Anatomy of a Vulnerable Fitness Tracking System: Dissecting the Fitbit Cloud, App, and Firmware

JISKA CLASSEN, TU Darmstadt, Germany
DANIEL WEGEMER, TU Darmstadt, Germany
PAUL PATRAS, University of Edinburgh, Scotland, UK
TOM SPINK, University of Edinburgh, Scotland, UK
MATTHIAS HOLLICK, TU Darmstadt, Germany

Fitbit fitness trackers record sensitive personal information, including daily step counts, heart rate profiles, and locations visited. By design, these devices gather and upload activity data to a cloud service, which provides aggregate statistics to mobile app users. The same principles govern numerous other Internet-of-Things (IoT) services that target different applications. As a market leader, Fitbit has developed perhaps the most secure wearables architecture that guards communication with end-to-end encryption. In this article, we analyze the complete Fitbit ecosystem and, despite the brand's continuous efforts to harden its products, we demonstrate a series of vulnerabilities with potentially severe implications to user privacy and device security. We employ a range of techniques, such as protocol analysis, software decompiling, and both static and dynamic embedded code analysis, to reverse engineer previously undocumented communication semantics, the official smartphone app, and the tracker firmware. Through this interplay and in-depth analysis, we reveal how attackers can exploit the Fitbit protocol to extract private information from victims without leaving a trace, and wirelessly flash malware without user consent. We demonstrate that users can tamper with both the app and firmware to selfishly manipulate records or circumvent Fitbit's walled garden business model, making the case for an independent, user-controlled, and more secure ecosystem. Finally, based on the insights gained, we make specific design recommendations that can not only mitigate the identified vulnerabilities, but are also broadly applicable to securing future wearable system architectures.

CCS Concepts: • **Security and privacy → Security protocols**; **Software reverse engineering**; *Mobile and wireless security*; Software security engineering; Social aspects of security and privacy;

Additional Key Words and Phrases: Wearables, health, firmware reverse engineering, Nexmon

ACM Reference Format:
Jiska Classen, Daniel Wegemer, Paul Patras, Tom Spink, and Matthias Hollick. 2018. Anatomy of a Vulnerable Fitness Tracking System: Dissecting the Fitbit Cloud, App, and Firmware. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 1, 1, Article 42 (March 2018), 24 pages. https://doi.org/0000001.0000001

1 INTRODUCTION

The popularity of fitness wristbands and smartwatches has grown significantly over previous years. Recent studies estimate 830 million wearable gadgets will be shipped by 2020 [1]. This confirms the consumers' increasing appeal for monitoring their daily activity patterns and is expected to have broader implications on healthcare [2], workplace provisions [3], and the judiciary [4]. In the first half of 2017, the wearables market was dominated by Apple, Xiaomi, and Fitbit [5], the latter having sold more than 70 million devices since their inception [6].

Authors' addresses: Jiska Classen, jclassen@seemoo.de, TU Darmstadt, Germany; Daniel Wegemer, dwegemer@seemoo.de, TU Darmstadt, Germany; Paul Patras, ppatras@inf.ed.ac.uk, University of Edinburgh, Scotland, UK; Tom Spink, tspink@inf.ed.ac.uk, University of Edinburgh, Scotland, UK; Matthias Hollick, mhollick@seemoo.de, TU Darmstadt, Germany.

© 2018 Association for Computing Machinery.
This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, https://doi.org/0000001.0000001.

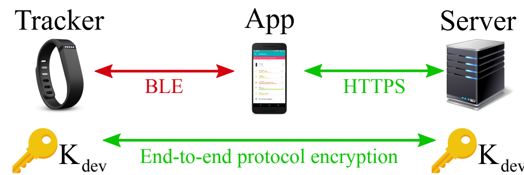


Fig. 1. Fitbit's secure communication model: tracker communicates with smartphone app over unsecured BLE (red); mobile device interacts with Fitbit server using HTTPS. Symmetric end-to-end encryption supported by default.

Wearable devices—and the mobile apps that enable their users to interact with them—record, store, and exchange sensitive personal information with remote servers. This includes age, gender, weight, heart rate, distances traveled, geo-position, duration of activity periods, and sleep habits. Fitness data is stored locally on trackers, yet the user can only examine personal statistics after the tracker synchronizes with the manufacturer's cloud. The cloud service fuses sensor readings, computes both activity and sleep summaries, and pushes these to the user's app. This model can facilitate additional revenue streams, if the user consents to share fitness data with third parties. The manufacturer, however, must safeguard user privacy and security. This is both critical, given the potentially delicate nature of user records, and challenging to implement, given the ever-expanding attack surface—the wireless communication between trackers and smartphones is prone to eavesdropping, risks of compromising mobile apps exist [7], and a range of man-in-the-middle (MITM) attacks could be mounted at access networks using open-source tools [8]. As fitness trackers are rushed to market, the research community is actively investigating different attack vectors, albeit thus far focusing only in isolation on different aspects of the end-to-end communication, firmware upgrade verification, or hardware exploits [9–14].

In this article, we take a holistic approach and investigate, in-depth, the security and privacy of the *complete Fitbit ecosystem*, namely, tracker firmware, mobile app, and communication protocols as shown in Figure 1. User privacy and security are of paramount importance to this leading brand, which is reflected in the design of its end-to-end architecture. Despite Fitbit's efforts to strengthen their products against attacks, we show the not straightforward means to reverse engineer, modify, and exploit all of the system's components. To this end, we use our extensive repertoire of protocol and code analysis techniques to enable us to modify tracker firmware and allow run-time debugging over testing pins. In addition, we construct a virtualized instance of a real smartphone, to sniff Bluetooth Low Energy (BLE) packets exchanged with trackers. We further introduce changes to the official smartphone app, such that it can snoop on and modify encrypted Fitbit traffic, and program a custom Android demo app featuring cloud independence. Building upon this multifaceted infrastructure, we reveal substantial vulnerabilities with severe real-world consequences for the operation of recent Fitbit models, which impact their users' privacy, including stalking, harassment, intrusive health monitoring, denial-of-service, IoT weaponization, and business model breach. More precisely, we make the following **key contributions:**

(1) **Privacy leakage:** We demonstrate how attackers can exploit information present in activity reports extracted from nearby victim trackers to associate these with a controlled account and subsequently obtain all recently recorded fitness measurement data. In addition, we show that an attacker can acquire authentication credentials for any device, then replay these to gain access to tracker commands. This includes triggering "live mode" operation, thereby forcing target trackers to leak unencrypted data in real-time.
(2) **Firmware customization:** We compromise the firmware update protocol and demonstrate that the firmware running on selected device types within wireless range can be modified. We craft and inject malicious firmware without consent, which disables all security mechanisms supported, that is, authentication and encryption.

(3) **Subversion of Fitbit cloud:** We modify the official smartphone app and discover developer options that can be enabled via configuration files for all original app versions. We reveal that the Fitbit server endpoint is reprogrammable, and argue that it is possible to develop an independent custom service, which subverts the Fitbit cloud and gives full control over user privacy.

Many of the vulnerabilities described in this article could only be detected by examining all the components of this ecosystem jointly, for example, Fitbit server logic, smartphone app, and tracker firmware. Despite the complexity of the steps required to perform these attacks, our findings are reproducible by end-users in possession of a malicious app. Along with this article, we release a smartphone demo app that implements "cloud independence" features and a tool chain that facilitates Nexmon-based firmware modifications [15, 16].

On the one hand our work showcases a rigorous approach, which developers can follow when attempting to reverse engineer IoT infrastructure, in order to assess the presence of security flaws. The design recommendations we make based on our findings serve as best practices for protecting user data generated by activity trackers and pave the way to follow-on research on strengthening fitness tracker systems; recent comparative analyses of fitness tracker security indicate that there are indeed striking similarities between the solutions of different vendors [10]. On the other hand, the type of ecosystem we analyze is not unique to Fitbit or fitness tracking alternatives, as numerous IoT applications are governed by similar gadget↔app↔cloud paradigms where the smartphone acts as a mediator between the user, device, and service. These include wearable blood pressure and glucose measurement systems that enable remote heathcare monitoring [17, 18], as well as mobile based home automation systems by which users can monitor and control electrical appliances [19]. Vulnerability analyses of such systems are sparse, but existing ones range from smart lighting system attacks to smart homes turning into espionage environments [20, 21]. Therefore, the synergy between our analyses, the general findings, and lessons learned, are broadly applicable to other IoT technology vendors, many of whom ship products considerably less secure than those produced by Fitbit [22].

Responsible disclosure: Prior to submitting this article we have disclosed all of the identified security vulnerabilities to Fitbit and in October 2017, the company began rolling out patches to address the issues our work highlights. We thank Fitbit for acknowledging us in their patches [23].

The remainder of this article is structured as follows. In Section 2, we describe the basic communication model, and build on this in Section 3 by describing adversary and attack scenarios. We then present our reverse-engineering findings in the following three sections: the protocol in Section 4, the official and demo smartphone app in Section 5, and the tracker's firmware in Section 6. In Section 7 we specifically discuss design recommendations that aim to assist in building more secure trackers and IoT protocols in the future, and we review related work in Section 8. Finally, we summarize and conclude our work in Section 9.

2 THE FITBIT COMMUNICATION PARADIGM

In this section, we introduce the end-to-end communication paradigm governing the Fitbit tracker↔app↔server ecosystem. The understanding of protocol semantics, as well as authentication and encryption mechanisms, is essential to describing attack scenarios and vulnerabilities revealed in later sections.

The main purpose of the Fitbit communication model is to allow trackers to send activity records to the vendor's server. By default, such records are encrypted. The server replies with statistics and settings, which are again encrypted by default. Activity records are classified as either "microdumps" or "megadumps". The former contain only tracker metadata, whilst the latter include detailed step counts, traveled distances, floors climbed, and other metrics captured since the last synchronization. These "megadumps" also contain an activity summary.

Given battery and size constraints, trackers are only equipped with BLE, and rely on a mobile app or computer software (such as the official Windows/Android app or the open-source Galileo tool [24]), to forward activity data to the Fitbit server. We illustrate this communication chain in Figure 1.

Users are permitted to push records and access activity statistics via a Web front-end, or through the official Web API used by the app, after following the authentication procedure shown in Figure 2. After user login (1), a local tracker pairing is performed. Crucially, anyone within wireless range of a device can perform this without being logged in, unless they are using the official Fitbit app (2). This is followed by an association process, which includes tapping on the tracker or entering into the smartphone a code that the tracker displays, thereby authorizing access to the wearable's data stored on the server (3). The associated user can request server-generated tracker-specific authentication credentials to authenticate the app towards the tracker (4) for more powerful local commands (5). Such commands include setting alarms, reading the device's internal memory, and starting "live mode" operation. Live mode is used to directly display user statistics in the app dashboard in real-time, without interacting with the server. Although the local pairing is not secured, recent trackers (normally) encrypt the remote association protocol of a device to a user account.

While app↔server traffic is secured with HTTPS, anyone can initiate the local wireless BLE tracker↔app communication. Under normal circumstances, the unsecured tracker↔app communication should not be a security issue. This is because the tracker↔server communication is expected to be end-to-end encrypted. Symmetric encryption with a manufacturer-generated 128 bit secret device key K_{dev} is performed on all trackers. Older devices employ eXtended Tiny Encryption Algorithm (XTEA), while newer use Advanced Encryption Standard (AES), in encrypt-then-authenticiate-then-translate (EAX) mode respectively.

Yet, models before 2015 employ optional encryption only—when buying old trackers online, the chances are high of coming across one that still needs to be migrated to the "always encrypt" mode of operation. Furthermore, recent firmware revisions have an internal "disable encryption" switch [13]. Previously, when the ability to disable encryption was discovered, this caused the servers to reply in plaintext. However, in the meantime, Fitbit rolled out a strict "trial encryption" process to fix these affected trackers. This caused plaintext trackers to regularly try to send an encrypted dump to the server, and, if they receive encrypted server replies in return, they will persistently encrypt, and the server will refuse hardware-manipulated "disable encryption" plaintext communication. Although Fitbit has taken measures to refuse plaintext communication, operating trackers in plaintext mode helps to understand the meaning of the data routinely transmitted.

3 ADVERSARIAL MODEL

Next, we discuss the potential capabilities an attacker might employ to damage the Fitbit ecosystem (Section 3.1) and present a set of critical attack scenarios, as well as their implications (Section 3.2). In the following Sections 4–6, we present an in-depth overview of the reverse-engineering efforts we have undertaken, to demonstrate

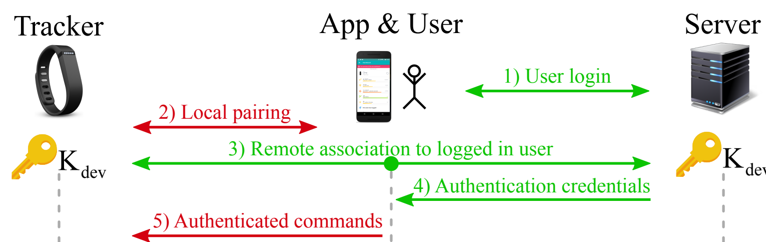


Fig. 2. Authentication model: local wireless pairing vs. association of a tracker to a user account. Local BLE commands (marked in red) are not cryptographically secured.

which parts of the Fitbit architecture can be compromised, even though it is regarded as one of the most secure systems of its kind [22].

3.1 Attacker Capabilities

We expect that an attacker has already set up an account on the Fitbit cloud, and may either be temporally within wireless range of a victim's tracker, or be able to visually inspect the original device packaging in a store. Attackers do not require physical access to the fitness tracker hardware, they can simply use a customized smartphone app instead. Scenarios running a Fitbit-independent infrastructure require additional resources.

Some of the weaknesses we reveal assume having copies of valid tracker firmware (which can be obtained through an update process or a memory readout vulnerability), the Fitbit app binary (which is available online), and app/firmware (de)compiling tools. We do not explicitly consider physical hardware attacks, though access to the tracker hardware can be useful, for example, to expedite custom firmware code development. Note that the sum of all these capabilities are necessary to develop the attacks we have documented, yet they are not required for execution by an attacker.

Attack	**System**			**Adversary Goal**				**Affected Products**
	Server	Tracker	App	Spying	Financial	DoS	Cloud ind.	
Remote association to attacker account								
Using replay	✓	✓	-	✓	✓	✓	-	All
Using plaintext serial number	✓	✓	-	✓	✓	✓	-	Trackers before 2015*
Using plaintext brute force	✓	✓	-	✓	✓	✓	-	Trackers before 2015*
Wireless commands								
Enter live mode	-	✓	-	✓	-	-	-	All
Live mode sniffing	-	✓	-	✓	-	-	-	All
Set date	-	✓	-	-	✓	✓	✓	Trackers before 2015*
Set alarms	-	✓	-	-	✓	✓	-	Trackers before 2015*
Hardware access including GDB	-	✓	-	-	✓	-	-	Tested on Fitbit Flex
Wireless firmware modifications								
Replay including downgrade	-	✓	-	✓	✓	✓	✓	All
Custom plaintext malware	-	✓	-	✓	✓	✓	✓	Trackers before 2015*
Custom encrypted malware	-	✓	-	✓	✓	✓	✓	All, but memory readout limitation
Hidden debug & development features	-	-	✓	-	-	-	✓	All

Table 1. Vulnerabilities identified in different system components, adversary goals that can be achieved with these vulnerabilities, and Fitbit tracker models affected. *Trackers before 2015: Zip, One, Flex, Charge, Charge HR.

3.2 Attack Scenarios

We categorize adversary goals into **(1) spying** targeting victim's privacy, **(2) financial interests** such as ransomware or monetary rewards, **(3) denial of service** affecting service availability any of the parties involved, and **(4) cloud independence** from Fitbit's official services. We also discuss **(5) non-adversarial applications** such as exploitation of fitness data for medical research. The technical aspects that enable different attacks related to these adversary goals are summarized in Table 1, which comprises only attacks that we have discovered. The detailed explanations of these attacks follow in Sections 4–6. The following paragraphs describe the concrete attack scenarios that can be achieved based on our findings, as well as their implications to both users and the Fitbit business model.

(1) Spying: The spectrum of spying attackers interested in fitness data is large, since the information that trackers collect could be exploited for private, political, or financial motives. Attack scenarios range from stalking, to health analysis and user profiling.

- *Sniffing Ongoing Communication* — Spies can sniff ongoing traffic during routine tracker and app interaction. Local wireless BLE communication is unencrypted, and only selected data types transmitted over BLE are encrypted end-to-end. Remote HTTPS connections can be tampered with if any part of the network connection used is under an attacker's control. We observe that the smartphone app does not employ certificate pinning and the login credentials transferred over HTTPS connections are not further secured. This enables later re-using intercepted credentials for espionage purposes.
- *Proximate Espionage* — Fitbit does not currently recognize proximate wireless espionage, though it is possible to infer, for instance, if someone was really jogging outside. We find that eavesdroppers can connect to any tracker in wireless range and request stored data. Typically, this data is end-to-end encrypted, but encrypted dumps contain metadata revealing the number of synchronizations performed, user activity levels, and the tracker's serial number. In many scenarios, a spy is able to issue authenticated commands to obtain information in plaintext live mode, or sniff live mode data currently being exchanged with the victim. We observe that some trackers further allow fully reading their memory.
- *Remote Espionage* — Spies can obtain all locally encrypted data that trackers store by associating them to a controlled Fitbit account and then triggering synchronization. This decrypts user activity dumps while the attacker can be located virtually anywhere on the Internet. In the case of trackers in plaintext mode, association is simply limited to knowing a valid serial number. Trackers that implement encryption must be initially either within wireless or physical range (depending on the model) for an attacker to capture and replay an association. The victim can re-associate her tracker to a personal account, limiting the espionage period between the last legitimate data synchronization and re-association. Full-featured, persistent espionage can be mounted by configuring an alternative server address inside a victim's smartphone app. This is a more sophisticated MITM attack that does not require any control over the victim's network.
- *Spyware* — Modified firmware can also enable full-featured, persistent espionage on trackers that have been in the attacker's wireless range for minutes. Indeed, we find that spies can flash malware without the victim's interaction, compromising critical security and privacy functions. Trackers that work in plaintext mode accept plaintext firmware updates, which are neither signed nor device specific. For encrypted trackers, the spy requires a valid device-specific encryption key, which we have been able to extract wirelessly on Fitbit Flex and One.

(2) Financial Interests: Next, we consider attacks that can directly generate monetary value.

- *Selling Fitness Data* — Attackers can sell activity records that falsely certify "healthy" lifestyles, as third parties including insurances companies offer rewards to users who prove physically fit by sharing Fitbit trackers data [2]. Flashing a firmware that randomly multiplies all step counts or reports a healthier heart

rate produces enhanced fitness records. Consumers interested in fake fitness services would allow the attacker to analyze or tear-down their tracker, in order to extract all tracker-specific information and automatically generate plausible activity records.

- *Ransomware* — Victims can be extorted to have a compromised tracker working again by exploiting one of the following vulnerabilities. While in wireless range, an attacker can disrupt tracker functionality via an unauthenticated "set date" command, which can make fitness records appear erratic and potentially alert an employer of unusual employee behavior. Likewise, we find that an attacker can set alarms with prior authentication, in order to wake up the victim at arbitrary times. Repeated remote association with the attacker's account stops the victim from collecting any activity summaries. Similar to spyware, ransomware that disables official Fitbit firmware updates and limits tracker functionality can be flashed without user consent.

(3) Denial of Service: Attacks resulting in denial of service not only frustrate users, but also harm Fitbit's reputation, if such attacks can develop at scale.

- *Local Wireless Commands* — Beyond ransomware, dates and alarms on trackers can be manipulated to deny service to Fitbit end-users. For instance, an attacker can exploit authentication replay and local commands to turn a reliable fitness tracker into a vexing random alarm clock.
- *False Firmware Updates* — A malicious firmware update can be used to permanently make trackers non-working by removing the wireless flashing option and other useful functions. It is also possible to harm the Fitbit cloud by flashing mischievous firmware that sends randomized tracker data to the backend servers, which might be hard to filter automatically.
- *Virtually Stealing Trackers* — Vulnerabilities we identify in relation to the remote association procedure can be leveraged to "steal" trackers from the currently associated accounts, thereby disabling synchronization until the victim performs re-association. Such attacks can be performed at scale, since an attacker can re-associate a tracker multiple times using previously recorded traces, as long as the tracker is temporally within wireless range. In addition, association information for plaintext trackers can be deduced from the original packing, or attackers could bruteforce plaintext associations.

(4) Cloud Independence: Making a device independent of the Fitbit cloud can be regarded as an attack to the brand's business model, since users would no longer synchronize their data to the official servers.

- *Full-Featured Separate Infrastructure* — Currently, Fitbit users are constrained to upload their data to the official cloud, to be able to monitor their activity. This is not intrinsic to the Fitbit business model, it is a general practice in the fitness tracker industry [22]. A custom firmware could feature different encryption functions and keys, hence enabling the user to synchronize data with other custom services different from Fitbit. We also observe that the official Android smartphone app has a hidden option that can enable configuring alternative servers. In such a scenario, the look and feel of the app will not change, while older trackers could be migrated to solutions that provide superior encryption solutions. Users could also run such servers themselves or allow someone they trust to do so.
- *Cloud Independence with Reduced Function Set* — Authentication credentials, which are valid for an unlimited time, enable an attacker to issue a subset of commands including live mode operation. This can be used to extract daily activity summaries and average heart rates, and redirect such statistics to an independent cloud, while preserving some interaction with the Fitbit cloud. Users of the newest tracker models, which are not yet completely reverse engineered, could still prefer this reduced feature cloud independence over synchronizing all their data with Fitbit.

(5) Non-Adversarial Applications: Finally, we consider scenarios where there is not necessarily a malicious intent or a personal incentive.

- *Medical Research Sensors* — Trackers have an interesting selection of sensors, ranging from unsophisticated accelerometers to fancier heart rate monitors and GPS receivers. Activity records contain only *interpreted* sensor data, for example the accelerometer only saves aggregated data that includes step counts and floors climbed. While medical research is not an attack vector *per se*, it can benefit from inexpensive platforms that can record and transmit raw personal sensor data. In the case of fitness trackers this could be achieved using firmware modifications. Raw tracker sensor data is more verbose than output of the public sensor interface of even the most recent Fitbit Ionic smartwatch, which for example only provides the overall heart rate but not the pulse curve [25].
- *IoT Security Research* — Given Fitbit's market lead and the reputation of their devices for being relatively secure, we expect other industry players and academic researchers have incentives to continue investigating their security and privacy properties. On the one hand, competitors could learn from Fitbit's mistakes (if identified), in order to strengthen the robustness of their products from initial designs and enlarge customer base. On the other hand, researchers may seek to discover vulnerabilities that can be exploited in other gadgets and expand the body of knowledge in the area of IoT.

4 REVERSE-ENGINEERING AND EXPLOITING THE FITBIT PROTOCOL

In this section, we delve into how an attacker can manipulate the protocol governing the tracker↔app↔server ecosystem. We show that manipulation enables powerful actions, such as retrieving plaintext tracker data or flashing custom firmware. Communication steps triggered by the "vanilla" official smartphone app are as follows:

(1) A tracker is associated to a Fitbit user account as described in Section 4.2.
(2) Dumps (normally encrypted) are synchronized to the server; server responses are forwarded to the tracker by the app.
(3) Summarized data can be retrieved from the server for a user-associated tracker over HTTPS/JSON.
(4) When the user wishes to observe their current activity without delay, the app authenticates towards the tracker as explained in Section 4.3, and receives plaintext live mode data, as discussed in Section 4.4.

We reverse-engineer the tracker↔app protocol, which uses BLE with Generic Attribute Profile (GATT), making wireless operation independent of the proprietary Fitbit USB dongle. The most important wireless BLE/GATT commands are specified by the `gatttool` [26] syntax, which we illustrate graphically throughout, to ease reproducibility. Additionally, we reverse-engineer the app↔server protocol, with a focus on understanding undocumented parts of the Web API. All important wireless commands and cloud independence features are contained in our open-source demo app [15].

In the following subsections, we explain vulnerable protocol parts in the typical order in which they take place. A larger context of all attacks and their technical dependability is given in Table 1.

4.1 Local BLE Pairing

Below, we distinguish between the (1) **local pairing** of a device to the app for wireless communication, and (2) **remote association** of a device to a particular user account. Local pairing means any device within wireless range can be issued data requests and uploads; remote association through the Web API enables an attacker to connect any device to their Fitbit account, if the attacker has the necessary model-dependent device information.

Local pairing is required each time the app establishes a tracker connection. The tracker itself does not verify the commands' origin, thereby accepting any local link establishment, making other attacks more easy to orchestrate. The local pairing procedure is proprietary, which means an attacker attempting standard BLE device pairing will fail. Steps necessary to locally pair a tracker are illustrated in Figure 3a. First, the standard command mode is enabled via a special GATT handle. Afterward, a command to initialize the local link is sent along with connection timing parameters.

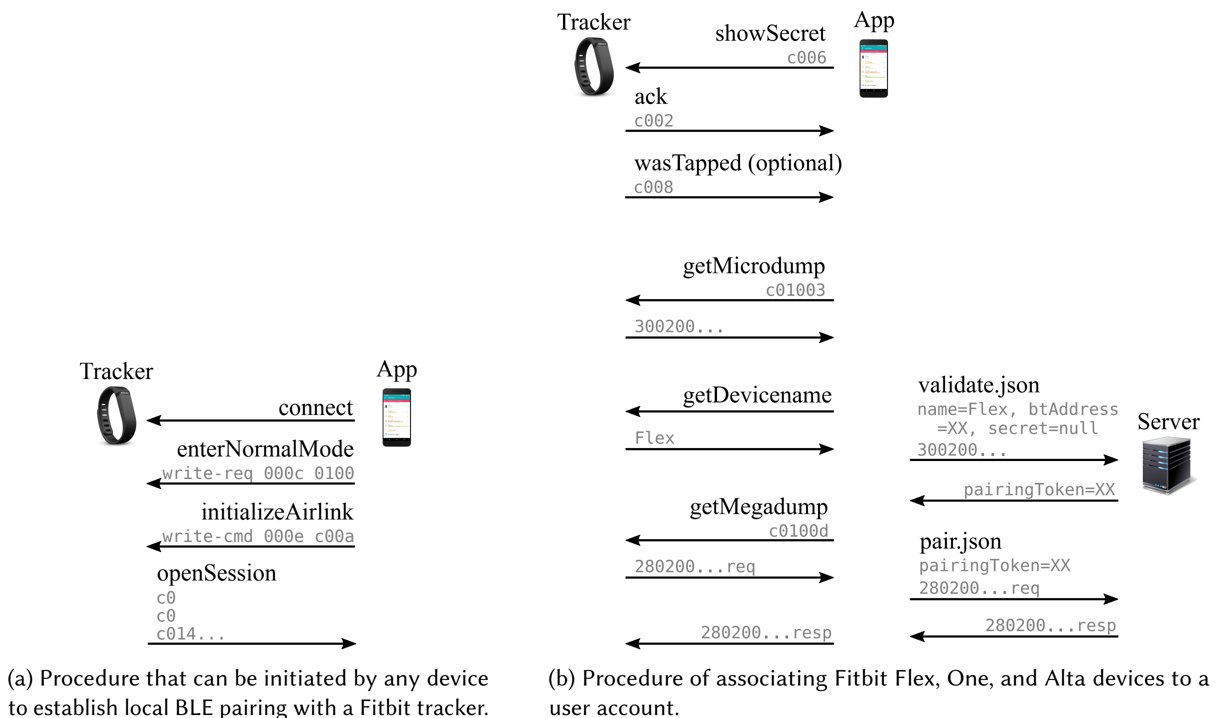
4.2 Associating Oblivious Fitbit Devices to Any User Account

Remote association is performed once to verify that a tracker is physically owned by the user. Re-association to another user account requires the same verification process. Association security depends on protecting the user account itself, as well as properly checking for the physical presence of the tracker. Once associated, the user can request JSON-formatted dump analysis results, and authentication credentials for special wireless commands.

Prior to association, the user who wants to associate the tracker must be logged in. A MITM could be controlling the victim's network or, with temporal smartphone access, employ an alternative server in the app's configuration. User login follows an OAuth standard negotiation, which transfers login credentials in plaintext, solely secured by HTTPS.

> Plaintext login credentials sent to the server are solely secured with HTTPS, and additional mechanisms to protect HTTPS against MITM are missing.

After login, a tracker↔user association can start. Fitbit implements three association mechanisms, as we know from the reverse engineered app. Fitbit Flex, One, and Alta support two of these, i.e. tapping or displaying a code. Both procedures are very similar. Figure 3b illustrates the detailed steps of associating a tracker to a user account.



(a) Procedure that can be initiated by any device to establish local BLE pairing with a Fitbit tracker.

(b) Procedure of associating Fitbit Flex, One, and Alta devices to a user account.

Fig. 3. Local pairing and remote association steps.

In order to communicate with the server, the tracker is first validated, then associated to the initiator's account, as explained next.[1]

Association Step 1. The first validation step involves physical interaction (tapping or keying in the app a code displayed by the tracker) and the reception of a valid microdump. If the tracker only supports tapping, we observe that the validation secret is set to `null`. Fitbit One secrets always start with a zero, hence, only three digits are used, while Fitbit Alta secrets use four digits. Fitbit Alta is using megadumps only, but keeps the same message order. The information retrieved from the tracker is then forwarded to the server to complete the validation. The server can reject this validation step for various reasons, for example if it expects the tracker to perform message encryption but the dump received is in plaintext, or if the secret from the dump does not match the secret entered into the app by the user. In case of a successful validation, the server responds with a `pairingToken`.

Association Step 2. The second step requires a megadump (not depending on the `pairingToken`), which is sent to the server along with the `pairingToken` previously received by the app. The server replies with a megadump response containing user settings required to perform precise calculations on the tracker, including the user's walking stride length and local time zone.

Attacks involving virtually stealing a tracker can be categorized into (1) plaintext, (2) tapping-only, and (3) code-verified. For our security analysis, we modify the official smartphone app as explained in Section 5.1, which enables us to replace selected parts of the association requests towards the server.

First, plaintext trackers can be associated by generating dumps, which enables attackers to associate trackers they have never seen before; valid serial numbers to generate such dumps are printed on the original packing or can be bruteforced.

Second, we observe that any recorded or crafted combination of megadump and microdump of a tapping-only encrypted tracker can be reused to associate the tracker to an attacker's account. The tapping confirmation is only a local wireless command and not forwarded to the server at all.

Third, encrypted trackers requiring a secret can be associated by replaying a recently recorded dump combination. The attacker also needs to know the corresponding secret—limiting the attacker being or having been in tracker view range, with only the former being intended by Fitbit. Outside view range, encrypted tracker dumps and secret combinations can only be accessed via app↔server MITM or by controlling the smartphone app. In either case, both scenarios would provide the attacker with login credentials.

> Attackers can associate trackers to another Fitbit account and, thereby, virtually steal trackers. Security barrier effectiveness depends on encryption and on the tracker's verification method during association.

4.3 Authentication Replay

Reusable authentication credentials are a design flaw in the Fitbit ecosystem. Their intended purpose is authorizing the smartphone app to issue special commands, typically a live mode that is designed for offline usage. Authentication credentials are cryptographically derived from, and verified with, the device key, while not actually transmitting the device key itself. To enable offline usage, the app simply retrieves authentication credentials once, and caches them for reuse indefinitely.

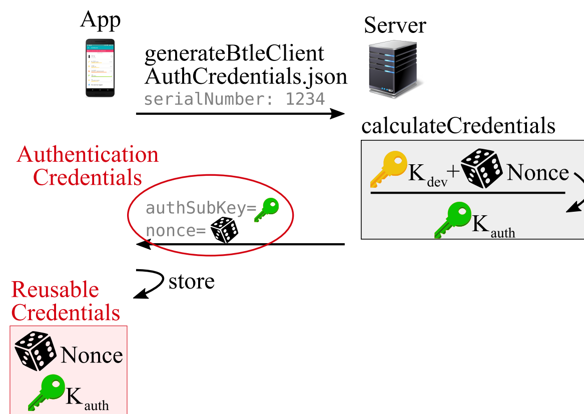
We introduce the term *fake-associated tracker* to indicate a tracker that was associated with a user account against normal procedures, and *real-associated tracker* for one that was associated as intended. Below, we show

[1]Note that Fitbit internally calls the remote association "pairing", but is not related to local wireless pairing. Due to inputting a code read from the screen on some tracker models, the user experience of the remote association feels similar to wireless pairing, but an unsecured wireless pairing is required before.

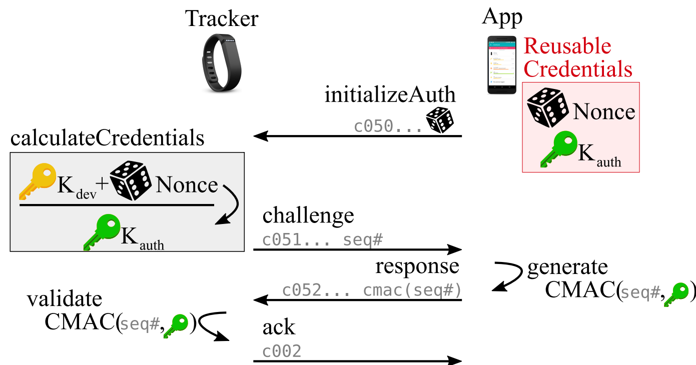
how to obtain reusable authentication credentials for any fake-associated or real-associated tracker. We briefly explain the authentication replay attack discovered in [27], then use our previous fake-association data leakage to authenticate the app towards the tracker as often as required.

4.3.1 Obtaining Authentication Credentials for (Fake-)Associated Trackers Once. The procedure of obtaining authentication credentials from a server is depicted in Figure 4a, which (from the app's perspective) depicts a simple request that causes the server to generate and reply with new authentication credentials. These authentication credentials consist of an authentication subkey K_{auth} and a **reusable seed** internally called *Nonce*. K_{auth} is derived with *Nonce* from the pre-shared secret device key K_{dev} using XTEA Cipher-based Message Authentication Code (CMAC) and padding on older devices [27]; AES CMAC is used on newer devices. The server only sends K_{auth} and *Nonce* to the app, keeping K_{dev} secret.

Obtaining reusable authentication credentials for oblivious trackers is an important attacker goal, which can be achieved by (1) fake-associating a tracker and requesting or (2) observing a request. The first method requires the fake-association from Section 4.2, otherwise the server returns an error message that indicates the tracker with the requested serial number was not found. In the second scenario, authentication credentials are extracted



(a) Obtaining forever valid authentication credentials from the server.



(b) Replaying authentication credentials locally to authenticate a tracker.

Fig. 4. Obtaining one valid pair of authentication credentials as in Figure 4a and reusing it many times locally as in Figure 4b.

by using an MITM setup, enabling the smartphone app's debugging mode, or creating a new request with a custom app. In the official app, a new authentication credential request must be forced, which can be achieved by clearing the app's cache through re-installation. Once the cache is empty, the request appears to an MITM and in the debug log. Our demo app can issue its own authentication credential request. The overall protocol design is secure enough that only sniffing BLE is not sufficient, because K_{auth} is never sent over the air.

Attackers can use a (temporally) associated tracker or MITM setup to obtain valid authentication credentials. Since the device key is persistent, the obtained tracker-specific authentication credentials stay valid forever.

4.3.2 Repeatable Local Authentication Mechanism. Local authentication between tracker and app is shown in Figure 4b. The protocol is designed to prove that both the tracker and the app know the authentication credentials without transmitting the full credentials. The app only sends the *Nonce* to the tracker, from which the tracker can derive K_{auth} using its locally-known K_{dev}. Based on this, the tracker and app can check if they have the same K_{auth} and perform mutual authentication. Mutual authentication consists of three steps as briefly described and discovered in [27]: (1) authentication initialization, (2) authentication challenge, and (3) authentication response. We initialize the authentication and generate a valid challenge response by reusing authentication credentials, and retrieve the incrementing number *seq#* from the authentication challenge. Code for this can be extracted from the official app, and works for both XTEA- and AES-based trackers.

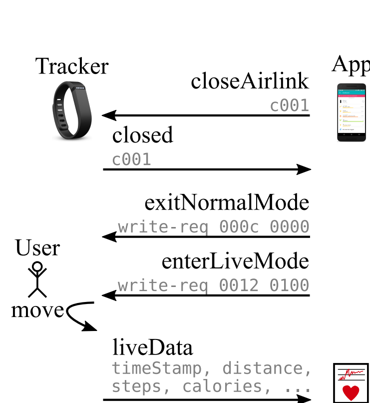


Fig. 5. Live mode via an authenticated local connection.

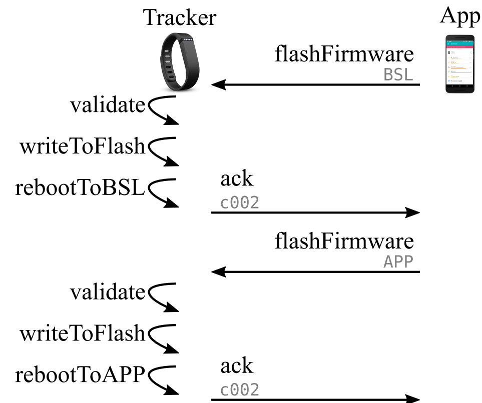


Fig. 6. Client-side firmware update process.

4.4 Live Mode

Displaying the step count, distance traveled, and heart rate in real-time would incur a high load on the Fitbit servers, under the current architecture. This is because megadumps would have to be transmitted to the Fitbit server, then decrypted and interpreted by the server backend. Subsequently, responses summarized in JSON format would be sent back to the user's smartphone app. By doing so, the user may always experience a few seconds lag, while multiple users could rapidly put an intensive load on the server infrastructure.

Live mode introduces a change in the tracker↔app communication paradigm. Specifically, after authentication, the tracker allows switching to unencrypted communication that takes place directly between the tracker and the smartphone app. This mode is supported for all the fitness tracker models, and we explicitly tested it with

Fitbit One, Flex, and Alta. We were able to reproduce the switch to live mode and understand the meaning and format of all protocol fields by disassembling the app and sniffing BLE packets.

In Figure 5 we illustrate how a live mode connection is established, noting that first the normal command mode must be left. Consequently, the tracker does not accept any further commands while in live mode operation. A live mode frame is generated roughly every second while the user is moving. Such frames contain the *current date, step count, calories, floors climbed, distance traveled, and optionally heart rate and heart rate confidence*. Older trackers do not check the origin of a BLE connection, but simply accept instructions to switch into live mode from any device, if there was previously a successful authentication [9].

On one hand, by triggering live mode, even the newest trackers can be used open-source without sharing any data to the Fitbit cloud. On the other hand, attackers can sniff unencrypted live mode data, and possessing a tracker's authentication credentials is enough to be able to activate live mode arbitrarily.

4.5 Firmware Update

Available firmware updates are shown in the smartphone app. The original smartphone app requires the user to initiate an update. User interaction ensures that the tracker stays in range and is charged during the process, since flashing firmware over BLE onto the tracker takes a few minutes. Firmware updates are requested by the tracker from the server with a microdump payload, such that only firmware for this specific tracker can be retrieved. A firmware update is encapsulated into a (typically encrypted) microdump response, which can be sent to the tracker directly after local pairing without authentication.

Initiating firmware updates does not require authentication or user interaction. Plaintext updates and downgrades can be retransmitted to any tracker of the same model that is also in plaintext mode.

Fitbit employs a two-stage update process shown in Figure 6, which we reverse engineered from the firmware itself utilizing methods explained in Section 6. In the first stage, the `BSL` is flashed, which contains enough functionality to enable the main application update procedure. In the second stage, the tracker reboots to the `BSL`, and flashes the `APP`, which provides the full functionality of the tracker.[2] The update process is finished by rebooting into the `APP`.

A firmware update must pass a number of validity and integrity checks before being written to the persistent flash memory. First, generating valid firmware is difficult, since a checksum is contained in an intermediate field inside the firmware, and prior to generating the checksum a bit in the firmware must be flipped. This basic check prevents one from simply exchanging strings in the sniffed firmware update. Second, firmware needs to be transferred in the firmware dump format (which was first observed in [27]), but with important information to reproduce firmware flashing missing. Notably, the first header contains the firmware length and destination memory address; the firmware segments start with the tracker's model number, continue with the action (`rebootToBSL`, `writeToFlash`, etc.) and end with segment length and checksum. Reboot actions have length and checksum set to zero, and they can be issued stand-alone.

Firmware updates have three fail-safe features. First, the separate update stages prevent from flashing non-functioning firmware into both sections—as long as one of them is still working, the broken one can be repaired. Second, each firmware update segment starts with the first byte of the serial number, which is the tracker model number, to prevent from flashing incompatible firmware, in case the smartphone app would mix up connected trackers. Last, the firmware checks if the address range of the new firmware is within flash. Fail-safe features

[2]The Fitbit internal naming scheme distinguishes between firmware `APP` and smartphone app, which interact with each other.

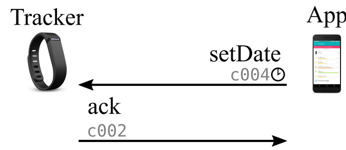


Fig. 7. Setting any date in the past or future.

can be circumvented by two rounds of flashing, where the first round is malware disabling them. BSL and APP separation can be removed to make more room for malicious functionality.

The BLE communication setup only allows for a 20 byte transmission length per frame, which are acknowledged by the tracker individually, making the firmware update process last several minutes. Faster flashing can be enabled by only flashing the APP, or by reducing firmware size at the cost of functionality.

> Attackers can wirelessly flash malware on trackers that are in plaintext mode, or that are vulnerable to authenticated memory readout.

4.6 Setting Arbitrary Dates

Older tracker models will respond to a wireless unauthenticated command to set any date, as shown in Figure 7. Dates can be both in the past or in the future and records stored in the device's EEPROM will contain this date.

> Setting the tracker's date can be used by attackers to selfishly manipulate activity timestamps or to trigger malfunction of nearby devices if time jumps arbitrarily.

4.7 Setting Alarms

Normally, alarms are set by the user via the Web API and then included in the next server's megadump response. The format of combined megadump alarms was first discovered and discussed on the Galileo mailing list, but the documentation has since gone offline [28]. Using megadump alarms requires the Fitbit server or broken encryption. We find that there is an authenticated stand-alone alarm dump format, which is plaintext even on encrypted trackers. It works similar to microdumps and megadumps, but with a different format type and dump start. Up to eight alarms can be set and stored in the EEPROM. This feature is included in our demo app as shown in Figure 8b.

> Trackers provide an authenticated command to set alarms without requiring valid encryption.

5 ANDROID APP REVERSE-ENGINEERING

In Section 5.1, we reverse engineer and modify the official Fitbit mobile app to gain insight into the underlying communication protocols as well as access to hidden developer settings. We use this knowledge, combined with further details from the protocol and firmware reverse-engineering effort to build an open demo app in Section 5.2, which can be employed to exchange data and firmware with trackers.

5.1 Modifying the Original App

We decompile the official Fitbit app using `apktool` [29], specifically targeting version 2.38. However, our approach is not limited to just this version—more recent versions, such as 2.60, can also be re-compiled as described below.

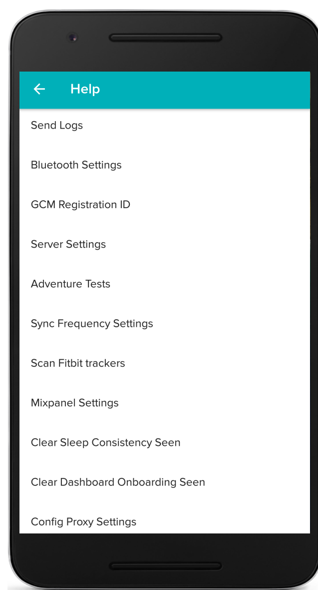
Decompiling with `apktool` outputs a Smali representation of the Dalvik bytecode. This representation is designed to be editable and to be human-readable, and as such gives a clear insight into the working of the app.

Taking the Smali representation, the `DEBUG` flag can be enabled as described in [27], revealing the contents of many interesting variables along with comprehensively logging the actions performed during runtime. These variables include authentication credentials, BLE parser information, and task state handles. Based on the debug log, not only can we identify interesting functions, but we can also add custom logging statements, which assist in the extraction of communication payloads. Therefore, we no longer require HTTPS and BLE sniffing to obtain access to various data. Our logging extensions always print the full authentication credentials whenever they are used, along with OAuth state information. Using this technique, we discovered all the necessary information required for switching to live mode, as well as a translation of the proprietary BLE protocol's error codes.

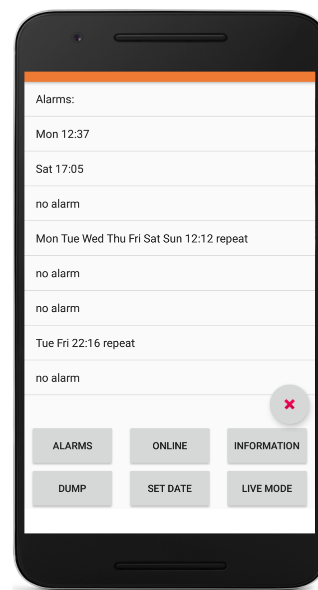
We make alterations to the communication payloads, in order to further understand and analyze the protocol internals. We accomplish this by exchanging function parameters with known static values, and observing the result. For example, the pairing procedure expects a BLE address, various data dumps, and an association secret. Exchanging static values has the advantage that one can still use the vendor app, and also the complete BLE stack. Thus, partial understanding and modification of the protocol is sufficient to launch attacks and confirm assumptions.

> App modifications allow for exchanging minor parts of the protocol and exploiting their vulnerabilities.

We find that there are hidden menu settings, which can be enabled by compiling the app in developer mode. This mode has a number of additional capabilities (than just debug mode), and serves to manually enable entries on the help page, as shown in Figure 8a. Hidden settings include valuable options for protocol reverse-engineering



(a) Hidden advanced app settings enabled by bytecode modifications.



(b) Independent custom implementation to set up to 8 alarms locally.

Fig. 8. Modified original app with developer options and custom app rewrite.

and developing cloud independence. For example, it is possible to configure an HTTPS proxy and change the remote server addresses, which could also be misused by attackers for MITM purposes. Further analysis shows that these settings can even be modified in a "vanilla" app locally in XML configuration files.

> The Fitbit smartphone app contains many developer and testing functions, including security-critical options, that are only disabled through easily accessible configuration files.

A custom server could be implemented as an alternative independent open-source solution to synchronize trackers, while still maintaining the look and feel of the original app. Ongoing developer community efforts coordinated via the Galileo mailing list focus on understanding plaintext server responses in detail [30]. This makes the scenario of building a custom server synchronizing devices more realistic, whilst also opening the possibility of developing firmware images that implement alternative, more robust, encryption.

> The smartphone app's configuration options allow for using an alternate ecosystem.

5.2 Building an Open-Source Demo App

Along with this article, we have released an open-source demo app, which implements a subset of the presented attacks, with the goal of having a more open fitness tracker interface [15]. With this, we aim to help the open source tracker community build better applications. Our app currently supports unauthenticated commands on any nearby tracker, as well as authenticated commands for trackers that a user is already associated with.

Unauthenticated commands are local pairing, reading general information about the tracker, setting and getting a tracker's date, retrieving microdumps and megadumps, and uploading dump responses generated by a server. Uploads can also be firmware updates–the app generates encrypted firmware updates based on binaries.

Authenticated commands can be performed after logging in and requesting reusable authentication credentials for an already associated tracker. We explicitly do not provide fake association capabilities to prevent abuse. The set of available authenticated commands depends on the tracker model. Live mode is supported by all of trackers that we analyzed, including the most recent Fitbit Alta and Ionic. Fitbit Flex and One also support setting alarms. A screenshot of the demo app is shown in Figure 8b, showing how to edit alarms. Interestingly, these can be enabled for multiple days in a row, once, daily, or perpetually using a special "repeat" flag. Until a firmware update in October 2017 [23], the Fitbit Flex and One were vulnerable to a memory readout attack, which can be leveraged to extract encryption keys. Since the firmware update does not change the encryption keys, modified firmware can be flashed over the air on trackers with known keys even after this recent update.

6 DISSECTING AND MODIFYING THE TRACKER'S FIRMWARE

While most of the Web API communication is officially documented by Fitbit, we were able to extract missing details from analysis of the smartphone app. The proprietary wireless tracker commands sent via BLE/GATT are undocumented. Reverse-engineering the tracker's firmware is required to fully understand command interpretation and dump generation. Firmware can be extracted from trackers themselves or from a sniffed firmware update [13, 27]. In this article, we go beyond firmware extraction, and we reverse-engineer and modify the firmware internals.

Static and dynamic analyses give further insights. We follow protocol parsing and error handling, and identify undocumented commands. We modify the firmware and enable support for GNU Debugger (GDB) to facilitate dynamic analysis. We successfully change security relevant functions inside the original firmware code, to disable authentication and encryption, and open up the possibility of crafting tracker malware.

6.1 Static Firmware Analysis

The firmware we dive into are the Flex versions 7.64, 7.81, and 7.88. These versions do not contain function names or strings in the main part, which complicates reverse-engineering. We use IDA Pro [31] for a static analysis of the firmware binaries.

Memory Layout. Knowing the main processing chip used (STM32L151UC based on an ARM Cortex M3 [13]) we can infer which purpose certain memory areas serve. This explains all addresses we show in the left part of Figure 9 [32]. A closer inspection reveals that a firmware update targets two memory areas, `BSL` and `APP`. The `start` area is never updated and contains code which either boots into `BSL` or `APP`. While `APP` contains all functions used during normal operation, `BSL` contains a reduced set of functions that are required for upgrading the `APP` code. We show the structure of this flash memory on the right-hand side of Figure 9.

The remaining two regions are the `EEPROM` and `SRAM`. Data with long-term persistence, such as the K_{dev}, the serial number, and encryption options are stored in the `EEPROM`. The `EEPROM` also contains data that should survive an empty battery, such as time zone settings, information about the user, and not yet synchronized fitness data. The short-term memory required for parsing and generating frames is effectively mapped to the `SRAM`.

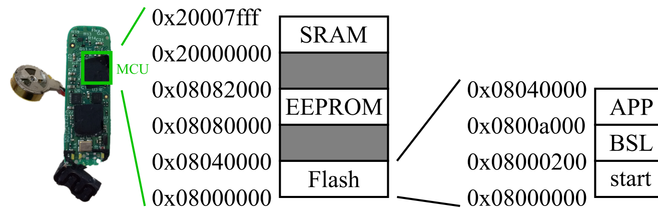


Fig. 9. Memory layout for a disassembled Fitbit Flex.

Bluetooth Chip. The BLE chip used in the Fitbit Flex is the nRF8001 from Nordic Semiconductor [33]. The product specification [34] states the one byte long commands that must be used to communicate with the chip. We use these commands to locate the functions responsible for the Fitbit's BLE communication. Further investigation reveals that the library in the firmware is almost identical to an open-source library used for an Arduino BLE Breakout Board [35]. Therefore, all BLE related function names can be carried over to the tracker's firmware. This helps us to identify the execution path that is used by incoming BLE frames. From there on, we can follow the frame processing through the firmware.

Command Line. We discover that Fitbit firmware exposes an internal command line interface, not secured by authentication credentials, and probably used in factory for testing and initialization of trackers. The commands available on this interface can be listed with a helper function, which is built around a list of function references and a string that describes their behavior. This convenient mapping gave us the ability to label many functions in the firmware.

> Trackers have an unsecured command line interface providing functionality testing, as well as setting the serial ID and encryption key.

Fitbit Frames and Commands. The firmware code contains a central switch-case statement, which determines whether to parse a command (starting with `c0`) or a chunk of data (everything else). Starting from this decision, we find the actual command and frame parsing structure, as well as the command response generation routine. After

each successful wireless command, an acknowledgement is generated by the firmware. If the tracker receives completely invalid commands, it will stop responding and require a BLE reconnect to restore its operation.

We extract a complete list of wireless commands, including date and alarm modifications.

Encryption. We identify XTEA by its characteristic 64-bit delta string. By comparing open source libraries to the decompiled code, we find that `LibTomCrypt` is used, and compare it to further functions, thereby finding that XTEA is used in EAX mode [36]. With the debugger enabled, we find all the required parameters. In contrast to first research on Fitbit decryption [37], we are able to generate valid encrypted messages containing a CMAC.

6.2 Modifying the Firmware

We use the Nexmon framework [38] to apply modifications to the firmware used by the ARM chip. Nexmon allows writing firmware patches in C. These patches can be compiled into a firmware binary which can then be flashed onto a tracker's microcontroller either via a ST-Link debugger or via BLE as explained in Section 4.5. We adapted the Nexmon framework to the memory layout in the Fitbit Flex. Since the `APP` and `BSL` memory regions are not completely used for code, we use the free areas to introduce the following changes in the firmware:

Disabling Authentication. In Section 4.3, a multi-step replay authentication was shown. Patching the firmware allows us to permanently disable authentication, which enables us to skip the authentication step when using, for example, live mode. We identify the function responsible for the authentication by the byte value of the error message issued to signal that authentication is required (that was previously extracted from the Android app). The function called before the error is raised returns `true`—if the authentication was successful. Therefore, we can disable authentication by always returning `true`, without performing the checks of the original function.

Disabling Encryption. The method we use to disable authentication can also be applied to encryption. A central function is used in the firmware to check if the communication is encrypted or not. Disabling encryption this way makes it impossible for the server to force a tracker to switch back to encryption mode. Hence, microdumps and megadumps will always be transmitted in plaintext. This poses a substantial risk of privacy leaks.

Firmware Versions. Code changes between the firmware versions are minimal, most functions are not modified. Reverse engineered functions from version 7.64 were easy to port to 7.81 and 7.88. As a proof of concept, we published a hook that multiplies the step count by 100 in the most recent firmware version 7.88 released in October 2017 [16].

Other than disabling authentication and encryption, we can also introduce new BLE commands to trigger certain events. One example is the configuration of the debug pins, which we describe in the following section.

6.3 Dynamic Analysis and GDB Support

We gain further insights by dynamically analyzing the firmware during its runtime. The ST-Link debugger can be used as an interface to a GDB server [39]. This allows setting breakpoints as well as memory watchpoints. We use the GDB server, which is included in the Atollic TrueSTUDIO [40].

Connecting the debugger with the original firmware is only possible directly after a hard reset of the ARM microcontroller. Continuing execution causes the debugger to disconnect, because of the general-purpose input/output (GPIO) port configuration after firmware initialization. Directly after reset, GPIO pins 13 and 14 in group `A` are assigned to `SWCLK` and `SWDIO` respectively, which enables debugging. When continuing execution, the Fitbit firmware reconfigures these pins to analog mode, thereby disabling debugging. In order to restore the debugging capability, we sought to reconfigure the pins to their original functionality. Finding and disabling the

responsible code in the Fitbit firmware can prove tedious. We used the Nexmon framework to add an additional function to the firmware, which is executed after initialization and resets the pins to their original purpose. The appropriate addresses that must be used (i.e. for `GPIOA->MODER`) are specified in the reference manual [41].

The reconfiguration of the GPIO pins, however, comes with certain side effects, as the pins were intended to be used for a different purpose during runtime of the Fitbit code. To minimize the risk of side effects, we implemented an additional BLE command that triggers the reconfiguration of the GPIO pins. Being able to debug the tracker during runtime enables us to verify assumptions made during the static analysis.

7 DESIGN RECOMMENDATIONS

In this section we give a series of recommendations that may be used to mitigate the vulnerabilities we have identified and documented in this article. Accordingly, Fitbit's ecosystem can be generalized to other IoT platforms, as many of these systems use pre-configured hardware and a smartphone app to communicate with the cloud. We hope that our recommendations not only help to further strengthen the security of the Fitbit ecosystem, but also provide insights to other vendors who choose to develop secure IoT systems from scratch.

7.1 Security by Design

Based on the findings presented in the previous sections, we conclude that the original Fitbit architecture was not designed with security in mind from the start. The manufacturer's initial attack modeling does not appear to have considered the fact that users may have incentives to spoof fitness data, or to perform espionage within proximity. Instead, it was assumed that users (or more specifically tracker owners) care only about tracking their own activity and fitness, and hence storing only small amounts of data on the tracker or in the app would provide sufficient safeguards. Security was added while products were already shipped to end users, resulting in trackers that are potentially still compromised. For example, if the encryption keys to a particular tracker were leaked with an old firmware version, valid firmware can still be generated and transferred to such trackers over the air, putting second hand devices at risk. New encryption keys preventing unwanted firmware modifications could only be rolled out via the tracker's insecure BLE connection.

We recognize that many of the weaknesses identified could have been circumvented if a "security by design" approach was adopted from the first steps of developing fitness tracker prototypes. Even a robust patching mechanism is compromised by the user not having enough knowledge or patience to correctly follow the upgrade procedures. Moreover, given that the hardware was produced with cost constraints in mind, wearable devices may not have sufficient memory to accommodate the extra code necessary for strengthening their security.

7.2 Secure Device Association

Recall that there are two distinct pairing procedures: (1) local pairing, and (2) remote association. The proprietary local pairing mechanism has no security mechanisms, and although the majority of features necessary to perform secure local pairing are present, they are only enforced at the later remote association to user accounts.

One way to mitigate this problem is to enforce tracker-initiated pairing, and to refuse pairing without user↔tracker interaction. On trackers with a display, the proprietary handshake can be replaced with the standardized BLE pairing method. Trackers without a display should require physical tapping, and should present pairing codes as LED patterns.

During remote association, the server believes that the tracker is honest, and does not verify data freshness. Hence fake-association attacks become possible. This problem can be overcome by implementing an additional step in the pairing process, depicted in Figure 3b—the already transmitted `pairingToken` should not only be forwarded to the app, but also to the tracker, and the tracker's next response must depend on this token. The server should check if the `pairingToken` transmitted initially is present in the dump replied by the tracker at

the second association step. Adding this security feature does not require any Web API changes, but simply a new command in the smartphone app (for forwarding the token), and an update to the command parser in the tracker's firmware.

7.3 Serial Number Protection

Under certain conditions, only the tracker's serial number is required for association (see Section 4.2). Therefore, this should be replaced with an alias on the packaging, so as not to tamper with vendor inventory stock-keeping. As the serial number is also present in the non-encrypted header portion of megadumps (to facilitate payload decryption at the server), this allows spies to trace trackers.

To mitigate this risk, we suggest employing an asymmetric encryption scheme. Trackers would be pre-loaded with the server's public key, and use this to encrypt the entire megadump (including the serial number). Upon reception, the server would use its private key to decrypt the message, and then use the decrypted serial number to lookup the tracker's public key as before, allowing an encrypted response to be sent. We recognize two challenges facing this approach: (1) if the server's private key were to be compromised, the public key would have to be revoked and re-programmed into all trackers (one way to deal with this is by implementing an OpenPGP (RFC 4880) variant); and (2) asymmetric encryption requires more complicated calculations, thus longer message parsing times and higher energy consumption – the impact should, however, be minimal, given the relatively infrequent megadump transmissions.

7.4 Stricter Encryption

Trackers and servers currently assume that each other is trustworthy, and recent models uphold this trust by employing end-to-end encryption. Unfortunately, this assumption breaks when tampering with tracker firmware, or whilst under MITM attacks. Attackers can also issue any local BLE command to victim trackers and replay the authentication sequence. A solution to this is stricter usage of the pre-installed encryption key. Any BLE command should be issued by the server and encrypted. Strict encryption would increase security tremendously, though at the cost of harder debugging during development.

Although enabling live mode, and issuing certain commands, requires local authentication to take place, that authentication is vulnerable to replay attacks, and the following authenticated communication channel (e.g. the live mode data stream) is in plaintext. We propose instead to use end-to-end encrypted mutual authentication between the server and trackers, then utilise session keys to encrypt the communication between the tracker and the app. After this, the live mode data stream could use a session key to encrypt and transfer heart rate and step count without delay, securing the channel from attackers, and preventing them from obtaining persistent authentication tokens.

7.5 Android App and Usable Security

Attackers can tamper relatively easily with the official Android smartphone app. While users may also wish to purposely modify the app in order to free their data, they should be notified if a detectable attack is performed. To this end, we recommend three changes, the first two also being applicable to specific iOS versions:

(1) Vendors should consider the environment in which their app is used as hostile, which also means the list of certification authoritys (CAs) held by the operating system can not be trusted. Certificate pinning and HTTP Strict Transport Security (HSTS) may be obvious solutions, though it should also be possible to authenticate client and server entities without a CA. IETF efforts to specify the DNS-Based Authentication of Named Entities (DANE) protocol (RFC 7218) make an important step towards this goal.

(2) Users should be made aware, e.g. through a notification icon in the app, if their tracker could be transmitting health data in plaintext to anyone in wireless range. Encrypted dumps have a slightly different header as

compared to unencrypted ones, thus the app can already check locally for encryption. Moreover, encrypted dump headers have a counter field that increases with each synchronization. This can be used to detect if another entity (e.g. a spy) has requested dumps.

(3) As Android supports build flavors, developer options should be removed from any firmware or application shipped. This does not make modifications impossible, but considerably harder.

7.6 Firmware and Microcontroller

Firmware updates are secured by checksums and optional encryption on newer tracker models. Checksums themselves are just a form of payload integrity protection, and as such can be calculated by anyone. The symmetric key used for the firmware integrity check can be obtained via the aforementioned memory readout vulnerability on older tracker models or via physical access on newer models. Once the symmetric key is obtained, any modified firmware can be encrypted and will be accepted by the tracker, irrespective of the algorithm used (XTEA or AES in EAX mode). Using a symmetric key makes it necessary to store the key on the tracker itself. A more robust method of firmware verification would be to sign updates using a public and private key pair, which can be rolled out at manufacturing. Using asymmetric cryptography for firmware signing eliminates the possibility of an attacker generating valid firmware updates by accessing a key in the tracker's memory.

The STM32 microcontroller used in the trackers we have studied has ARM security extensions disabled, even though they are fully supported by the platform. The memory protection unit (MPU) could be enabled and employed to guard regions of memory from buffer overflows. Additionally, the `XN` "Execute Never" bit should be set for the RAM memory region. This would help in preventing stack overflows from directly executing code that may have been crafted in RAM. In addition to this, memory protection should be activated, which prevents reading and writing memory. More generally, debugging facilities should be disabled.

8 RELATED WORK

We confine consideration to related work that delves deeper into the Fitbit ecosystem and omit research on fitness trackers and and IoT ecosystems in general. To the best of our knowledge, there are no scientific analysis of big real-world IoT ecosystems including all their heterogeneous components. Fitbit puts many security measures in place where others employ no security at all, as a comprehensive study looking into 17 fitness trackers from various vendors except Fitbit shows [22].

Cyr *et al.* were the first to research Fitbit security mechanisms by reverse-engineering the smartphone app [42]. They found that cryptographic mechanisms are indeed used to secure the communication between tracker and server, but did not reveal in-depth understanding of these mechanisms and whether they may be compromised. The closest work to our contributions is by Schellevis *et al.*, who used tracker↔app authentication to extract memory at chosen addresses and undertook app and firmware analysis to understand in detail the encryption mechanisms implemented [27]. The memory readout vulnerability was found by comparing Fitbit Charge firmware updates, though this is fixed in the current firmware version. However, we still encountered this issue in the most current Fitbit One and Flex firmware. Despite understanding the basics of the firmware update, important gaps exist in their findings, which would need to be covered in order to generate custom updates. Schellevis recently extended the open-source Galileo software with authentication and megadump XTEA decryption features [37], but the correct EAX CMAC mode are missing. A different firmware analysis also found the test command menu and integrated it in a Unicorn/Radare2-based emulation, which is an interesting tool chain for further investigations, but did not yet continue with the analysis [43].

On the Galileo mailing list it was claimed that trackers in encrypted mode could be switched back to plaintext by sending a valid plaintext server response [44]. We tried this for the same tracker model, Flex, with both firmware versions, but it did not work—the encryption options are not only "enable encryption" and "disable

encryption", but also "trial encryption" when the tracker tries to initiate an end-to-end encrypted communication with the server and still switches back to plaintext if the server response is plaintext. The observed attack works for trackers in "trial encryption", but not for trackers in the wild with persistently enabled encryption.

Fereidooni *et al.* modified the behavior of Fitbit Flex through hardware exploitation and memory contents manipulation in regions that cannot be flashed remotely [13]. How to read and flash all memory regions was not understood up to now, and this is important for a comprehensive understanding of the firmware code and its vulnerabilities. Our work is the first to modify the firmware itself, and perform wireless modifications.

AV-Test discovered that with older Fitbit Charge firmware versions, live mode can be accessed by any device nearby after a recent authentication from a smartphone, obfuscating their finding by only listing the associated UUID [9]. Goyal *et al.* revealed that a denial of service attack on a selected tracker is possible via BLE [12]. Galileo recently implemented an experimental BLE synchronization, supporting local pairing and dump retrieval. Our commands listed in Table 1 are not yet known by the community.

In [45], Apvrille summarizes everything recently found about the Fitbit protocols. Note that most of the commands documented are already implemented in the open Galileo tools or in her own `fittools`. Apvrille proposed to use random numbers generated by the tracker for the authentication challenges and published `fittools` implement this feature [46], certainly authentication challenges are not random but cryptographically generated from partially predictable parameters and should not be used for such purposes. Moreover, Apvrille found a simple 20 byte echo command and claimed that this could already be an attack vector for firmware modifications—however, Fitbit stated this is not possible with this command which we confirm with our firmware reverse-engineering [47].

Earlier work focused on the security of old Fitbit Ultra models, which use an ANT protocol stack instead of BLE, and thus are not supported by the smartphone app. Instead, synchronization requires a Windows tool that was last updated in 2012 [48]. The Ultra security model is very weak, as demonstrated by Rahman *et al.*, who documented the security and privacy issues encountered and proposed a FitLock tool to add authentication and encryption [11]. Zhou *et al.* later highlighted security issues pertaining to the FitLock [49], which prompted Rahman *et al.* to revisit Fitbit Ultra security and propose SensCrypt as a remedy [50]. The Fitbit Ultra system is now only supported by the manufacturer for legacy reasons and the current Fitbit infrastructure already employs encryption and authentication.

9 CONCLUSIONS

Wearable technology has become extremely popular and users are entrusting manufacturers with handling increasingly more of their personal data, while having less control over it along the way. These companies must take responsibility for safeguarding such private information, and it is clear that more needs to be done across the board to tackle privacy and security concerns.

In this paper we have demonstrated that by combining a repertoire of investigative techniques, such as protocol analysis, software (de)compiling, and static and dynamic embedded code analysis, we were capable of reverse-engineering the Fitbit's communication protocol to forcibly associate victim trackers to an attacker controlled user account and enable unencrypted live streaming of fitness data, thereby leaking private information. We have gained in-depth understanding of the tracker firmware, showed it is possible to craft new firmware with malicious intent, and successfully deploy this over-the-air without user interaction. This allowed us to disable encryption and reveal another potential privacy breach. We further showed that recompiling a modified version of the official Fitbit smartphone app enabled developer options not normally accessible, and in combination with customized firmware this is a major step towards building an alternative custom "Fitbit cloud", with full control over personal data and potentially with better security provisions. To demonstrate the attacks are reproducible

by end-users despite the complexity of the steps involved, we released a smartphone demo app and firmware modification framework showing how to exploit the protocol and firmware [15, 16].

Lastly, we suggest that the recommendations identified in Section 7 should be seriously considered when implementing new, or upgrading existing IoT systems. Privacy should be a central concern for any organization handling sensitive and personal data, but so should be the right for a user to handle their own data. With such technology gaining in popularity, increasing in complexity, exploding with more features, and collecting more and more data, companies must be vigilant and not let privacy concerns take a back seat.

ACKNOWLEDGMENTS

This work has been co-funded by the DFG as part of projects S1 within the CRC 1119 CROSSING and C.1 within the RTG 2050 ”Privacy and Trust for Mobile Users”, and by the BMBF within CRISP. Paul Patras has been partially supported by the Scottish Informatics and Computer Science Alliance (SICSA) through a PECE grant.

We thank the Fitbit security team for their professional collaboration and acknowledging our work along with their security fixes. Moreover, we thank Max Maass for proof-reading and discussing our work, as well as Sven Ströher and Matthias Hanreich for helping to develop our demonstration app.

REFERENCES

[1] Statista. Statistics & facts on wearable technology, November 2017.
[2] PwC. The Wearable Life 2.0. https://www.pwc.com/ee/et/publications/pub/pwc-cis-wearables.pdf, 2016.
[3] Jessica Twentyman. Wearable devices aim to reduce workplace accidents. *Financial Times*, June 2016.
[4] Jamiles Lartey. Man suspected in wife’s murder after her Fitbit data doesn’t match his alibi. *The Guardian*, April 2017.
[5] IDC. Worldwide quarterly wearable device tracker, August 2017.
[6] Fitbit. Q3’17 earning summary, November 2017.
[7] Pern Hui Chia, Yusuke Yamamoto, and N. Asokan. Is This App Safe? A Large Scale Study on Application Permissions and Risk Signals. In *Proceedings of the 21st International Conference on World Wide Web*, pages 311–320, Lyon, France, 2012. ACM.
[8] M. Conti, N. Dragoni, and V. Lesyk. A Survey of Man In The Middle Attacks. *IEEE Communications Surveys Tutorials*, 18(3), 2016.
[9] Eric Clausing, Michael Schiefer, and Maik Morgenstern. AV-TEST Analysis of Fitbit Vulnerabilities. https://www.av-test.org/fileadmin/pdf/avtest_2016-04_fitbit_vulnerabilities.pdf, 2016.
[10] A. Hilts, C. Parsons, and J. Knockel. Every Step You Fake: A Comparative Analysis of Fitness Tracker Privacy and Security. 2016.
[11] Mahmudur Rahman, Bogdan Carbunar, and Madhusudan Banik. Fit and Vulnerable: Attacks and Defenses for a Health Monitoring Device. In *Proceedings of the 13th Privacy Enhancing Technologies Symposium (PETS)*, Bloomington, Indiana, USA, July 2013.
[12] 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*, pages 131–136, Pisa, Italy, 2016.
[13] H. Fereidooni, J. Classen, T. Spink, P. Patras, M. Miettinen, A.-R. Sadeghi, M. Hollick, and M. Conti. Breaking Fitness Records without Moving: Reverse Engineering and Spoofing Fitbit. In *Research in Attacks, Intrusions and Defenses (RAID)*, September 2017.
[14] Jakob Rieck. Attacks on Fitness Trackers Revisited: A Case-Study of Unfit Firmware Security. *CoRR*, abs/1604.03313, 2016.
[15] Open Source Fitbit Fitness App. https://github.com/seemoo-lab/fitness-app, 2018.
[16] Open Source Fitbit Fitness Firmware Modifications. https://github.com/seemoo-lab/fitness-firmware, 2018.
[17] Christian Holz and Edward Wang. Glabella: Continuously sensing blood pressure behavior using an unobtrusive wearable device. *Proc. ACM Interactive, Mobile, Wearable and Ubiquitous Technologies (IMWUT)*, 1(3), Sept 2017.
[18] Weixi Gu, Yuxun Zhou, Zimu Zhou, Xi Liu, Han Zou, Pei Zhang, Costas J. Spanos, and Lin Zhang. Sugarmate: Non-intrusive blood glucose monitoring with smartphones. *Proc. ACM Interactive, Mobile, Wearable and Ubiquitous Technologies (IMWUT)*, 1(3), Sept 2017.
[19] K. Mandula, R. Parupalli, C. A. S. Murty, E. Magesh, and R. Lunagariya. Mobile based home automation using internet of things(IoT). In *International Conference on Control, Instrumentation, Communication and Computational Technologies (ICCICCT)*, Dec 2015.
[20] Philipp Morgner, Stephan Mattejat, and Zinaida Benenson. All your bulbs are belong to us: Investigating the current state of security in connected lighting systems. *CoRR*, abs/1608.03732, 2016.
[21] Roman Zaikin, Dikla Barda, and Oded Vanunu. HomeHack: How Hackers Could Have Taken Control of LG’s IoT Home Appliances. Available at: https://blog.checkpoint.com/2017/10/26/homehack-how-hackers-could-have-taken-control-of-lgs-iot-home-appliances/, October 2017.
[22] H. Fereidooni, T. Frassetto, M. Miettinen, A.-R. Sadeghi, and M. Conti. Fitness Trackers: Fit for Health but Unfit for Security and Privacy. In *IEEE International Workshop on Safe, Energy-Aware, & Reliable Connected Health (CHASE workshop: SEARCH)*, 2017.

[23] What's changed in the latest Fitbit device update? https://help.fitbit.com/articles/en_US/Help_article/1372, 2018.
[24] Galileo. https://bitbucket.org/benallard/galileo/, 2017.
[25] Fitbit Sensors API. https://dev.fitbit.com/reference/device-api/sensors/, 2017.
[26] gatttool. https://github.com/pauloborges/bluez/blob/master/attrib/gatttool.c, 2017.
[27] Maarten Schellevis, Bart Jacobs, Carlo Meijer, and Joeri de Ruiter. Getting access to your own Fitbit data. 2016.
[28] Dany. Fitbit Flex MegaDump and ServerResponse Data Format Description. Available at: https://www.freelists.org/post/galileo/Fitbit-Flex-MegaDump-and-ServerResponse-Data-Format-Description, May 2017.
[29] Apktool—A tool for reverse engineering 3rd party, closed, binary Android apps. https://ibotpeaches.github.io/Apktool/, 2017.
[30] Dany. Fitbit Flex MegaDump and ServerResponse Data Format Description. Galileo Mailing List https://www.freelists.org/post/galileo/Fitbit-Flex-MegaDump-and-ServerResponse-Data-Format-Description, 2017.
[31] Hex-Rays IDA Pro. https://www.hex-rays.com/, 2017.
[32] STM32L141UC datasheet. www.st.com/resource/en/datasheet/stm32l151cc.pdf, 2017.
[33] Fitbit Flex teardown from ifixit.com. https://www.ifixit.com/Teardown/Fitbit+Flex+Teardown/16050, 2017.
[34] nRF8001 Bluetooth Chip Product Specification. http://www.nordicsemi.com/eng/nordic/download_resource/17534/16/6078997/2981.
[35] Adafruit nRF8001. https://github.com/adafruit/Adafruit_nRF8001/tree/master/utility, 2017.
[36] LibTomCrypt. https://github.com/libtom/libtomcrypt, 2018.
[37] Maarten Schellevis. Maarten committed 3601372: Add a utility to decrypt older dumps. https://bitbucket.org/benallard/galileo/commits/3601372658e5e6da271300656d4ec503c5c87ddc, 2017.
[38] Matthias Schulz, Daniel Wegemer, and Matthias Hollick. Nexmon: The C-based Firmware Patching Framework. https://nexmon.org, 2017.
[39] GDB: The GNU Project Debugger. https://www.gnu.org/software/gdb/, 2017.
[40] Atollic TrueSTUDIO. https://atollic.com/truestudio/, 2017.
[41] STM32L141UC reference manual. www.st.com/resource/en/reference_manual/cd00240193.pdf, 2017.
[42] Britt Cyr, Webb Horn, Daniela Miao, and Michael Specter. Security Analysis of Wearable Fitness Devices (Fitbit). *Massachusets Institute of Technology*, 2014.
[43] Hugo Reinaldo. Hello Quark! Fitbit firmware reversing (Lessons learned). *AlligatorCon*, 2016.
[44] Dany. Fitbit Flex: switching between encrypted and unencrypted mode. Available at: https://www.freelists.org/post/galileo/Fitbit-Flex-switching-between-encrypted-and-unencrypted-mode, May 2017.
[45] Axelle Apvrille. Research on Fitbit Flex. Available at: http://www.fortiguard.com/events/1869/research-on-fitbit-flex, March 2017.
[46] Axelle Apvrille. Fitness Tracker: Hack In Progress. Available at: https://hackinparis.com/data/slides/2015/axelle_aprville_hackinparis.pdf, June 2015.
[47] Forbes. Fitbit Disputes Claim Fitbit Trackers Can Be Hacked And Infect PCs, October 2015.
[48] Fitbit Ultra Setup. https://www.fitbit.com/de/setup/ultra, 2017.
[49] Wei Zhou and Selwyn Piramuthu. Security/privacy of wearable fitness tracking IoT devices. In *Proceedings of the 9th Iberian Conference on Information Systems and Technologies (CISTI)*, pages 1–5, Barcelona, Spain, 2014. IEEE.
[50] Mahmudur Rahman, Bogdan Carbunar, and Umut Topkara. Secure Management of Low Power Fitness Trackers. *IEEE Transactions on Mobile Computing*, 15(2):447–459, 2016.

Received November 2017