

Received: 2022-01-15 Accepted: 2022-03-15 Published: 2022-06-08

Xiaomi is the worldwide fitness tracking leader. In 2020, Xiaomi sold 13.5 million devices and had 24.5% of the market share [Cor20]. Nevertheless, the Xiaomi fitness tracking ecosystem received little attention from security researchers, despite being pervasive and promising security and privacy guarantees to its users [Hua20]. Currently, there is only incomplete and outdated information about Xiaomi security mechanisms [HPK16, FFM+17]. On the other side, *Fitbit* (its main competitor) has received more consideration from security researchers. For example, recent work demonstrated that popular Fitbit devices are susceptible to attacks such as packet sniffing, data exfiltration, and code injection through firmware updates [RCB13, CHMS14, GDS16, SJMdR16, CWP+18].

In this work, we perform an extensive and up-to-date security evaluation of the Xiaomi fitness tracking ecosystem that is currently lacking. We analyze all Xiaomi trackers since 2016 (i.e., Mi Band 2/3/4/5/6 and Amazfit Cor 2) and up-to-date Xiaomi companion apps (Mi Fit and Zepp). Via extensive static and dynamic reverse-engineering experiments, we reconstruct Xiaomi's *proprietary* Pairing, Authentication, and Communication protocols used to connect trackers and apps via BLE. We find that these protocols are implemented at the application-layer over a BLE link-layer. Moreover, we discover that Xiaomi ignores standard BLE security mechanisms (e.g., BLE pairing and secure sessions), although its devices support them.

Then, we uncover *severe specification-level vulnerabilities* affecting the self-baked Xiaomi protocols. For example, keys are sent in cleartext, authentication is unilateral and replayable, and the BLE traffic is neither encrypted nor integrity protected. As the vulnerabilities target the protocols' design, they can be exploited *regardless of* the hardware and software details of the target (e.g., firmware, operating system, app, and BLE versions). Additionally, those issues might even be exploitable on other Xiaomi products sharing the same application-layer security mechanisms.

To demonstrate the impact of the presented vulnerabilities, we develop and evaluate *six practical attacks* on actual devices. With our over-the-air (OTA) attacks, an attacker in Bluetooth range with a victim can impersonate a tracker to an app, an app to a tracker, man-in-the-middle them, and eavesdrop on their communication. Alternatively, with our remote attacks, indicated in the paper as software-based (SB), an attacker can remotely eavesdrop on data from a tracker or impersonate an app by abusing Android BLE API within a malicious app. Our attacks are high impact because they affect the whole Xiaomi ecosystem and enable the attacker to achieve valuable goals such as eavesdropping on sensitive data exchanged by a tracker and a smartphone (e.g., health data, SMS, and notifications) or sending arbitrary commands to the tracker and the smartphone.

We developed `breakmi`, a security evaluation toolkit for fitness tracker ecosystems to automate our RE efforts and attacks. `breakmi` has three modules: protocol dissector, security mechanisms, and attacks. The protocol dissector module understands Xiaomi's proprietary application-layer protocols, allowing for fast and automated analysis of its application-layer packets. The security mechanisms module reimplements Xiaomi's custom security mechanisms, such as Pairing and Authentication. The attacks module deploys our attacks automatically, including over-the-air or remote impersonation and MitM on arbitrary trackers and apps. We will release `breakmi` in the open after responsible disclosure.

To show the effectiveness of our attacks, we present an *extensive evaluation* of Xiaomi trackers and apps. In particular, we successfully attacked all Xiaomi trackers released since 2016 (i.e., Mi Band 2/3/4/5/6 and Amazfit Cor 2) and the latest versions of the Xiaomi fitness mobile apps (i.e., Mi Fit version 4.8.1 and Zepp version 5.9.2). Two of the presented attacks are remotely targeting the Android platform. We successfully conduct them on six popular Android versions (i.e., Android 6/8/9/10/11/12) to confirm their widespread impact. According to [Sta21], these versions represent 90% of the Android ecosystem.

To effectively address the presented vulnerabilities and attacks, we propose *five* coun-

termeasures. All of them incur minimal overhead because they rely on a few additional messages and on lightweight security features. We redesign Xiaomi proprietary protocols to implement four out of the *five* proposed countermeasures at the application-layer. The fifth countermeasure applies to the link-layer, where we encourage the activation of the standard BLE link-layer security (i.e., BLE legacy pairing, LE Secure Connections), already supported by all Xiaomi devices.

To check the effectiveness of our attacks and the extensibility of `breakmi` to other vendors, we also analyzed the *Fitbit ecosystem* (the second biggest ecosystem after Xiaomi). We looked at two popular Fitbit trackers (i.e., Charge 2 and Charge 4) and the Fitbit mobile app (v 3.54.1). Our results show that the Fitbit ecosystem provides better (still proprietary) security mechanisms than Xiaomi. However, it is *still vulnerable* to five out of the six presented attacks. While testing our attacks on Fitbit, we extended `breakmi` by adding Fitbit's custom protocols and security mechanisms.

To encourage further research on the topic, we describe our reverse-engineer *methodology* in detail. Specifically, we used a mix of static and dynamic techniques for reconnaissance, traffic analysis, and app analysis. Additionally, we developed custom scripts and tools to automate our analyses that are now part of `breakmi`. Overall we spent a considerable time reversing the Xiaomi ecosystem (i.e., one year RE effort).

We summarize our contributions as follows:

- We reverse-engineer the proprietary security protocols used by Xiaomi to protect the BLE link between its trackers and companion apps. Those protocols include Pairing, Authentication, and Communication at the application-layer, and do not take advantage of BLE link-layer security mechanisms already supported by its devices.

- We uncover novel and severe vulnerabilities in the specification of those protocols enabling an attacker to target the ecosystem as a whole. The list of vulnerabilities includes unilateral and replayable authentication, improper key agreement, and lack of encryption and integrity protection of sensitive data.

- We show how to exploit these vulnerabilities, and we perform high-impact Xiaomi-compliant attacks either over-the-air or remotely (via a malicious app). We design and release `breakmi`, a toolkit to automatically analyze and attack the Xiaomi ecosystem. We address the presented vulnerabilities and attacks by proposing five practical and low overhead countermeasures that fix Xiaomi's vulnerable protocols.

- We compare Xiaomi with Fitbit, and we find that four of the identified vulnerabilities and five of the proposed attacks are portable to Fitbit. We extend `breakmi` to the Fitbit ecosystem, and we successfully conduct the attacks.

Responsible disclosure We responsibly disclosed our findings to Xiaomi in March 2021 via the HackerOne platform. Xiaomi considered our report as a single and known vulnerability, namely "lack of encryption," scheduled to be fixed on an undisclosed timeline. We disagree with this response as we reported multiple classes of vulnerabilities leading to several attacks (and not a single vulnerability)[1]. We also *responsibly disclosed* our findings to Fitbit in January 2022 through Google Vulnerability Reward Program, and Fitbit acknowledged our attacks and will deploy a fix in April 2022.

[1]Our experiments did *not* involve or expose third-party users, but we analyzed our own devices in a controlled environment.

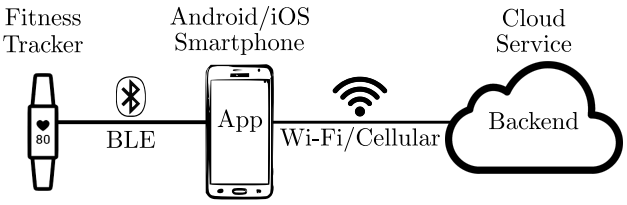

Figure 1: Xiaomi fitness tracking system architecture. The tracker is a battery-powered embedded device supporting BLE. The smartphone runs a fitness tracking application and is capable of communicating with the tracker via BLE and with a backend server via the Internet.

2 Background

In this section, we introduce BLE and what is known about the Xiaomi fitness tracking ecosystem.

2.1 Bluetooth Low Energy (BLE)

BLE is the de-facto standard wireless technology for low-power wireless services, including fitness tracking. It is defined in the Bluetooth standard [SIG19] and provides a client-server architecture to exchange data using a specific format. Security-wise, BLE includes *pairing* and *session establishment* mechanisms that should provide confidentiality, integrity, and authenticity guarantees at the link-layer.

The BLE client-server architecture is specified by the Generic Attribute Profile (GATT) and uses the Attribute Profile (ATT) protocol [SIG19, p. 1531]. In Bluetooth terminology, the client is defined as the *central*, and the server as the *peripheral*. The client sends read, write, and notification requests to the server. The server answers accordingly to the request type and the availability of data.

In a fitness tracking use case, the GATT server is the tracker device (e.g., wristband) and the GATT client is a smartphone application. The server exposes fitness data, such as heart rate and step count, while the client can periodically query such data. BLE data is exchanged using a hierarchical and object-oriented format defined in [SIG19, p. 284].

The top-level of the hierarchy is a profile, and it contains a set of services. Each service provides characteristics or other services. A characteristic provides a value field with optional fields, such as descriptors and properties. Each characteristic can be configured with access-control flags (e.g., read-only, write-only, or read-write).

BLE provides pairing and secure session establishment protocols to secure the link-layer. Before exchanging data over GATT, the client and the server can pair to agree upon a long-term pairing key and use it to establish a secure session (e.g., by using ECDH).

On the contrary, session establishment is implemented using AES-CCM authenticated-encryption, keyed with a fresh session key (derived from a pairing key). The server can protect a GATT characteristic by requiring a client to pair before accessing it by setting its encryption and authentication security permissions.

A BLE device supporting a Bluetooth version greater than or equal to 4.2 can support a security mode known as *Secure Connections (SC)* that enhances pairing and session establishment by only using Federal Information Processing Standards (FIPS) compliant algorithms.

2.2 Xiaomi Fitness Tracking Ecosystem

The Xiaomi fitness tracking ecosystem includes wearable tracking devices, smartphones running a tracker companion app, and backend infrastructure. As we see from Figure 1, the Xiaomi components communicate *wirelessly* using a combination of short-range and long-range technologies. The tracker and the smartphone use *BLE* (introduced in Section 2.1), while the smartphone and the backend require Internet connectivity through Wi-Fi or a cellular network. We note that other fitness tracker vendors, including Fitbit, employ the same general architecture.

Xiaomi trackers are wearable and battery-powered devices composed of sensors to collect health data, such as step count and heart rate, and actuators, such as buttons and a touch screen. They can also control the associated smartphone (e.g., lock/unlock the screen) and receive notifications (e.g., SMS, WhatsApp).

Xiaomi ships two families of trackers called *Mi Band (MB)* [Incf] and *Amazfit* [Inca]. Mi Band is the most popular family and so far includes *six* generations: MB 1 (2014), MB 2 (2016), MB 3 (2018), MB 4 (2019), MB 5 (2020), and MB 6 (2021). Amazfit has two generations: Cor 1 (2018) and Cor 2 (2019). The two families are manufactured by the same company (Huami [Incb]) and have similar hardware and software capabilities. For example, the Cor 1 is a MB 2 clone, and the Cor 2 clones the MB 3.

Xiaomi's official companion app is *Mi Fit* and is freely available for Android [Co.] and iOS [Incc]. The app provides a user interface to configure and manage a tracker and is compatible with all Mi Band and Amazfit Cor generations. Another official app that supports Xiaomi trackers is *Zepp*, available on Android [Incd] and iOS [Ince]. There are also several third-party apps compatible with Xiaomi.

The Xiaomi backend is an Internet-accessible infrastructure that manages several aspects of the ecosystem. It stores the list of registered users and their associated trackers. It also backups the configurations of the trackers and the apps. Additionally, it distributes firmware and resource file (e.g., fonts, images) updates to the trackers. The backend is managed by Huami, which is also the developer of the Mi Fit and Zepp applications (and the tracker's manufacturer).

On one hand, Xiaomi does not provide any information about its security architecture and mechanisms. However, on the other hand, it claims to guarantee its users' confidentiality, security, and privacy (see the Privacy Policy [Hua20] dated May 2020). Hence our work investigates how these claims are actually implemented in practice.

3 Analysis of Xiaomi Fitness Tracking

From our reverse-engineering experiments, we found that Xiaomi uses three *proprietary* application-layer protocols in three different operations: *Pairing*, *Authentication*, and *Communication*.

These three protocols define the format, and the purpose of any BLE packet exchanged between Xiaomi companion apps and trackers. A single vulnerability in the protocols can be used to exploit *any* supported Xiaomi tracker and app. Hence they must be well designed and implemented, but this work (experimentally) shows the contrary. We note that Xiaomi protocols are orthogonal to the link-layer security mechanisms provided by BLE introduced in Section 2.1.

In our experiments, we also uncover that Xiaomi *disables* BLE link-layer security and privacy features despite being supported by its trackers and apps. BLE security mechanisms were designed to protect the emerging IoT market, including the fitness tracking industry, and it is not evident why Xiaomi simply ignores them. This choice considerably increases the risk of a security breach as Xiaomi only trusts its vendor-specific protocols.

Now we describe these protocols in detail. The presented information required extensive

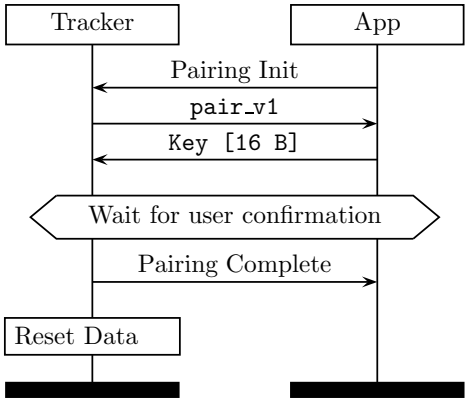

Figure 2: Xiaomi Pairing v1. The app generates and sends to the tracker a 16-byte pairing key (`Key`) in the clear. The tracker shows a pairing confirmation message to the user. Once the user confirms, the tracker resets its data, and pairing is completed.

reverse-engineering (RE) efforts described in Section 9. Then, we pinpoint vulnerabilities in their specification, including lack of mutual authentication and replay protection. The vulnerabilities are critical as they affect the whole Xiaomi ecosystem regardless of the hardware and software details of the trackers and the apps.

3.1 Reverse-Engineered Protocols

We isolate three Xiaomi proprietary application-layer protocols, and we name them Pairing, Authentication, and Communication. Pairing is used to establish a long-term secret (i.e., pairing key) between a tracker and an app. Authentication is employed to prove ownership of a pairing key. Communication runs only after a successful Authentication and enables interaction between trackers and an app.

Those interactions include sending health data from the tracker and sending commands or notifications from the app to the tracker or vice versa. Now we describe their technical details. We use the terms *tracker* and *app* to refer to any Xiaomi-compliant device, and, when needed, we indicate the specific tracker model or app name.

3.1.1 Pairing

Pairing is used to establish a 16-byte pairing key between the tracker and the app. The pairing key is the *root of trust* between the devices and must be kept secret and stored securely. We observed two versions of Pairing.

The first version, which we call Pairing *v1*, is used by Mi Band 2/3 and Amazfit Cor 1/2 trackers. Instead, Pairing *v2* is employed by Mi Band 4/5/6 and is server-based as it involves Xiaomi backend. Mi Fit and Zepp apps support both pairing versions. Now we describe the technical details of Pairing v1/v2.

Pairing v1 is supported by MB 2/3 and works shown in Figure 2. The app sends a pairing initialization message (`Pairing Init`). The tracker responds with a Pairing version message (`pair_v1`). The app generates and sends a 16-byte pairing key (`Key`) to the tracker *in the clear*. Then, the tracker shows the user a pairing confirmation message (see Mi Band 2/3 in Figure 3) and waits for user confirmation (together with the app). Once the user confirms pairing, the tracker sends a success message (`Pairing Complete`) to the app and resets its stored data, completing Pairing v1.

Xiaomi introduced Pairing v2 in 2019, and it is supported by MB 4/5/6. The protocol involves interactions with the Xiaomi *backend* using HTTPS. The protocol works as

Figure 3: Mi Bands pairing confirmation messages. To accept pairing, a user must either press a hardware button (Mi Band 2/3) or touch a software button (Mi Band 4/5/6). Note that Amazfit Cor 1/2 use similar pairing confirmation messages.

depicted in Figure 4. The app sends a Pairing Init message, and the tracker replies with a `pair_v2` and a truncated digest of its public key (SHA1(`pub_k`)). As we observed the same digest on all trackers that we tested (i.e., `1863c2cce5d159413bed92c4b163c279`), we are confident that all trackers are using the *same* public key.

Then, the app sends a random number request, and the tracker answers with `R`, a 16-byte random number. `R` and the tracker public Bluetooth address (`TR_A`) are inputs to a *custom* key derivation function (`kdf`) that generates the pairing key (`Key`). As such, `R` is the *pairing key seed* and is sent in the clear. We reverse-engineered `kdf` and discovered that it computes a SHA256 of the concatenation of `TR_A` and `R` and outputs `Key`, the leading 16 bytes of the digest. The function is expressed as:

$$\texttt{Key} = \texttt{kdf}(\texttt{TR\_A}, \texttt{R}) = \text{SHA256}(\texttt{TR\_A} \| \texttt{R})[0:16].$$

Next, the app and the backend establish a TLS session, and the app sends SHA1(`pub_k`) and the `Key` encoded as base64 to the backend. The backend computes a signature (`Sig`) of `Key` using its private key (`pri_k`) and sends the base64-encoded signature to the app (B64(`Sig`)). The app provides the signature to the tracker that verifies it and sends back an acceptance message (Valid `Sig`). Then, pairing completes identically to Pairing v1, with the user having to accept a pairing confirmation message (see Mi Band 4/5/6 in Figure 3).

3.1.2 Authentication

Authentication has only one version and works as depicted in Figure 5. The app sends an authentication request message (`Auth Req`). The tracker answers with a 16-byte challenge (`Chal`). The app computes a 16-byte response (`Resp`) by encrypting `Chal` with `Key` using AES in ECB mode and sends it to the tracker. The tracker computes its own response, checks it against `Resp`, and sends a positive authentication message (`Auth OK`) if `Resp` is verified. As a result, the tracker authenticates that the app owns the correct pairing key and unlocks access to its private data (e.g., step count). However, the tracker *never* authenticates to the app, and the authentication messages are sent in the clear.

3.1.3 Communication

Once a tracker and an app complete Pairing and Authentication, they run the Communication protocol to exchange data, commands, and notifications. We discovered that Communication *is neither encrypted nor integrity protected* despite the tracker and the app sharing a pairing key. This finding is surprising, as Xiaomi trackers and apps do support encryption primitives and crypto hardware acceleration. Moreover, prior authoritative reports, such as the ones from Mozilla [Fou20b, Fou20c, Fou20a], state that Xiaomi uses encryption (and meets Mozilla's minimum security standards) when this is not the case.

Communication is implemented on top of BLE GATT (introduced in Section 2.1). The tracker is the GATT server, and the app is the GATT client. The server has a set of public

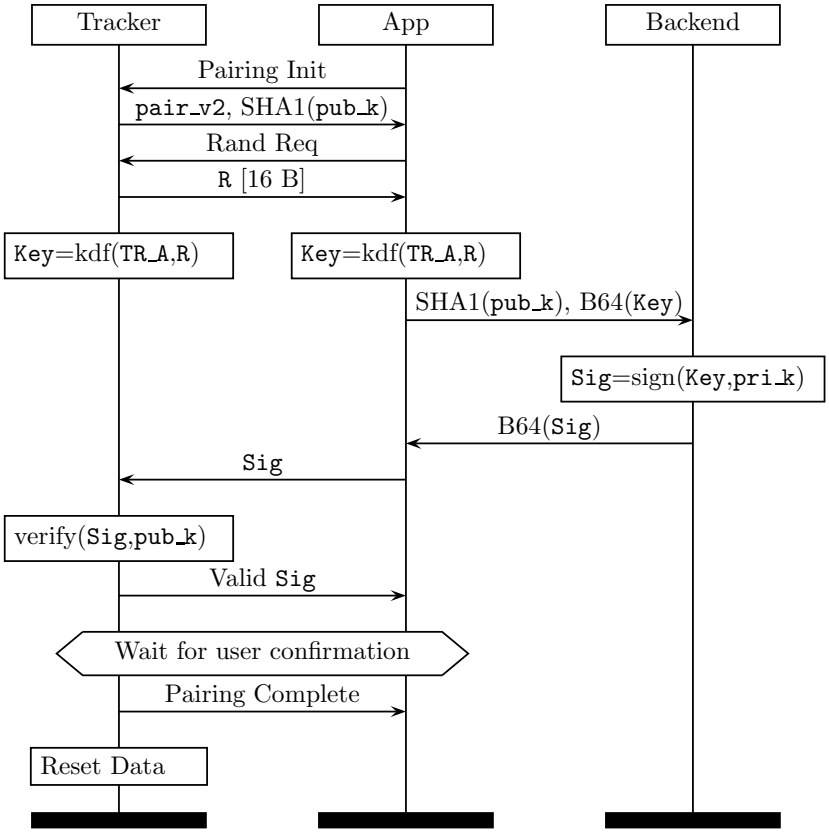

Figure 4: Xiaomi Pairing v2. The app starts the protocol by sending a Pairing Init message. The tracker sends back `pair_v2` and a public key digest (SHA1(`pub_k`)). Then the app requests a random number, and the tracker replies with `R`, which is 16 bytes long. Both devices run a custom key derivation function (`kdf`) to compute `Key` from `R` (key seed) and the Bluetooth address of the tracker (`TR_A`). Then, the app sends SHA1(`pub_k`) and `Key` base64-encoded to the backend. The backend computes a signature (`Sig`) of `Key` with its private key and sends `Sig` base64-encoded to the app. The app presents `Sig` to the tracker, which verifies it and sends back a confirmation message. Then, the tracker shows the user a pairing confirmation message, and if the user accepts, pairing is completed, and the tracker resets its data.

services (e.g., Generic Access) and characteristics (e.g., Device Name). Each characteristic has access control bits to set reading and writing permissions.

On top of GATT, Xiaomi uses a custom data locking mechanism where a tracker GATT characteristic cannot be accessed until the app has authenticated to the tracker (via a successful run of Authentication). Two examples of locked characteristics are heart rate and step count. For the list of GATT services and characteristics exposed by Mi Band 2/3/4/5/6, see Table 6 and Table 7 in the Appendix.

3.2 Protocol-level Vulnerabilities

We analyzed the Pairing, Authentication, and Communication protocols (described in Section 3.1), and we identified thirteen severe vulnerabilities in their *specifications*. Most of them, such as unilateral/replayable authentication and lack of encryption and integrity protection, were *publicly unknown*. The issues affect *all* trackers released since 2016, even

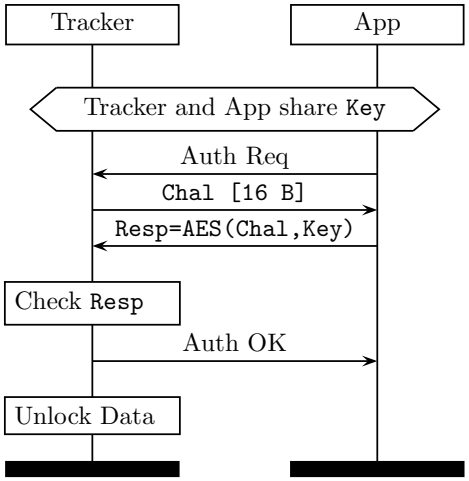

Figure 5: Xiaomi Authentication. Using a challenge-response procedure, the tracker *unilaterally* authenticates that the app owns the shared pairing key (`Key`).

the newest releases (i.e., MB 6) and the most recent app versions. We now describe the vulnerabilities grouped by protocol.

3.2.1 Pairing v1 (MB 2/3, AC 1/2)

- *Pairing key sent in the clear.* The app sends the pairing key to the tracker in the clear and with no integrity protection (see `Key` in Figure 2).
- *Pairing not authenticated.* The devices do not authenticate each other during Pairing. Hence the app and the tracker cannot determine if they are pairing with a legitimate device.
- *Pairing key generated by the app.* The pairing key is generated and distributed by the app, despite the tracker being able to run a key agreement protocol (e.g., Elliptic-curve Diffie–Hellman (ECDH)).
- *Weak user confirmation.* As shown in Figure 3, pairing confirmation is weak as the user only has to interact with the tracker, and the tracker does not show any contextual information.

3.2.2 Pairing v2 (MB 4/5/6)

- *Pairing key seed sent in the clear.* The tracker sends the pairing key seed to the app in the clear and with no integrity protection (see `R` in Figure 4). This issue is as bad as in Pairing v1, as the pairing key is deterministically computed from its seed and its publicly available information (i.e., the tracker's Bluetooth address).
- *Pairing only (weakly) authenticates the app.* Pairing v2 does not authenticate the tracker to the app but authenticates the app to the tracker. In particular, the tracker must receive a correct signature from the app. The signature algorithm is deterministic, and both inputs are public, so an attacker could reverse the algorithm and obtain the signature.
- *Pairing version can be downgraded/upgraded.* The tracker decides the version of Pairing by sending either `pair_v1` or `pair_v2`. As this message is not integrity

protected, it can be manipulated to force a specific pairing version (e.g., downgrade from v2 to v1 or upgrade from v1 to v2).

- *Pairing key generated by the tracker.* The pairing key only depends on the tracker, despite the app being able to run a key agreement protocol (e.g., ECDH). This issue is worse than the one highlighted for Pairing v1 key generation. The tracker is a computationally-constrained device and is more likely to generate a low-entropy key than an app running on a smartphone.

- *Default keypair.* The hash of the public key is the same for all MB 4/5/6 that we tested, and this entails that all our trackers share the same public key. Our device sample is limited, but we found the same key even across devices bought in different countries. This implementation is risky because an attacker can compromise the whole ecosystem by leaking the default private key.

- *Weak user confirmation.* As shown in Figure 3, pairing confirmation is weak for the same reasons as Pairing v1.

3.2.3 Authentication

- *Unilateral app authentication.* The protocol unilaterally authenticates the app (see `Resp` in Figure 5), but it does not require to authenticate the tracker. Indeed, there is no way for an app to check if it is connected with a legitimate tracker.

- *Replayable authentication.* The protocol is vulnerable to replay attacks as, given a fixed challenge, there is no way to generate different responses (i.e., there is no nonce). Hence a fitness tracker cannot be certain that a valid response comes from a trusted app.

3.2.4 Communication

- *No encryption.* Despite sharing a pairing key and supporting encryption algorithms, the tracker and the app do not encrypt their sessions. Hence sensitive data exchanged over BLE can be effortlessly obtained.

- *No integrity protection.* Despite sharing a pairing key and supporting Message Authentication Codes (MAC), the tracker and the app do not integrity-protect their communication. As a result, sensitive data can be manipulated at will.

4 Proposed Attacks

We now describe six attacks to demonstrate the severity of the issues presented in Section 3.2. As the attacks exploit *architectural* vulnerabilities in the Xiaomi protocols, they are effective on all devices employing those protocols. Developing the attacks required extensive RE efforts as we target proprietary and unknown protocols (unlike BLE pairing and session establishment). We discuss four over-the-air attacks and two software-based attacks.

Our OTA tracker impersonation, OTA app impersonation, and OTA MitM require proximity with the target, as they exploit BLE traffic and minimal equipment.

Our SB app impersonation and SB eavesdropping are *remote* and require the installation of a malicious app on the victim's phone. The app only asks for Internet and Bluetooth normal permissions (*no root access*) and abuses Android BLE API to interfere with BLE traffic. Next, we present our threat model, describe the attacks, discuss how each attack maps to the vulnerabilities presented earlier and their impact.

4.1 System Model

Our system model has the same architecture presented in Section 2.2 and depicted in Figure 1. There are three entities: a tracker, a companion app, and a backend. The tracker and the app communicate over BLE, and the app communicates with the backend via Wi-Fi or a cellular network. The entities use the strongest security mechanisms at their disposal.

For example, the communication between the tracker and the app is protected using Xiaomi Pairing, Authentication, and Communication protocols that we reverse-engineered and described in Section 3.1. Similarly, the app and the backend use TLS. Such mechanisms should protect against passive and active attacks, including eavesdropping, device impersonation, and MitM attacks.

Our victim is a user of the Xiaomi ecosystem. She might use any supported trackers, such as Mi Band (MB) 2/3/4/5/6 and Amazfit Cor (AC) 1/2, and any version of the Mi Fit and Zepp companion apps. We assume that the victim has installed the app, registered an account with the Xiaomi backend, and paired her tracker with the smartphone app. Hence the user can establish authenticated sessions between the tracker and the app.

4.2 Attacker Model

Our attacker targets Xiaomi Pairing, Authentication, and Communication protocols as the vulnerabilities in these protocols can be exploited regardless of the hardware and software details of the target tracker and app. In other words, she is looking for *Xiaomi-compliant* vulnerabilities.

The attacker only knows public information advertised by the tracker over BLE (e.g., the public BLE address of the tracker), and she has no physical access to the target devices. Hence the attacker does not know any pre-shared secret between the victims (e.g., pairing keys) and cannot tamper with the devices' operating system and firmware.

The attacker has *four* goals: (i) she aims at impersonating the tracker to the app and (ii) the app to the tracker; (iii) she wants to establish a MitM position between the tracker and the app; (iv) she desires to eavesdrop the data exchanged between the tracker and the app.

The attacker can use *over-the-air (OTA)* or *software-based (SB)* attacks. Our attacker model is based on the Android threat model proposed by Mayrhofer et al. [MSBK21]. This threat model labels our OTA attacks as both *Proximal Access* and *Network-level* threats. In particular, our OTA attacks involve *T.P1 - Devices in physical proximity, but not under direct control, of an attacker who can control radio communication channels, including BLE)*, *T.N1 - Passive eavesdropping and traffic analysis*, and *T.N2 - Active manipulation of network traffic*. The attacker can sniff BLE traffic, jam the BLE spectrum, craft, and send custom BLE packets to the app and the tracker.

The same threat model labels our SB attacks as *Application Code* threats. In particular, our SB attacks involve *T.A1 - Abusing APIs supported by the OS*. The attacker can remotely attack the victim through a malicious app already installed on the victim's smartphone, a common requirement for most Android malware [Lak21b, Lak21a, Lak21c]. This requirement is reasonable as users often install unwanted apps on their smartphones fairly often. Kotzias et al. [KCB21] estimated that 67% of unwanted apps are directly installed from the Google Play Store or alternative markets (10.4%). When launched, our malicious app stealthily abuses Android BLE API. It only requires normal permissions related to the Internet and Bluetooth and does not need root privileges.

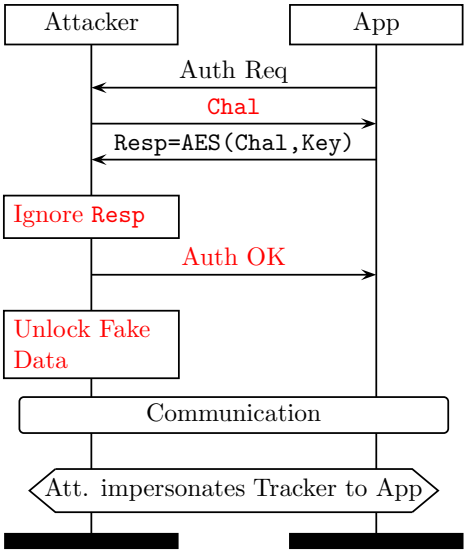

Figure 6: OTA tracker impersonation attack. The attacker impersonates a tracker by spoofing the tracker's BLE address and BLE advertisement packets. The victim (Mi Fit or Zepp app), which is already paired with the impersonated tracker, recognizes the attacker as trusted and sends her an authentication request (Auth Req). The attacker's device answers with a random challenge (`Chal`). The app solves it and sends back a response (`Resp`) computed from the pairing key unknown to the attacker. The attacker ignores `Resp`, answers with a positive authentication message (`Auth OK`), and unlocks its fake GATT server. As a result, the app starts the Communication protocol with the impersonated tracker, believing it is trusted.

4.3 OTA Tracker Impersonation Attack

The attacker can wirelessly impersonate any tracker by presenting itself to a victim app as a spoofed tracker, running the unilateral Authentication protocol without having to authenticate, and then starting a Communication session that is neither encrypted nor integrity protected. The attack does not require knowledge of the pairing key, does not trigger Pairing (which requires user interaction), and can be launched anytime a target app is in BLE range with the attacker. The attack leverages Xiaomi's unilateral authentication and the lack of encryption and integrity protection of Communication.

The technical details of the attacks are presented in Figure 6. The attacker advertises her presence as the impersonated tracker by copying its features, including its Bluetooth address and GATT server. The victim (App) recognizes the attacker as trusted and sends her an authentication request (`Auth Req`). The attacker answers with a random challenge (`Chal`), and the app computes and sends back a response (`Resp`) derived from a pairing key unknown to the attacker. The attacker ignores `Resp`, answers with a positive authentication message (`Auth OK`), and unlocks her own GATT server. Afterward, during Communication, the app considers the impersonated tracker as trusted.

4.4 OTA App Impersonation and MitM Attacks

The attacker can impersonate any app over-the-air or MitM an app and a tracker using a replay attack on the non replay-protected Authentication protocol and can start an insecure Communication session.

In particular, the attacker can start Authentication in parallel with the app and the

tracker. When the tracker sends a challenge, the attacker replays it to the app, relays the app response to the tracker, and successfully authenticates to the tracker without knowing the pairing key. Then, the attacker can either impersonate the app by dropping her connection with the legitimate app and starting a Communication session with the tracker or MitM the Communication session between the app and the tracker.

The attacks do not require knowledge of the pairing key and do not trigger Pairing. Unlike the OTA tracker impersonation attack, these ones require both the app and the tracker to be in BLE range with the attacker. This issue is not that significant as the tracker and the app are typically carried and used by the same person.

The technical details of the OTA app impersonation and MitM attacks are presented in Figure 7. The attacker advertises her presence as a trusted tracker, and the app sends an `Auth Req` message to the attacker to start Authentication. The attacker sends a parallel `Auth Req` message to the tracker to initiate Authentication with the tracker. The tracker sends a `Chal` to the attacker, who relays it to the app. Then, the app computes `Resp` and sends it to the attacker, who replays it to the tracker to prove ownership of a pairing key that she does not know. The tracker sends an `Auth OK` message to the attacker, and the attacker sends an `Auth FAIL` message to the app if she wants to impersonate it or sends an `Auth OK` message to preserve her MitM position.

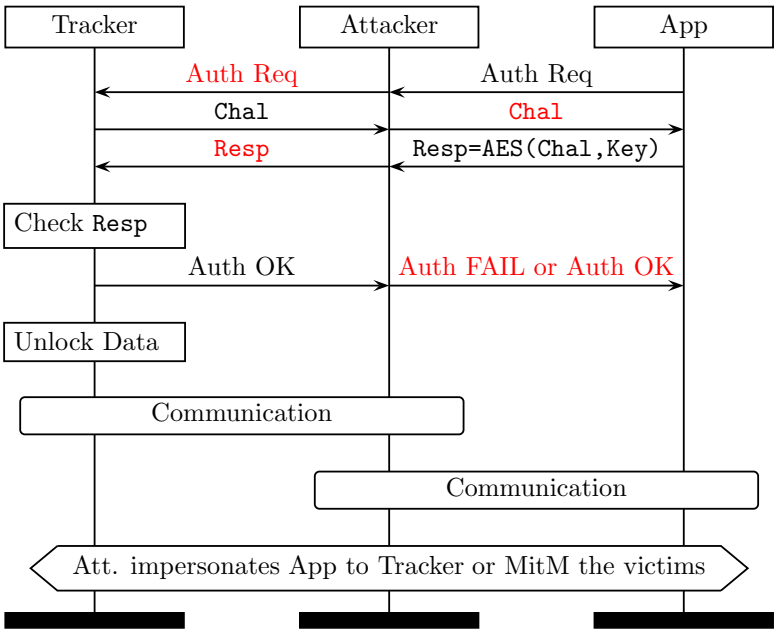

Figure 7: OTA app impersonation and MitM attacks. The attacker impersonates the app and sends an authentication request (`Auth Req`) to the tracker (MB 2/3/4/5/6 and AC 1/2) as the app. The tracker sends an authentication challenge (`Chal`), and the attacker relays it to the legitimate app. Then, the app computes a response (`Resp`) and sends it to the attacker, believing that it is talking to the victim tracker. The response is computed from a pairing key (`Key`) only known to the victims. The attacker relays `Resp` to the tracker, which checks it and sends back a positive authentication message (`Auth OK`). Then, the attacker has two options. She can impersonate the app by sending the app a negative authentication message (`Auth FAIL`) and taking over the communication session. Otherwise, she can relay the positive authentication message (`Auth OK`) to the app and establish a man-in-the-middle position between the victim app and tracker.

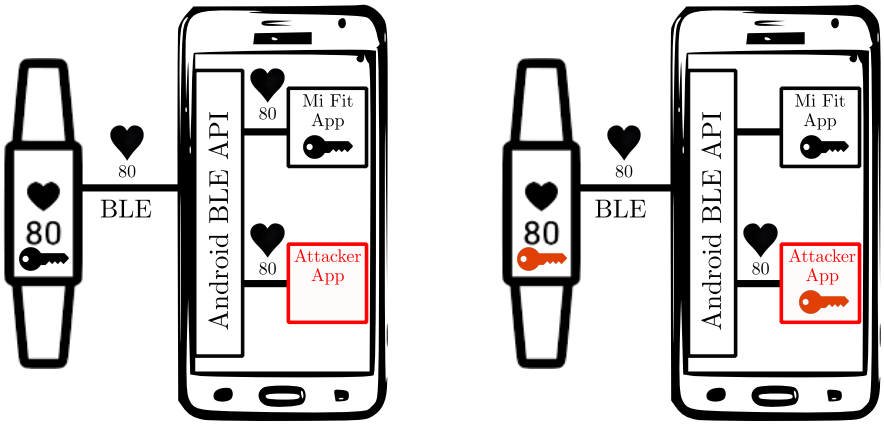

Figure 8: SB eavesdropping (left) and app impersonation (right) attacks. On the left, the attacker eavesdrops on all data exchanged by the tracker and the app by querying the tracker's BLE GATT server without knowing the pairing key (black key). On the right, the attacker impersonates the app by re-pairing with the tracker and establishing a new pairing key (red key) unknown to the impersonated app. SB eavesdropping and app impersonation on Pairing *v1* require only `BLUETOOTH` and `INTERNET` Android permissions. SB app impersonation on Pairing *v2* also requires `BLUETOOTH_ADMIN` and `ACCESS_FINE_LOCATION`.

4.5 SB App Impersonation Attack

The attacker can impersonate any Xiaomi Android app and remotely pair with a victim tracker. The attack works *regardless of* the pairing's protocol version. Similarl to other Android malware threat models [MSBK21], the malicious app is already installed on the victim's smartphone. This app acts stealthily, runs with normal `BLUETOOTH` and `INTERNET` permissions, and does not require root access. The main advantage of this attack over the OTA counterpart is that it can be conducted *remotely* (i.e., over the Internet). Its main drawback is that it requires *user interaction* to trigger a new pairing session. However, since pairing confirmation involves interactions only with the trackers, the user has no way to tell if the tracker is pairing with a malicious or legitimate app.

The SB app impersonation on Android is depicted in the right part of Figure 8 and works as follows. The attacker abuses the Android BLE API to discover a tracker paired with the Xiaomi app. This task can be done by finding Xiaomi proprietary commands in the smartphone GATT traffic or by looking for the BLE address of the tracker in the list of connected devices.

If the target tracker supports Pairing v1, the attacker starts a pairing protocol with the tracker. The tracker cannot tell if the pairing request is authentic and completes pairing with the attacker's malicious app. Otherwise, if the tracker supports Pairing v2 (i.e., server-based), the attacker must use a different strategy. In particular, the attacker sends a *factory reset* proprietary command (that we reverse-engineered). The command does not require prior authentication. It deletes the bond between the tracker and the legitimate app, changes the tracker's BLE address, and puts it in pairing mode. Then, the attacker can find the tracker and complete the pairing. This attack strategy completely defeats server-based pairing, which still lacks strong device authentication. Finally, regardless of the attacked pairing version, the legitimate app *cannot* connect back to the tracker (e.g., in Figure 8, the tracker accepts the red key, but the app has the black one).

4.6 OTA and SB Eavesdropping Attacks

Due to the lack of encryption during Pairing, Authentication, and Communication, the attacker can effortlessly launch OTA and SB eavesdropping attacks.

In the OTA case, the attacker can sniff sensitive data exchanged between tracker and app. For example, during Pairing v1 she can sniff the pairing key, during Pairing v2 the pairing key seed, during Authentication the challenge-response pairs (to be used in an offline brute-force attack), and during Communication all the data sent including health parameters from the tracker and notifications from the app.

In the SB case, the attacker can sniff sensitive data from remote trackers via a malicious app, taking advantage of the lack of encryption and a known issue with the Android BLE API. Android allows applications with `BLUETOOTH` permissions to sniff all GATT data received by a smartphone. Thus, any application co-located with the Mi Fit app can sniff all the traffic coming from a tracker without pairing or authenticating with it. For example, in Figure 8, we depict a malicious app (Attacker App in red) sniffing the heart rate value sent by the tracker by abusing Android BLE API.

4.7 Discussion

Mapping between attacks and vulnerabilities In Table 1, we present a mapping between our attacks and the vulnerabilities presented in Section 3.2. The Pairing v1/v2 protocols can be targeted by OTA eavesdropping, SB eavesdropping, and SB app impersonation. OTA and SB eavesdropping exploit the data sent in the clear during the Pairing protocol (pairing key in Pairing v1 and pairing key seed in Pairing v2). SB app impersonation exploits the missing/weak authentication and weak user confirmation to connect a malicious device to a legitimate tracker. The Authentication protocol can be targeted by OTA tracker impersonation, OTA app impersonation, and OTA man-in-the-middle. OTA tracker impersonation exploits unilateral app authentication (the tracker does not need to authenticate with the app). The OTA app impersonation exploits the replayability of BLE traffic between app and tracker, allowing any device to mimic a legitimate tracker. The OTA man-in-the-middle is a combination of the OTA app and tracker impersonations. The Communication protocol can be targeted by OTA tracker impersonation, OTA app impersonation, OTA man-in-the-middle, OTA eavesdropping, and SB eavesdropping. The lack of encryption and integrity protection in any BLE packet exchanged between app and tracker allows an attacker to eavesdrop and manipulate BLE traffic freely.

Attacks' Impact The attacks' impact is high for several reasons. The proposed attacks, and their root causes (that we RE), were not known by the community. Also, our attack techniques include novel aspects. For example, the SB remote attacks combine unknown Xiaomi vulnerabilities with known Android issues.

The proposed attacks disprove the security and privacy claims made by Xiaomi in their Privacy Policy [Hua20], as we highlight in Section 6. We demonstrate that all Xiaomi trackers released since 2016 are vulnerable to the proposed attacks, and future releases will be vulnerable as well, as our attacks are Xiaomi-compliant. Our attacks affect millions of users, as Xiaomi is the world's leading fitness tracker manufacturer.

The proposed attacks are cheap and low-effort (e.g., only require commercial-off-the-shelf products and minimal equipment) and are easy to deploy. An attacker can violate users' privacy by leaking and manipulating sensitive data (e.g., health records and 2FA SMS) and enforcing malicious factory reset and firmware update requests.

Table 1: Mapping between the exploited vulnerabilities identified in Section 3.2 and the OTA and SB attacks presented in Section 4. A checkmark (✓) indicates that a vulnerability is exploited to conduct an attack.

Vulnerabilities	**OTA Attacks**			**SB Attacks**		
	Tracker Imp.	App Imp.	MitM	Eaves.	Eaves.	App Imp.
Pairing v1						
Pairing key sent in the clear	-	-	-	✓	-	-
Pairing not authenticated	-	-	-	-	-	✓
Weak user confirmation	-	-	-	-	-	✓
Pairing v2						
Pairing key seed sent in the clear	-	-	-	✓	✓	-
Pairing only (weakly) authenticates app	-	-	-	-	-	✓
Weak user confirmation	-	-	-	-	-	✓
Authentication						
Unilateral app authentication	✓	-	✓	-	-	-
Challenges and responses replayable	-	✓	✓	-	-	-
Communication						
No encryption	✓	✓	✓	✓	✓	-
No integrity protection	✓	✓	✓	-	-	-

5 Implementation

In this section, we present the implementation of `breakmi`, a toolkit that we developed to reverse-engineer and attack Xiaomi's proprietary Pairing, Authentication, and Communication protocols. `breakmi` reimplements these protocols and automates our experiments and attacks. The toolkit contains a protocol dissector module, a security mechanisms module, and an attacks module. We will release `breakmi` as open-source, and now we describe how we implemented each module.

5.1 Protocol Dissector Module

Our toolkit, `breakmi`, includes a protocol dissection module capable of speaking Pairing v1/v2, Authentication, and Communication protocols (presented in Section 3). The dissectors module can detect and craft any Xiaomi proprietary message given a capture. We develop the dissectors as an aid for RE. We implement the module defining fifteen custom dissection classes for scapy [BtSC21], an interactive packet manipulation program. Each class encodes a message type using a specific binary layout. Table 2 lists all messages that we can dissect and customize. The table's first column indicates the message type, the second column the message sender, and the third column the packet layout. For example, the Pairing Key message is used by an app during Pairing v1 to send the pairing key and

Table 2: Reversed Xiaomi application-layer opcodes and relevant values. `Key`, `Const`, `R`, `Chal` and `Resp` are 16-byte values shown in hex. `Const` equals to `1863c2cce5d159413bed92c4b163c279`.

Message	Sender	Opcode/Value
Pairing Init	App	0100
`pair_v1`	Tracker	100104
Pairing Key	App	0100, `Key`
Pairing Complete	Tracker	100101
Pairing Fail	Tracker	100204
`pair_v2`	Tracker	10018101
SHA1(`pub_k`)	Tracker	`Const`
Random Req	App	820002
Random Resp	Tracker	108201, `R`
User Confirmation	Tracker	108301
Server Check	Tracker	10008401010000
Auth Req	App	0200 or 820002
Auth Chal	Tracker	100201, `Chal` or 108201, `Chal`
Auth Resp	App	0300, `Resp` or 8300, `Resp`
Auth Complete	Tracker	100301 or 108301
Auth Fail	Tracker	100304 or 108307

the packet contains a leading `0100` and then `Key`.

Pairing v1/v2 messages are managed by Xiaomi's custom Auth GATT characteristic (`00:00:00:09:00:00:35:12:21:18:00:09:AF:10:07:00`), which can be found under Xiaomi's custom GATT service `0xFEE1`. The dissectors extract capture packets with Pyshark [New21], a Python API for Wireshark [Wir21], and label them as v1 or v2. For Pairing v2, they also look for the `Signature` transmitted on Xiaomi's custom Chunked Transfer characteristic (`00:00:00:20:00:00:35:12:21:18:00:09:AF:10:07:00`), under Xiaomi's `0xfee1` service.

The Auth characteristic also serves Authentication messages. The dissectors monitor the status of Authentication and the challenge-response. In our experiments, we crafted Authentication messages with different opcodes and noticed that MB 4/5/6 accept opcodes used by MB 2/3, as shown in Table 2.

The Communication protocol involves several characteristics, but we focused on the standard Heart Rate Measurement (`00:00:2A:37:00:00:10:00:80:00:00:80:5F:9B:34:FB`) and the custom Steps (`00:00:00:07:00:00:35:12:21:18:00:09:AF:10:07:00`). The dissectors decode the custom data format and display the effective value transmitted from the tracker to the app.

5.2 Security Mechanisms Module

Our toolkit also implements the custom key derivation function for Pairing v2 and challenge-response Authentication procedures. By using these functions, `breakmi` is capable of deriving a valid pairing key from its seed (`R`) and a valid authentication response (`Resp`) from a challenge (`Chal`), and a pairing key (`Key`). We invested much time in understanding how the key derivation and challenge-response procedures work, and we reimplemented them with Python. In particular, we used Python's cryptography module from PyCA [Aut21] to implement SHA256 for the key derivation and AES-ECB for the challenge-response part.

To reverse the key derivation and the challenge-response, we used a mix of static and dynamic techniques. For the static analysis, we decompiled the Mi Fit APK with JADX [Sky21] and looked at the recovered source code. Unfortunately, the Mi Fit app is obfuscated, and we could not recover the key derivation and authentication logic. At this point, we switched to dynamic binary instrumentation with Frida [Ple21]. Our dynamic approach was successful, as we were able to reverse the key derivation and authentication logics by hooking their entry point at runtime and observing their inputs and outputs.

5.3 Attacks Module

The OTA tracker impersonation attack, presented in Figure 6, was implemented using Bleno [Mis21a], an open-source BLE peripheral written in Node.js. Our Bleno script imitates any Xiaomi tracker by exposing the same BLE features (e.g., BLE advertisements and GATT server) collected by an extensive study of legitimate Mi Band 2/3/4/5/6, as shown in Table 6 and Table 7 in the Appendix. It was challenging to collect this information from all trackers and make sense of the (proprietary) characteristics and services exposed by the trackers. Once a victim app finds our Bleno tracker, we can complete Pairing v1 and Authentication as a trusted device, and we expose a malicious GATT server to the app during Communication.

The OTA app impersonation attack, presented in Figure 7, was implemented using Noble [Mis21b], an open-source BLE central, and by re-using the tracker impersonation described above. Our tool connects to nearby Xiaomi apps and trackers, performs a replay attack on the Authentication protocol, disconnects from the app, and establishes a communication session with the tracker as a trusted app. Once connected, our fake app retrieves data from the tracker, such as the step count and the user's heart rate. Furthermore, it can send fake SMS and phone call notifications and activate alarms. The OTA MitM attack implementation uses the same logic of the app impersonation. However, instead of disconnecting from the app after the replay attack is completed, the tool keeps two parallel connections and establishes a MitM position between the app and the tracker.

The toolkit also includes a malicious app that can be used to perform SB app impersonation and eavesdropping attacks. Our app requires only Android's `BLUETOOTH` permission to interact with the tracker over BLE and `INTERNET` permission to exfiltrate data remotely. Android classifies these permissions as normal, so they are granted during installation without triggering any user prompt.

SB eavesdropping and SB app impersonation utilize the same setup that periodically checks active BLE connections using Android `getConnectedDevices` API and waits for a Xiaomi tracker to appear. As soon as a tracker connects, the malicious app launches the attack.

During SB eavesdropping, our app subscribes to relevant characteristics and can eavesdrop on all BLE data coming from the tracker, including sensor readings, commands, pairing key seeds, and authentication challenges. During app impersonation, our app starts a new pairing session with the tracker by sending a `Pairing Init` without disrupting the communication between the tracker and the legitimate app. The malicious app eventually negotiates a new pairing key as in the right part of Figure 8 and gains access to protected data. The SB app impersonation attack on Pairing *v2* entails the additional challenge of interacting with the Xiaomi backend to retrieve a signature. The malicious app sends a factory reset command to the tracker, which causes the change of its BLE address. A scan (requiring Android `BLUETOOTH_ADMIN` and `ACCESS_FINE_LOCATION` permissions) allows our app to perform a new pairing process and associate the tracker to our malicious Xiaomi account stealing its ownership from the legitimate user.

Since all attacks performed by `breakmi` are fully automated, we propose it as a continuous evaluation tool for the Xiaomi ecosystem. Even if Xiaomi were to enable link-layer security, the vulnerabilities we found at the application-layer would continue to exist. By

Table 3: Fitness trackers' technical specifications. The columns contain the device name, release year, supported Bluetooth version (BTv), Pairing protocol version (Pv), BLE link-layer security support (BS), BLE Secure Connections (SC) support, used system-on-chip (SoC), and Mi Fit firmware version (FW).

Device	Year	BTv	Pv	BS	SC	SoC	FW
Mi Band 2	2016	4.2	1	✓	✗	DA14681	1.0.1.81
Mi Band 3	2018	4.2	1	✓	✗	DA14681	2.4.0.32
Cor 2	2019	4.2	1	✓	✗	DA14681	0.3.0.44
Mi Band 4	2019	5.0	2	✓	✓	DA14697	1.0.9.66
Mi Band 5	2020	5.0	2	✓	✓	DA14697	1.0.2.64
Mi Band 6	2021	5.0	2	✓	✓	DA14699	1.0.1.36

using `breakmi`, anyone would be able to test for those vulnerabilities in future Xiaomi fitness trackers.

6 Evaluation

This section describes the evaluation of the attacks (presented in Section 4) using `breakmi` (described in Section 5). Our evaluation confirms that all Xiaomi trackers since 2016 and the most recent version of Xiaomi companion apps are *vulnerable* to our protocol-level attacks. In addition, it proves that `breakmi` works in practice and is cheap to deploy. As a result, millions of Xiaomi users are potential targets, and their sensitive health and personal data can be leaked and manipulated by bad actors.

6.1 Setup

In our evaluation, we tested all trackers shipped by Xiaomi since 2016. The sample includes Mi Band 2/3/4/5/6 and Amazfit Cor 2. The MB 1 is out of scope because it is known to ship with no security at all [of19]. We did not test the Amazfit Cor 1 because of its limited availability and market share. However, we expect that it is vulnerable to our attacks as it is a MB 2 clone. Moreover, we tested the latest versions of Mi Fit (v 4.8.1) and Zepp (v 5.9.2) as the official Xiaomi mobile applications. The apps are compatible with all Xiaomi trackers and are available for Android and iOS. Despite our attacks *not* requiring root permissions, we use rooted devices to facilitate our experiments.

Table 3 presents the trackers' technical specifications. The Mi Band 2, Mi Band 3, and Cor 2 support Bluetooth version (BTv) 4.2 and Pairing version (Pv) v1. The others support Bluetooth 5.0, Pairing v2, and BLE Secure Connections (SC). All trackers support BLE security (BS) at the link-layer, but Xiaomi is not taking advantage of that. The tracker system-on-chip (SoC) is either a DA14681, a DA14697, or a DA14699, all manufactured by Dialog Semiconductor [Sem21]. We also report the firmware version (FW) of the tracker at the evaluation time.

Since our SB attacks depend on an issue with the Android BLE API (in addition to Xiaomi ones), we tested six popular Android versions using different smartphones: Android 12 (Google Pixel 4A), Android 11 (Google Pixel 2 XL), Android 10 (Google Pixel XL), Android 9 (Samsung Galaxy J5), Android 8.1 (Xiaomi Redmi 5 Plus) and Android 6 (Samsung Galaxy S5). We note that we cannot test our attacks on the Android emulator as it does not support Bluetooth emulation.

Our OTA attacking device is an Acer Aspire 3 laptop connected with a BLE sniffer and a BLE dongle. The laptop runs Ubuntu version 18.04 and supports Bluetooth 4.2.

Table 4: Evaluation results for OTA and SB attacks. The first column shows the attack name, the following eight columns contain the targets (two companion apps and six fitness trackers). A checkmark (✓) means that the attack was successful, and a hyphen (-) means that the attack does not apply to that target. MB and AC abbreviate Mi Band and Amazfit Cor. The SB attacks on the Mi Fit and Zepp apps were successfully tested on six Android versions (see Table 5).

	Mi Fit	Zepp	MB2	MB3	AC2	MB4	MB5	MB6
OTA Tracker Impersonation	-	-	✓	✓	✓	✓	✓	✓
OTA App Impersonation	✓	✓	-	-	-	-	-	-
OTA Man-in-the-Middle	✓	✓	✓	✓	✓	✓	✓	✓
OTA Eavesdropping	✓	✓	✓	✓	✓	✓	✓	✓
SB Eavesdropping	-	-	✓	✓	✓	✓	✓	✓
SB App Impersonation	✓	✓	-	-	-	-	-	-

The sniffer uses three BBC Micro Bit boards and btlejack [Vir21]. The BLE dongle is a CSR8510 A-10 Controller with Bluetooth 4.0 support. The dongle is needed as we have to change the attacking device's BLE address, and the laptop (BLE controller) does not allow this operation.

The OTA impersonation and MitM attacks were performed by the attacking device running `breakmi` and acting as both a spoofed tracker and a spoofed app. When impersonating the tracker, the attacking device advertises as a spoofed tracker, while during an app impersonation, it scans for trackers as a spoofed app. OTA eavesdropping was performed using the BLE sniffer. The SB app impersonation and eavesdropping attacks were performed by running the malicious app in the background and letting the legitimate app and the tracker communicate as usual.

6.2 Results

The attacks' evaluation results are shown in Table 4, where we demonstrate that the attacks are effective across *all* evaluated devices (if the attack applies to that device). We can wirelessly impersonate all tested trackers and apps, MitM them, and eavesdrop on their communication. We can also remotely eavesdrop and impersonate the Mi Fit and Zepp apps for Android via a malicious app. All attacks work *regardless of* the hardware and software details of the victim device (e.g., SoC, firmware, app, tracker, and BLE versions).

The SB remote attacks target Android, so we test them on six popular Android versions, as we show in Table 5. We confirm that *all* six tested Android versions are vulnerable to our attacks. According to [Sta21], this means that (at least) 90.22% of Android devices are vulnerable. Since Android 12 was recently released in October 2021, statistical market share data is not available yet. We highlight that, up until Android 11, our attacks only ask for standard Bluetooth permissions as a requirement to access the `getConnectedDevices` method (that we exploit). Android 12 introduces the `BLUETOOTH_CONNECT` dangerous-level runtime permission, which must be declared to access `getConnectedDevices`. As a consequence, on Android 12, our malicious app must show the user a Nearby Devices dialog that explicitly states our intent to find, connect to and determine the location of nearby devices.

Table 5: Evaluation results for SB attacks against the latest six Android versions. All attacks were tested using a MB 4 as a tracker. The first column shows the smartphone model, the second one the Android version and its market share according to [Sta21]. Market share numbers for Android 12 are not available (n/a) as it is too recent. The third and fourth columns show that all Android versions we test are vulnerable to SB attacks (✓). If we sum the markets share numbers, our attacks are effective on at least 90.22% of Android devices.

Smartphone	Android	SB Eaves.	SB App Imp.
Pixel 4A	12 (n/a)	✓*	✓*
Pixel 2XL	11 (34.7%)	✓	✓
Pixel XL	10 (27.7%)	✓	✓
Galaxy J5	9 (13.9%)	✓	✓
Redmi 5 Plus	8 (10.56%)	✓	✓
Galaxy S5	6 (3.36%)	✓	✓

* Attack requires Android `BLUETOOTH_CONNECT` dangerous permission

7 Countermeasures

We now discuss five countermeasures addressing the vulnerabilities presented in Section 3.2 and the attacks presented in Section 4. The first four apply to the application-layer and the fifth to the link-layer.

C1 (Authenticated) Key Establishment Pairing v1/v2 should use an Authenticated Key Establishment (AKE) to prevent impersonation, man-in-the-middle, and eavesdropping attacks. Figure 9 illustrates an updated version of Pairing v1/v2 to support C1. The tracker and the app first generate a public-private key pair and share their public keys (`App_Pk` and `TR_Pk`). They proceed with the calculation of a key (`K`) through a Diffie-Hellman (DH) function and the sharing of two nonces (`App_N` and `TR_N`). Finally, the tracker and the app calculate a confirmation value (`V`) displayed on each screen and wait for user confirmation concerning the match of the displayed values.

C2: Strong Pairing Confirmation The updated Pairing v1/v2 protocol, shown in Figure 9, guarantees C2 thanks to the numeric comparison performed by the user. In particular, while pairing, a MitM attacker is not capable of generating `V` as she does not know `K` pairing. Moreover, during an app impersonation attack, the adversary would trigger an unexpected user interaction while re-pairing with the app.

C3: Strong Key Authentication The Authentication procedure should be mutual and resistant to replay attacks. Mutual authentication is easy to implement by letting the app and the tracker send their challenges and then verifying them on both ends. Replay protection is also straightforward and can be achieved by using nonces and generating a response from a challenge and a nonce. These measures raise the bar for tracker impersonation and app impersonation. Figure 10 illustrates the updated Xiaomi Authentication protocol. Tracker and app already share the pairing key (`K`) and exchange a challenge (`App_Ch` and `TR_Ch`). They calculate the solutions through a hash function `H`, which relies on the challenges and the pairing key. They verify the correctness of the received challenge solution, and if both checks are successful, the tracker unlocks its data. Finally, they start an AES-CCM encrypted communication session using a session key `SK` obtained through a

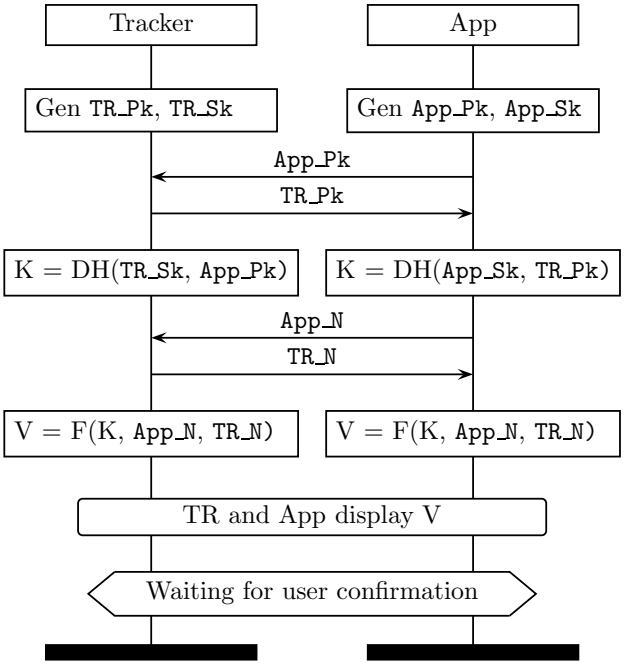

Figure 9: Authenticated key establishment (i.e., pairing) protocol providing C1+C2. Tracker and app generate a public-private key pair and share the public keys (`App_Pk` and `TR_Pk`) with each other. They calculate a new key through a Diffie–Hellman (DH) function and share two nonces (`App_N` and `TR_N`). Then, the tracker and the app both calculate a confirmation value (`V`) displayed on each screen and wait for user confirmation concerning the match of the displayed values. The DH function guarantees C1, while the user verifying tracker and app during pairing confirmation guarantees C2.

HKDF key derivation function from the pairing key `K`, and two exchanged nonces (`App_N` and `TR_N`).

C4: Authenticated-encryption The Communication protocol should use a fresh session key derived from the pairing key `K` to encrypt and integrity-protect the data exchanged between app and tracker. Devices can rely on AES-CCM and HKDF to introduce this countermeasure, as both functions are already supported by the devices SoC. C4 protects against eavesdropping and MitM attacks during Communication.

C5: BLE Link-Layer Security To complement the security at the application-layer, Xiaomi might also enable the BLE link-layer security mechanisms already supported by all its devices. The robustness of BLE link-layer security mechanisms depends on the BLE version of the device. The MB 4/5/6 implement BLE SC, so that they would benefit from secure protocols for pairing and session establishment. Instead, the MB 2/3 implement legacy BLE security, which is known to be insecure [Rya13a].

8 Comparison with Fitbit

In this section, we compare our findings of Xiaomi with the *Fitbit* [Fitb] ecosystem, which is Xiaomi's main competitor in the fitness tracker market. Our motivation for the comparison is twofold. Firstly, to assess if Fitbit is affected by similar vulnerabilities and attacks found on Xiaomi devices. Secondly, to evaluate how effective `breakmi` is on other large

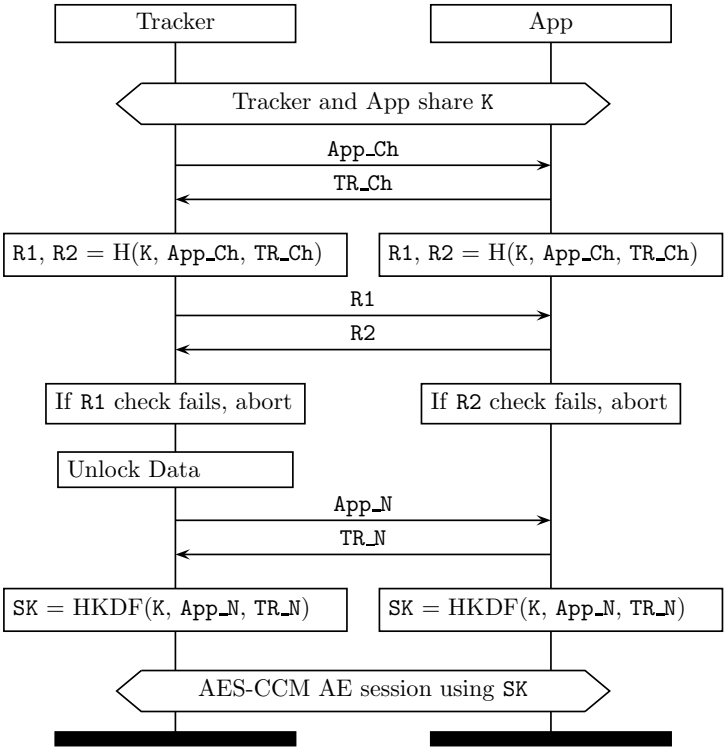

Figure 10: Mutual authentication protocol providing C3+C4. Tracker and app already share the pairing key (`K`) and exchange a challenge (`App_Ch` and `TR_Ch`). They calculate the solutions through a hash function `H`, that relies on the challenges and on the pairing key. They verify the correctness of the received challenge solution, and if both checks are successful, the tracker unlocks its data. Finally, they start an AES-CCM encrypted communication session using a session key `SK` obtained through a HKDF key derivation function from the pairing key `K`, and two exchanged nonces (`App_N` and `TR_N`). The mutual verification of challenges guarantees C3 and AES-CCM with a fresh session key provides C4.

fitness tracking ecosystems. We used the same RE methodology adopted for Xiaomi and described in Section 9 for this analysis, but we targeted different proprietary protocols.

In particular, we had a look at a Charge 2 tracker and the latest version of the Fitbit app for Android [Fita], as their security mechanisms were already reversed [CHMS14, SJMdR16, FCS+17, CWP+18]. The Charge 2 is from 2016, supports Bluetooth 4.1 and BLE link-layer security, and is powered by a BLUENRGCSP SoC from ST Microelectronics. We also considered the Charge 4 from 2020, but unlike the Charge 2, it uses unknown protocols, and reversing them is out of the scope of this work.

We now summarize what is known about the Fitbit protocols. Then we discuss their vulnerabilities and how the attacks presented in Section 4 apply to them. We describe how we extended `breakmi` to deploy five attacks on actual Fitbit devices successfully. Finally, we discuss how to port the countermeasures discussed for Xiaomi to the Fitbit ecosystem.

8.1 Architecture and Protocols

Fitbit uses the same system architecture as Xiaomi (see Figure 1). A tracker communicates over BLE with the Fitbit companion app, and the app communicates via Wi-Fi or a mobile network with the Fitbit backend. The devices use proprietary application-layer protocols

(e.g., Pairing, Authentication, and Communication) and ignore available BLE link-layer security mechanisms. Unlike Xiaomi, which utilizes public BLE addresses, Fitbit trackers use *random static* addresses. Regardless, Fitbit devices are still trackable because their BLE address never changes for their entire lifetime.

Pairing As described in [SJMdR16, CWP+18], Fitbit employs a Pairing protocol to establish a pairing key (authentication key in Fitbit terms) between the trackers and the app. During Pairing, the backend (which pre-shares a device key with a tracker) computes the pairing key from the device key and a salt, and sends them to the app. The app sends the salt to the tracker, and the tracker computes the pairing key from the salt and the device key. We note that, unlike Xiaomi, the key or its seed is *not* sent in cleartext, and pairing confirmation is strong. Pairing is accepted only when the user confirms on the app that she sees the same numeric sequence on the tracker and app screens. As such, Fitbit adopts a stronger user confirmation strategy than Xiaomi.

Authentication Fitbit Authentication, unlike Xiaomi, is *mutual* and works as follows. The app sends a random challenge together with a constant key salt (provided by the backend during Pairing). The tracker sends back a MAC and a counter value, where the MAC is computed using the counter and the random challenge and is keyed using the pairing key corresponding to the key salt. The app checks that the MAC is valid and sends back a different MAC computed using the counter and keyed with the pairing key. The tracker checks the MAC, and then mutual authentication of the pairing key is achieved. The counter is updated on each run of the Authentication protocol.

Communication Fitbit sessions, unlike Xiaomi, have two different modes: live and normal. Live mode is *not encrypted* and monitors the tracker readings in real-time. Live mode data stays in the app, and is not relayed to the backend. On the other hand, normal mode synchronizes the data from the tracker to the backend. The synchronization process is *encrypted* with the shared pairing key by using either XTEA (extended TEA) encryption [NW97] or AES-EAX authenticated-encryption [BRW04].

8.2 Vulnerabilities and Attacks

We compare Xiaomi and Fitbit security mechanisms, investigate if Xiaomi vulnerabilities can be found in the Fitbit ecosystem, and evaluate how our OTA and SB attacks perform on Fitbit.

Vulnerabilities Fitbit proprietary security mechanisms are slightly better than Xiaomi's, but severe vulnerabilities still affect them. In particular, the Fitbit Pairing protocol does not send the pairing key (seed) in the clear, has strong user confirmation but still *lacks* device authentication. The Authentication protocol is mutual but is *replayable*. The Communication protocol *partially* uses encryption and integrity protection. For example, only normal mode data is encrypted.

OTA attacks The OTA app impersonation and MitM attacks from Figure 7 are still effective, as Fitbit Authentication is not replay-protected. Instead, the OTA tracker impersonation presented in Figure 6 does not work as Fitbit Authentication is mutual. However, the attacker can still impersonate a tracker using a replay attack similar to the one described in Figure 7. The OTA eavesdropping works only with unencrypted data (e.g., live mode data).

SB attacks The SB app impersonation attack on Android in Figure 8 is still effective. A malicious app can get valid authentication credentials from the backend and re-pair with the victim tracker (see Fig 4a in [CWP+18]). Thus, we discovered a novel technique to steal trackers virtually. The SB eavesdropping suffers the same limitations as OTA eavesdropping.

Overall, the attacks' impact on Fitbit is lower than the one on Xiaomi but remains significant. For example, during impersonation or MitM, the attacker can only manipulate

and tamper with the packets sent in cleartext. Nevertheless, the attacker can still abuse the unprotected live mode data to report worrisome health conditions to the user.

8.3 Attacking Fitbit with `breakmi`

We extended `breakmi` to analyze and attack the Fitbit ecosystem. We can clone Charge 2 trackers (including their advertised data and GATT servers) and the Fitbit app. Moreover, we can speak the Pairing, Authentication, and Communication protocols described before. We created custom scapy dissection classes to generate valid packets just like we did for Xiaomi. Fitbit uses *static* (random) BLE addresses, and we updated `breakmi` to support this privacy feature.

We use `breakmi` to perform the OTA impersonation and MitM attacks by replaying packets during the Fitbit Authentication. We also developed an extra module for our malicious Android app that performs the SB eavesdropping and app impersonation attacks. The latter provides the trackers' serial number and BLE address to the backend to reset the tracker's owner and then triggers pairing from the malicious app. This strategy is different from those we presented in Section 4.5, targeting Xiaomi.

8.4 Porting our Xiaomi Countermeasures to Fitbit

Fitbit Pairing, Authentication, and Communication protocols described in Section 8.1 can be strengthened by using a subset of the countermeasures proposed for Xiaomi in Section 7. In particular, unencrypted live mode data can be protected with C1 (authenticated-encryption), Pairing can be enhanced with C2 (authenticated key establishment), Authentication can be improved by adding replay protection as in C3. In addition, defense in depth can be achieved using C4 (BLE link-layer security). C5 (strong pairing confirmation) is not needed as is already provided by Fitbit pairing confirmation.

9 Reverse-Engineering Methodology

In this section, we describe our reverse-engineering methodology and how we applied it to perform our security analysis of Xiaomi and Fitbit ecosystems. We describe how we perform reconnaissance on a fitness tracking ecosystem. We explain how we analyze the traffic exchanged between tracker, app, and backend and how we apply static and dynamic analysis techniques to a mobile app. We also discuss the development of automated scripts for reverse-engineering and security assessment.

9.1 Trackers and Apps Reconnaissance

We now describe how we performed reconnaissance of the Xiaomi trackers and apps.

Regarding the tracker, we inspect its BLE GATT server using the "nRF Connect for Mobile" app [ASA21]. The app allows to scan, explore and communicate with BLE devices. We extract data from every service and characteristic, collecting the information shown in Table 6 and Table 7 in the Appendix. We identify Xiaomi proprietary GATT services (i.e., `0xFEE0`, `0xFEE1`). We find a set of characteristics protected by Authentication. For example, the Auth characteristic manages Pairing and Authentication, and the Steps and Heart Rate characteristics contain sensitive data.

We install Mi Fit and Zepp apps and interact with them. The apps are very similar as they share the same UI, communicate with the same backend, and provide the same interface to the tracker. The apps' UI does not show any information about the Pairing, Authentication, and Communication protocols that we RE. We also find that their codebase is very similar despite being closed-source. GadgetBridge [Fre21], an open-source project, provides some insights into the apps' internals.

9.2 BLE and Web Traffic Analysis

First, we intercept over-the-air BLE traffic with a BLE sniffer and confirm that Xiaomi BLE traffic is not encrypted. Then, since we control the smartphone running the app, we simply enable the "Bluetooth HCI Snoop Log" under the "Developer Options" and directly access BLE capture files. This option does not work correctly on some smartphones, but other alternatives exist (e.g., hcidump [KHG21], adb bugreport [And21]). We visualize and inspect BLE packets using Wireshark [Wir21], a network protocol analyzer. We find several recurring opcodes, shown in Table 2. We define the Pairing, Authentication, and Communication protocols and implement them in our automated scripts.

We run mitmproxy [Pro22] on our machine, a MitM proxy tool that intercepts and logs Wi-Fi and cellular networks traffic. We also configure our smartphone to redirect its web traffic to mitmproxy, and we install the mitmproxy CA (Certificate Authority) certificate. We intercept traffic going from the app to the Xiaomi backend while performing various operations with the tracker (e.g., adding a new tracker, synchronizing user activity data, completing workout sessions). We discover several API endpoints (e.g., `account.xiaomi.com/oauth2/authorize`, `account.huami.com/v2/client/login`, `api-mifit-de2.huami.com/v1/device/binds.json`). We inspect the traffic looking for interesting requests. For example, we reverse-engineer how Xiaomi registers new trackers on its backend. Then, we test the API endpoints by sending custom-made requests and monitoring their responses (the tests were performed according to Xiaomi's Bug Bounty program guidelines).

We merge BLE and Web capture files using a Wireshark utility better to visualize the sequentiality of the packets in Xiaomi protocols. We discover how the app acts as a proxy between the tracker and the backend during Pairing v2 and how the app sends security packets to the unprotected (custom) Chunked Transfer characteristic.

9.3 Mobile Companion Apps Analysis

We examine Mi Fit and Zepp apps' code to uncover the implementation of Xiaomi proprietary protocols from the source. We describe which static and dynamic analysis techniques we applied to our app analysis.

We start our static code analysis by extracting Mi Fit and Zepp APKs and decompiling them with JADX [Sky21], a Dex to Java decompiler. We discover that the Java code is obfuscated and difficult to navigate. We experiment with different deobfuscation tools, either by directly acting on the APK files (i.e., JADX deobfuscation utility, DeGuard [BRTV16], simplify [Fen20]) or by converting them into JAR files first (i.e., Java Deobfuscator [Sam20]), but they were unable to deobfuscate it. We utilize apktool [iBo21] to reverse-engineer the APK, inspect its resources and experiment with repackaging.

We manually inspect the apps' code. We search for keywords, trying several interesting words (e.g., Pairing, Bonding, Authentication, and Characteristic strings) and cryptographic functions supported by the tracker (e.g., MD5 and AES-ECB). We utilize the Mobile Security Framework (MobSF) [Abr21], an automated pen-testing and malware analysis tool, to inspect app components and for its automated binary analysis. We create control-flow graphs with Androguard [DG19], a Python reverse-engineering tool. Then, we perform dataflow analysis to help us to track the pairing `Key`. Ultimately, we find the code sections responsible for Pairing, Authentication, and Communication.

We apply dynamic analysis techniques to confirm that those code sections are actually executed at runtime. We rely on Frida [Ple21], a dynamic binary instrumentation toolkit that allows us to inject code during runtime execution. We use Frida hooks to log runtime variables and confirm they match with the values found in the capture files. We also experiment with editing variables at runtime and test the robustness of Xiaomi's security mechanisms when receiving unexpected values.

9.4 Development of Scripts

Throughout our RE efforts, we develop a set of scripts that automate time-consuming tasks. We aggregate and upgrade those scripts in `breakmi`, an extensible modular toolkit.

First, we build automated scripts to interact with a tracker's GATT server using a Python library called Bleak [BL21]. Our scripts automatically connect, explore, and display information about any BLE device. Then, we build scripts that replicate interesting operations on the tracker (e.g., read requests, enable notifications, firmware update, and factory reset).

We automate BLE traffic analysis by developing protocol dissectors on Pyshark [New21], a Python wrapper for packet parsing, and by using the scapy [BtSC21] packet crafting library. We reverse-engineer the binary structure of Xiaomi firmware so that we can extract firmware from capture files containing a firmware update. We also automate web traffic analysis via mitmproxy by developing scripts with the mitmdump utility. Our mitmdump scripts analyze hundreds of web requests to find pairing-related messages, identify their purpose and retrieve their parameters.

We automate our OTA attacks using Bleno [Mis21a] and Noble [Mis21b]. We create spoofed BLE peripherals that mirror services, characteristics, and advertising from legitimate MB 2/3/4/5/6 and Cor 2. We implement the Pairing, Authentication, and Communication protocols on the Auth, Steps, Heart Rate, and Chunked Transfer characteristics. We also implement the Pairing, Authentication, and Communication protocols on a BLE central and configure it to scan for and connect to the target trackers. During OTA MitM, the malicious BLE central and peripheral communicate with each other through websockets.

10 Related Work

We discuss the current state of the literature concerning the security and cryptographic analysis of Mi Band, Fitbit, and other embedded devices, the attacks on BLE protocols, and Android vulnerabilities and compare it with our work.

Attacks against Mi Band devices Fereidooni et al. [FFM+17] looked at ways to inject false data from the tracking application to the backend of 17 popular trackers, including a Mi Band device. Hilts et al. [HPK16] presented a security and privacy analysis of six trackers, including a Mi Band. These analyses are useful yet orthogonal to ours as they do *not* cover Xiaomi's proprietary security protocols. Some developers released tools for Mi Band 2 [Cre19] and Mi Band 3 [Yog19] able to trigger Xiaomi's Pairing v1 to unlock private data as described in [Ojh18]. Mi Band 4 tools such as [Sat21] require the knowledge of the pairing key, because nothing is known about Pairing v2 apart from it being "server-based" [Fre21]. Our attacks are much stronger and stealthy. They do not require knowledge of the pairing key, no specific action from the victim, do not disrupt the pairing between the victim's app and the victim's tracker, and do not reset the data (as they completely skip Pairing). In fact, we improved and corrected the attack in [Ojh18], that claims to be targeting Authentication when it is actually targeting Pairing v1. The Xiaomi protocols reverse-engineered in [Gie18] belong to Mi Home [Incg, BXC], the Xiaomi app that manages smart home devices, which does *not* support fitness trackers. To conclude, existing attacks were ad-hoc and partial. Our work is the first to systematize and generalize attacks against the Xiaomi fitness tracking ecosystem.

Attacks against Fitbit devices Rahman et al. [RCB13, RCT16] attacked the legacy ANT Communication protocol used by Fitbit Ultra and Garmin Forerunner and provided fixes for them. Cyr et al. [CHMS14] utilized Ubertooth to sniff OTA BLE traffic from a Fitbit

Flex and an HTTP/HTTPS proxy, underlining several privacy-related issues, including device identification attacks. As described in [AV-15], Fitbit Charge was patched for being vulnerable to non-authenticated reads and a replay attack. Goyal et al. [GDS16] presented a comparative analysis of a Jawbone UP Move and Fitbit Charge. They discovered security issues in the GATT server, the mobile app, and the backend. Schellevis et al. [SJMdR16], reversed the Fitbit Charge HR authentication protocol through firmware analysis. Fereidooni et al. [FCS+17] found ways to spoof Fitbit Flex and One data by tearing down the devices and analyzing their firmware. Classen et al. [CWP+18] demonstrated attacks on Fitbit capable of leaking private data from a tracker, re-flashing a rogue firmware, and redirecting the app to a rogue cloud service. Other researchers could only perform eavesdropping while being near the target device, but we can do it remotely with our SB eavesdropping attack. Our SB app impersonation attack allows us to remotely inject fake data into the Fitbit account of our victim. Instead, previous studies required the ownership of the tracker or physical access. We also verified that Fitbit Charge 2 is still using the same protocols reversed in [CWP+18], and we have proven the feasibility of a MitM setup by deploying our OTA MitM attack.

Attacks against BLE protocol Fitness trackers communicate with the app using BLE. Privacy-oriented case studies show, by looking at BLE advertising [DPCM16, IT17, KS18, FKS16] or BLE UUID [ZWLZ19], that one can fingerprint the tracker and even identify the user's activities (e.g., walking or sitting). Several papers discussed standard-compliant attacks on BLE legacy pairing [Rya13b], key negotiation [ATR20], Secure Simple Pairing [BN20, ZWD+20a, ZWD+20a], Secure Connections Only Mode [ZWD+20b], associations [vTPFG21], GATT [Jas16], and reconnections [WNK+20]. Implementation-specific bugs and related exploits were also discussed [Inc19, GCW20]. Wang et al. [WHZ+20] exploited the BLE features designed for low-cost devices to downgrade the key negotiation and authentication procedures and access the stored BLE data. BLE legacy pairing and Secure Simple Pairing were also found to be vulnerable to misbinding attacks [SPA19]. Works about BLE are orthogonal to what we present as we are targeting proprietary protocols implemented at the application-layer on top of an insecure BLE link layer.

Cryptographic security on embedded systems Constrained embedded devices often misuse cryptographic primitives, thus introducing severe vulnerabilities in the whole system. Wouters et al. [WMA+19, WGP21] reverse-engineered the Tesla Model S and Model X key fob and found new vulnerabilities. Our study and methodology are similar to theirs, as they reversed proprietary protocols (i.e., Tesla's PKES and Pairing protocols), analyzed the BLE SoC, discussed cryptographic security measures, developed a proof-of-concept and proposed countermeasures. Their findings could apply to other key fob manufacturers, similar to what we did with `breakmi`. We exploited the same vulnerabilities found both in Xiaomi and Fitbit devices, two of the largest players in the fitness tracker market. An orthogonal approach to reverse-engineering embedded systems is firmware and binary code analysis, adopted by `Incision` [TdHV+21], an architecture and OS-agnostic RE framework. While similar tools are effective, we believe that our application-layer analysis is necessary to provide a full security assessment of an embedded device.

Vulnerabilities in Android It is known that Android does not properly isolate Bluetooth keys among apps. For example, all devices' Bluetooth pairing keys are shared by all apps enabling co-located attacks [NyZD+14, SB19]. This fact might affect fitness trackers that are relying on BLE pairing. However, this is *not* the case for Xiaomi, which only relies on proprietary security mechanisms at the application-layer. As a result, our attacks are still effective even if Android would fix the shared pairing key issue.

Smartwatches Prior work also studied the security of smartwatches, with Apple Watch as the main target. In particular, researchers focused on MagicPairing [HCR20], Apple's Bluetooth security mechanism, and reverse-engineering [HPK16, FFM+17]. In this work, we focus only on fitness trackers, as smartwatches represent a separate class of devices with far more capabilities than fitness trackers. For example, smartwatches ship with more capable SoC (e.g., ARM general-purpose and multi-core application processors) than trackers (e.g., ARM microcontroller), so comparing the two devices classes would be unfair.

11 Conclusion

Xiaomi is a market leader in the fitness tracking industry. Little is known about this ecosystem's security and privacy properties despite managing the sensitive information of millions of users (such as health and personal data). Nonetheless, Xiaomi claims to be "committed to protecting the privacy, confidentiality, and security of personal information" in its Privacy Policy [Hua20]. We address this relevant issue by performing an extensive and up-to-date security evaluation of the Xiaomi fitness tracking ecosystem.

After extensive RE experiments, we uncover several worrisome issues. Xiaomi uses proprietary and undocumented security mechanisms to protect the communication between its trackers and apps. In particular, Xiaomi provides Pairing, Authentication, and Communication application-layer protocols over an insecure BLE connection. Xiaomi's approach is extremely risky as the security of its ecosystem relies on custom protocols that cannot be peer-reviewed in the open by the security community. Moreover, Xiaomi ignores standard BLE link-layer security mechanisms despite being supported by its devices.

We uncovered thirteen severe vulnerabilities (most of which were unknown) in the specification of Xiaomi custom application-layer protocols and exploited ten of them. The issues range from unilateral and replayable authentication to the lack of encryption and integrity protection. Being Xiaomi-compliant, the issues are exploitable on all devices using these protocols.

We demonstrate how to exploit the vulnerabilities with proximity-based (OTA) and remote (SB) attacks. Specifically, we describe over-the-air eavesdropping, impersonation, and MitM attacks, and remote eavesdropping and impersonation threats based on a malicious app co-located with the Xiaomi app.

In our evaluation, we successfully attacked all Xiaomi trackers released since 2016 (e.g., MB 2/3/4/5/6, Cor 2) and the up-to-date versions of the Mi Fit (v 4.8.1) and Zepp (v 5.9.2) companion apps. We also positively test our remote attacks on six popular Android versions (i.e., Android 6/8/9/10/11/12), covering at least 90.22% of Android devices in the market, according to [Sta21].

We develop `breakmi`, a modular toolkit that automates RE experiments and attacks. `breakmi` includes protocol dissector, security mechanisms, and attacks modules. It is based on open-source software and requires cheap and available hardware. We test our toolkit on the Xiaomi ecosystem, and we extend it to also work on Fitbit. We will open-source `breakmi`.

We propose *five* effective countermeasures that fix the presented vulnerabilities and attacks. Our countermeasures provide stronger Pairing, Authentication, and Communication protocols. Moreover, we show how to integrate our protocols into the Xiaomi ecosystem.

We assess whether the Fitbit ecosystem suffers from the same vulnerabilities as Xiaomi. We find that Fitbit has better (still proprietary) Pairing, Authentication, and Communication protocols. Nevertheless, it is vulnerable to five out of the six attacks presented in this paper. We conduct such attacks on actual Fitbit devices, and we extend our toolkit to be compatible with Fitbit.

Overall, our work required an extensive and time-consuming RE effort. The hardest RE challenge was the server-based Pairing v2. More specifically, it took us six months to

understand how the tracker, app, and backend interact among themselves while pairing. We spent two months developing `breakmi`, which automated most of our protocol analysis and reverse-engineering and quickly made up for the time investment. For example, when Xiaomi released the new MB 6, our toolkit immediately detected its similarity to the previous generation of trackers. Using this information, we could deploy our attacks on the MB 6 within a single day. Another example would be the process of adding support for the Fitbit ecosystem. We only spent two weeks on it, while we spent three months working on fully supporting the Xiaomi ecosystem.

Acknowledgements

We thank our anonymous reviewers for their constructive feedback. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 850868) and DARPA under HR001119S0089-AMP-FP-034. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of our sponsors.

References

[Abr21] Ajin Abraham. Mobile Security Framework. https://github.com/MobSF/Mobile-Security-Framework-MobSF, 2021.

[And21] Android. Capture and Read Bug Reports. https://developer.android.com/studio/debug/bug-report, 2021.

[ASA21] Nordic Semiconductor ASA. nRF Connect for Mobile. https://play.google.com/store/apps/details?id=no.nordicsemi.android.mcp, 2021.

[ATR20] Daniele Antonioli, Nils O. Tippenhauer, and Kasper Rasmussen. Key Negotiation Downgrade Attacks on Bluetooth and Bluetooth Low Energy. *ACM Transactions on Privacy and Security*, 23(3), 2020.

[Aut21] Python Cryptographic Authority. Python Cryptography. https://cryptography.io/en/latest/, 2021.

[AV-15] AV-TEST Team. Analysis of Fitbit Vulnerabilities. https://www.av-test.org/fileadmin/pdf/avtest_2016-04_fitbit_vulnerabilities.pdf, 2015.

[BL21] Henrik Blidh and David Lechner. Bleak. https://pypi.org/project/bleak/, 2021.

[BN20] Eli Biham and Lior Neumann. Breaking the Bluetooth Pairing – The Fixed Coordinate Invalid Curve Attack. In *Selected Areas in Cryptography (SAC '19)*, pages 250–273. Springer International Publishing, 01 2020.

[BRTV16] Benjamin Bichsel, Veselin Raychev, Petar Tsankov, and Martin Vechev. Statistical Deobfuscation of Android Applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, page 343–355. Association for Computing Machinery, 2016.

[BRW04] Mihir Bellare, Phillip Rogaway, and David Wagner. The EAX Mode of Operation. In *International Workshop on Fast Software Encryption*, pages 389–407. Springer, 2004.

[BtSC21] Philippe Biondi and the Scapy Community. Scapy. https://scapy.net/, 2021.

[BXC] Ltd Beijing Xiaomi Co. Mi Home for iOS. https://apps.apple.com/us/app/mi-home-xiaomi-smart-home/id957323480.

[CHMS14] Britt Cyr, Webb Horn, Daniela Miao, and Michael Specter. Security Analysis of Wearable Fitness Devices (Fitbit). *Massachusetts Institute of Technology*, 1, 2014.

[Co.] Anhui Huami Information Technology Co. Mi Fit for Android. https://play.google.com/store/apps/details?id=com.xiaomi.hm.health&hl=en_US&gl=US.

[Cor20] International Data Corporation. Shipments of Wearable Devices Leap to 125 Million Units, Up 35.1% in the Third Quarter, According to IDC. https://www.idc.com/getdoc.jsp?containerId=prUS47067820, 2020.

[Cre19] Creotiv. Mi Band 2 - Python Library. https://github.com/creotiv/MiBand2, 2019.

[CWP+18] Jiska Classen, Daniel Wegemer, Paul Patras, Tom Spink, and Matthias Hollick. Anatomy of a Vulnerable Fitness Tracking System: Dissecting the Fitbit Cloud, App, and Firmware. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies (IMWUT '18)*, 2(1):1–24, 2018.

[DG19] Anthony Desnos and Geoffroy Guegue. Androguard. https://github.com/androguard/androguard, 2019.

[DPCM16] Aveek K. Das, Parth H. Pathak, Chen-Nee Chuah, and Prasant 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 (HotMobile '16)*, page 99–104, 2016.

[FCS+17] Hossein Fereidooni, Jiska Classen, Tom Spink, Paul Patras, Markus Miettinen, Ahmad-Reza Sadeghi, Matthias Hollick, and Mauro Conti. Breaking Fitness Records Without Moving: Reverse Engineering and Spoofing Fitbit. In *Research in Attacks, Intrusions, and Defenses - 20th International Symposium (RAID '17)*, pages 48–69, 2017.

[Fen20] Caleb Fenton. Simplify Android Deobfuscator. https://github.com/CalebFenton/simplify, 2020.

[FFM+17] Hossein Fereidooni, Tommaso Frassetto, Markus Miettinen, Ahmad-Reza Sadeghi, and Mauro Conti. Fitness Trackers: Fit for Health but Unfit for Security and Privacy. In *2017 IEEE/ACM International Conference on Connected Health: Applications, Systems and Engineering Technologies (CHASE '17)*, pages 19–24. IEEE, 2017.

[Fita] Inc. Fitbit. Fitbit app for Android. https://play.google.com/store/apps/details?id=com.fitbit.FitbitMobile&hl=en_US&gl=US.

[Fitb] Inc. Fitbit. Fitbit Homepage. https://www.fitbit.com/global/it/home.

[FKS16] Kassem Fawaz, Kyu-Han Kim, and Kang G. Shin. Protecting Privacy of BLE Device Users. In *25th USENIX Security Symposium (USENIX Security '16)*, pages 1205–1221. USENIX Association, 2016.

[Fou20a] Mozilla Foundation. Privacy Not Included - Amazfit Fitness trackers. https://foundation.mozilla.org/en/privacynotincluded/amazfit-fitness-trackers/, 2020.

[Fou20b] Mozilla Foundation. Privacy Not Included - Mi Band 5. https://foundation.mozilla.org/en/privacynotincluded/mi-band-5/, 2020.

[Fou20c] Mozilla Foundation. Privacy Not Included - Mi Band 6. https://foundation.mozilla.org/en/privacynotincluded/mi-band-6/, 2020.

[Fre21] Freeyourgadget. Gadgetbridge a free and cloudless replacement for your gadget vendors' closed source Android applications. https://codeberg.org/Freeyourgadget/Gadgetbridge, 2021.

[GCW20] Matheus Garbelini, Sudipta Chattopadhyay, and Chundong Wang. SweynTooth: Unleashing Mayhem over Bluetooth Low Energy. https://asset-group.github.io/disclosures/sweyntooth/sweyntooth.pdf, 2020.

[GDS16] Rohit Goyal, Nicola Dragoni, and Angelo Spognardi. Mind the Tracker You Wear: A Security Analysis of Wearable Health Trackers. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing (SAC '16), Pisa, Italy*, page 131–136, 2016.

[Gie18] Dennis Giese. Having fun with IoT: Reverse Engineering and Hacking of Xiaomi IoT Devices. https://dontvacuum.me/talks/DEFCON26/DEFCON26-Having_fun_with_IoT-Xiaomi.html, 2018.

[HCR20] Dennis Heinze, Jiska Classen, and Felix Rohrbach. MagicPairing: Apple's Take on Securing Bluetooth Peripherals. In *Proceedings of the 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, pages 111–121, 2020.

[HPK16] Andrew Hilts, Christopher Parsons, and Jeffrey Knockel. Every Step You Fake: A Comparative Analysis of Fitness Tracker Privacy and Security. *Open Effect Report*, 76(24):31–33, 2016.

[Hua20] Huami. Huami Privacy Note. https://upload-cdn.huami.com/tposts/9250, 2020.

[iBo21] iBotPeaches. Apktool. https://ibotpeaches.github.io/Apktool/, 2021.

[Inca] Huami Inc. Amazfit Homepage. https://www.amazfit.com/en/.

[Incb] Huami Inc. Huami Homepage. https://www.huami.com/investor.

[Incc] Huami Inc. Mi Fit for iOS. https://apps.apple.com/us/app/mi-fit/id938688461.

[Incd] Huami Inc. Zepp (formerly Amazfit) for Android. https://play.google.com/store/apps/details?id=com.huami.watch.hmwatchmanager&hl=en_US&gl=US.

[Ince] Huami Inc. Zepp (formerly Amazfit) for iOS. https://apps.apple.com/us/app/zepp-formerly-amazfit/id1127269366.

[Incf] Xiaomi Inc. Mi Band Homepage. https://www.mi.com/global/miband.

[Incg] Xiaomi Inc. Mi Home for Android. https://play.google.com/store/apps/details?id=com.xiaomi.smarthome&hl=it&gl=US.

[Inc19] Armis Inc. BLEEDINGBIT: The Hidden Attack Surface Within BLE Chips. https://armis.com/bleedingbit/, 2019.

[IT17] Taher Issoufaly and Pierre U. Tournoux. BLEB: Bluetooth Low Energy Botnet for Large Scale Individual Tracking. In *1st International Conference on Next Generation Computing Applications (NextComp '17)*, pages 115–120, 2017.

[Jas16] Sławomir Jasek. Gattacking Bluetooth Smart Devices. Black Hat USA Conference, 2016.

[KCB21] Platon Kotzias, Juan Caballero, and Leyla Bilge. How Did That Get In My Phone? Unwanted App Distribution on Android Devices. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 53–69, 2021.

[KHG21] Maxim Krasnyansky, Marcel Holtmann, and Fabrizio Gennari. Hcidump. https://manpages.debian.org/testing/bluez-hcidump/hcidump.1.en.html, 2021.

[KS18] Aleksandra Korolova and Vinod Sharma. Cross-App Tracking via Nearby Bluetooth Low Energy Devices. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy (CODASPY '18)*, page 43–52. Association for Computing Machinery, 2018.

[Lak21a] Ravie Lakshmanan. Attention! FluBot Android Banking Malware Spreads Quickly Across Europe. https://thehackernews.com/2021/04/attention-flubot-android-banking.html, 2021.

[Lak21b] Ravie Lakshmanan. Over 750.000 Users Downloaded New Billing Fraud Apps From Google Play Store. https://thehackernews.com/2021/04/over-750000-users-download-new-billing.html, 2021.

[Lak21c] Ravie Lakshmanan. WhatsApp-based wormable Android malware spotted on the Google Play Store. https://thehackernews.com/2021/04/whatsapp-based-wormable-android-malware.html, 2021.

[Mis21a] Sandeep Mistry. Bleno. https://github.com/noble/bleno, 2021.

[Mis21b] Sandeep Mistry. Noble. https://github.com/noble/noble, 2021.

[MSBK21] Rene Mayrhofer, Jeffrey V. Stoep, Chad Brubaker, and Nick Kralevich. The Android Platform Security Model. *ACM Transactions on Privacy and Security*, 24(3), 2021.

[New21] Kimi Newt. Pyshark. https://pypi.org/project/pyshark/, 2021.

[NW97] Roger M. Needham and David J. Wheeler. TEA Extensions. *Report, Cambridge University*, 1997.

[NyZD+14] Muhammad Naveed, Xiao yong Zhou, Soteris Demetriou, XiaoFeng Wang, and Carl 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 '14)*, 2014.

[of19] Freezed or frozen. Pymb1a. https://github.com/freezed-or-frozen/pymb1a, 2019.

[Ojh18] Yogesh Ojha. I hacked MiBand 3, and here is how I did it. Part I. https://medium.com/@yogeshoa/i-hacked-xiaomi-miband-3-and-here-is-how-i-did-it-43d68c272391, 2018.

[Ple21] Pleavr. Frida. https://frida.re/, 2021.

[Pro22] Mitmproxy Project. Mitmproxy. https://mitmproxy.org/, 2022.

[RCB13] Mahmudur Rahman, Bogdan Carbunar, and Madhusudan Banik. Fit and Vulnerable: Attacks and Defenses for a Health Monitoring Device. *ArXiv*, abs/1304.5672, 2013.

[RCT16] Mahmudur Rahman, Bogdan Carbunar, and Umut Topkara. Secure Management of Low Power Fitness Trackers. *IEEE Transactions on Mobile Computing*, 15(2):447–459, 2016.

[Rya13a] Mike Ryan. Bluetooth: With Low Energy Comes Low Security. In *7th USENIX Workshop on Offensive Technologies (WOOT 13)*. USENIX Association, 2013.

[Rya13b] Mike Ryan. Bluetooth: With Low Energy Comes Low Security. In *Proceedings of the 7th USENIX Conference on Offensive Technologies (WOOT'13)*, page 4. USENIX Association, 2013.

[Sam20] Samczsun. Java Deobfuscator. https://github.com/java-deobfuscator/deobfuscator, 2020.

[Sat21] Satcar77. Mi Band 4 - Python Library. https://github.com/satcar77/miband4, 2021.

[SB19] Pallavi Sivakumaran and Jorge 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 *28th USENIX Security Symposium (USENIX Security '19)*, pages 1–18. USENIX Association, 2019.

[Sem21] Dialog Semiconductor. Dialog Semiconductor. https://www.dialog-semiconductor.com/, 2021.

[SIG19] Bluetooth SIG. Bluetooth Core Specification v5.2. https://www.bluetooth.org/docman/handlers/downloaddoc.ashx?doc_id=478726, 2019.

[SJMdR16] Maarten Schellevis, Bart Jacobs, Carlo Meijer, and Joeri de Ruiter. Getting Access to Your Own Fitbit Data. Master's thesis, Radboud University, 2016.

[Sky21] Skylot. JADX. https://github.com/skylot/jadx, 2021.

[SPA19] Mohit Sethi, Aleksi Peltonen, and Tuomas Aura. Misbinding Attacks on Secure Device Pairing and Bootstrapping. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security (Asia CCS '19)*, page 453–464. Association for Computing Machinery, 2019.

[Sta21] Statcounter. Mobile and Tablet Android Version Market Share Worldwide (Nov 2020 - Nov 2021). https://gs.statcounter.com/android-version-market-share/mobile-tablet/worldwide, 2021.

[TdHV+21] Sam L. Thomas, Jan V. den Herrewegen, Georgios Vasilakis, Zitai Chen, Mihai Ordean, and Flavio D. Garcia. Cutting Through the Complexity of Reverse Engineering Embedded Devices. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021:360–389, 2021.

[Vir21] Virtualabs. BtleJack: a new Bluetooth Low Energy swiss-army knife. https://github.com/virtualabs/btlejack, 2021.

[vTPFG21] Maximilian von Tschirschnitz, Ludwig Peuckert, Franzen Franzen, and Jens Grossklags. Method Confusion Attack on Bluetooth Pairing. In *2021 IEEE Symposium on Security and Privacy (SP '21)*, pages 213–228. IEEE Computer Society, 2021.

[WGP21] Lennert Wouters, Benedikt Gierlichs, and Bart Preneel. My Other Car is Your Car: Compromising the Tesla Model X Keyless Entry System. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(4):149–172, 2021.

[WHZ+20] Jiliang Wang, Feng Hu, Ye Zhou, Yunhao liu, Hanyi Zhang, and Zhe Liu. BlueDoor: Breaking the Secure Information Flow via BLE Vulnerability. In *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services (MobiSys '20)*, page 286–298. Association for Computing Machinery, 2020.

[Wir21] Wireshark. Wireshark. https://www.wireshark.org/, 2021.

[WMA+19] Lennert Wouters, Eduard Marin, Tomer Ashur, Benedikt Gierlichs, and Bart Preneel. Fast, Furious and Insecure: Passive Keyless Entry and Start Systems in Modern Supercars. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019(3):66–85, 2019.

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

[Yog19] Yogeshojha. Mi Band 3 - Python Library. https://github.com/yogeshojha/MiBand3, 2019.

[ZWD+20a] Yue Zhang, Jian Weng, Rajib Dey, Yier Jin, Zhiqiang Lin, and Xinwen Fu. Breaking Secure Pairing of Bluetooth Low Energy Using Downgrade Attacks. In *29th USENIX Security Symposium (USENIX Security '20)*, pages 37–54. USENIX Association, 2020.

[ZWD+20b] Yue Zhang, Jian Weng, Rajib Dey, Yier Jin, Zhiqiang Lin, and Xinwen Fu. Breaking Secure Pairing of Bluetooth Low Energy Using Downgrade Attacks. In *29th USENIX Security Symposium (USENIX Security '20)*, pages 37–54. USENIX Association, 2020.

[ZWLZ19] Chaoshun Zuo, Haohuang Wen, Zhiqiang Lin, and Yinqian 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 '19), London, United Kingdom*, page 1469–1483, 2019.

Appendix

Table 6: Standard and vendor-specific services and characteristics exposed by every Mi Band GATT server. The first column shows the service name, and the second the characteristic name. The third column indicates the characteristic's access control policy (R = read, W = write, WWR = write without response, N = notify, I = indicate). The last column indicates which Mi Band supports the current characteristic.

Service	Characteristic	Properties	MB
Generic Access 0x1800	Device Name 0x2A00	R	All
	Appearance 0x2A01	R	All
	Peripheral Preferred Connection Params. 0x2A04	R	All
Generic Attribute 0x1801	Service Changed 0x2A05	I, R	2, 3
Device Information 0x180A	Serial Number String 0x2A25	R	2, 3
	Hardware Revision String 0x2A27	R	All
	Software Revision String 0x2A28	R	All
	System ID 0x2A23	R	All
	PnP ID 0x2A50	R	All
	Unknown 00000014-0000-3512-2118-0009af100700	N, WWR	5, 6
Alert Notification Service 0x1811	New Alert 0x2A46	W	2, 3
	Alert Notification Control Point 0x2A44	N, R, W	All
Immediate Alert 0x1802	Alert Level 0x2A06	WWR	All
Heart Rate 0x180D	Heart Rate Measurement 0x2A37	N	All
	Heart Rate Control Point 0x2A39	R, W	All
Human Interface Device 0x1812	Protocol Mode 0x2A4E	R, WWR	5
	Report Map 0x2A4B	R	5
	HID Information 0x2A4A	R	5
	HID Control Point 0x2A4D	WWR	5
	Boot Keyboard Input Report 0x2A22	N, R	5
	Boot Keyboard Output Report 0x2A32	N, R	5
Battery Service 0x180F	Battery Level 0x2A19	N, R	5
Firmware Service 00001530-0000-3512-2118-0009af100700	Firmware 00001531-0000-3512-2118-0009af100700	N, W	All
	Firmware Data 00001532-0000-3512-2118-0009af100700	WWR	All
Vendor Specific 00003802-0000-1000-8000-00805f9b34fb	Unknown 00004a02-0000-1000-8000-00805f9b34fb	N, R, W	4

Table 7: Huami services and characteristics exposed by every Mi Band GATT server. The first column shows the service name, and the second the characteristic name. The third column indicates the characteristic's access control policy (R = read, W = write, WWR = write without response, N = notify, I = indicate). The last column indicates which Mi Band supports the current characteristic.

Service	Characteristic	Properties	MB
Anhui Huami Information Technology Co. 0000fee0-0000-1000-8000-00805f9b34fb	Current Time 0x2A2B	N, R, W	All
	Chunked Transfer 00000020-0000-3512-2118-0009af100700	N, R, WWR	All
	Unknown 00000001-0000-3512-2118-0009af100700	N, WWR	All
	Unknown 00000002-0000-3512-2118-0009af100700	N	All
	Configuration 00000003-0000-3512-2118-0009af100700	N, WWR	All
	Peripheral Preferred Connection Params. 0x2A04	N, R, WWR	All
	Unknown 00000004-0000-3512-2118-0009af100700	N, WWR	All
	Activity Data 00000005-0000-3512-2118-0009af100700	N	All
	Battery 00000006-0000-3512-2118-0009af100700	N, R	All
	Steps 00000007-0000-3512-2118-0009af100700	N, R	All
	User Settings 00000008-0000-3512-2118-0009af100700	N, W	All
	Device Event 00000010-0000-3512-2118-0009af100700	N	All
	Unknown 0000000e-0000-3512-2118-0009af100700	W	3, 5, 6
	Unknown 0000000f-0000-3512-2118-0009af100700	N, WWR	3, 5, 6
	Unknown 00000011-0000-3512-2118-0009af100700	N, R, WWR	4, 5, 6
	Audio 00000012-0000-3512-2118-0009af100700	N, R, WWR	4, 5, 6
	Audio Data 00000013-0000-3512-2118-0009af100700	N, R, W	4, 5, 6
	Unknown 00000016-0000-3512-2118-0009af100700	N, WWR	5
	Unknown 00000017-0000-3512-2118-0009af100700	N, WWR	5, 6
Anhui Huami Information Technology Co. 0000fee1-0000-1000-8000-00805f9b34fb	Auth 00000009-0000-3512-2118-0009af100700	N, R, WWR	All
	Jawbone 0000fedd-0000-1000-8000-00805f9b34fb	W	All
	Coin, Inc. 0000fede-0000-1000-8000-00805f9b34fb	R	All
	Design SHIFT 0000fedf-0000-1000-8000-00805f9b34fb	R	All
	Apple, Inc. 0000fed0-0000-1000-8000-00805f9b34fb	R, W	All
	Apple, Inc. 0000fed1-0000-1000-8000-00805f9b34fb	R, W	All
	Apple, Inc. 0000fed2-0000-1000-8000-00805f9b34fb	R	All
	Apple, Inc. 0000fed3-0000-1000-8000-00805f9b34fb	R, W	All
	Unknown 0000fec1-0000-3512-2118-0009af100700	N, R, W	2, 3, 5, 6