Concise Paper: SensCrypt: A Secure Protocol for Managing Low Power Fitness Trackers

Mahmudur Rahman
Florida International University
Miami, FL
Email: mrahm004@cs.fiu.edu

Bogdan Carbunar
Florida International University
Miami, FL
Email: carbunar@cs.fiu.edu

Umut Topkara
IBM Research
Yorktown Heights, NY
Email: umut@us.ibm.com

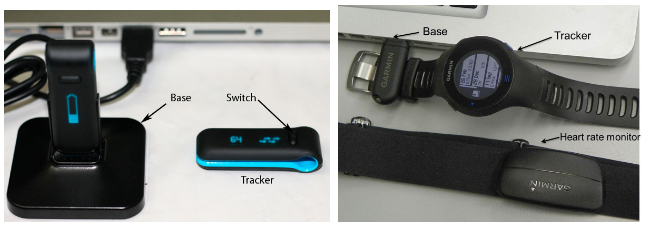
***Abstract*—The increasing interest in personal telemetry has induced a popularity surge for wearable personal fitness trackers. Such trackers automatically collect sensor data about the user throughout the day, and integrate it into social network accounts. Solution providers have to strike a balance between many constraints, leading to a design process that often puts security in the back seat. Case in point, we reverse engineered and identified security vulnerabilities in Fitbit Ultra and Gammon Forerunner 610, two popular and representative fitness tracker products. We introduce FitBite and GarMax, tools to launch efficient attacks against Fitbit and Garmin.**

We devise SensCrypt, a protocol for secure data storage and communication, for use by makers of affordable and lightweight personal trackers. SensCrypt thwarts not only the attacks we introduced, but also defends against powerful JTAG Read attacks. We have built Sens.io, an Arduino Uno based tracker platform, of similar capabilities but at a fraction of the cost of current solutions. On Sens.io, SensCrypt imposes a negligible write overhead and significantly reduces the end-to-end sync overhead of Fitbit and Garmin.

I. Introduction

Wearable personal trackers that collect sensor data about the wearer, have long been used for patient monitoring in healthcare. Holter Monitors [1], with large and heavy enclosures, that use tapes for recording, have recently evolved into affordable personal fitness trackers (e.g., [2]). Recently, popular health centric *social sensor networks* have emerged. Products like Fitbit [3], Garmin Forerunner[4] and Jawbone Up [5] require users to carry wireless trackers that continuously record a wide range of fitness and health parameters (e.g., steps count, heart rate, sleep conditions), tagged with temporal and spatial coordinates. Trackers report recorded data to a providing server, through a specialized wireless base, that connects to the user's personal computer (see Figures 1(a) and 1(b)). The services that support these trackers enable users to analyze their fitness trends with maps and charts, and share them with friends in their social networks.

All happening too quickly both for vendors and users alike, this data-centric lifestyle, popularly referred to as the Quantified Self or "lifelogging" is now producing massive amounts of intimate personal data. For instance, BodyMedia [6] has created one of the world's largest libraries of raw and real-world human sensor data, with 500 trillion data points [7]. This data is becoming the source of privacy and security concerns:



(a) (b)

Fig. 1. System components: (a) Fitbit: trackers (one cradled on the base), the base (arrow indicated), and a user laptop. The arrow pointing to the tracker shows the switch button, allowing the user to display various fitness data. (b) Garmin: trackers (the watch), the base(arrow indicated), and a user laptop.

information about locations and times of user fitness activities can be used to infer surprising information, including the times when the user is not at home [8], and company organizational profiles [9].

We demonstrate vulnerabilities in the storage and transmission of personal fitness data in popular trackers from Fitbit [3] and Garmin [4]. Vulnerabilities have been identified for similar systems, including pacemakers (e.g., Halperin et al. [10], Rasmussen et al. [11]) and glucose monitoring and insulin delivery systems (e.g., Li et. al. [12]). The differences in the system architecture and communication model of social sensor networks enable us to identify and exploit different vulnerabilities.

We have built two attack tools, FitBite and GarMax, and show how they inspect and inject data into nearby Fitbit Ultra and Garmin Forerunner trackers. The attacks are fast, thus practical even during brief encounters. We believe that, the vulnerabilities that we identified in the security of Fitbit and Garmin are due to the many constraints faced by solution providers, including time to release, cost of hardware, battery life, features, mobility, usability, and utility to end user. Unfortunately, such a constrained design process often puts security in the back seat.

To help address these constraints, in this paper we introduce SensCrypt, a protocol for secure fitness data storage and transmission on lightweight personal trackers. We leverage the unique system model of social sensor networks to encode data stored on trackers using two pseudo-random values, one generated on the tracker and one on the providing server. This enables SensCrypt, unlike previous work [10], [11], to protect not only against inspect and inject attacks, but also against attackers that physically capture and read the memory of trackers. SensCrypt's hardware and computation requirements are minimal, just enough to perform low-cost symmetric key encryption and cryptographic hashes. SensCrypt does not impose storage overhead on trackers and ensures an even wear of the tracker storage, extending the life of flash memories with limited program/erase cycles.

SensCrypt is applicable to a range of sensor based platforms, that includes a large number of popular fitness [3], [4], [5], [13], [14] and home monitoring solutions [15], [16], [17], as well as scenarios where the sensors need to be immobile and operable without network connectivity (e.g., infrastructure, traffic, building and campus monitoring solutions). In the latter case, the bases through which the sensors sync with the webserver are mobile, e.g., smartphones of workers, who may become proximal to the sensors with the intention of data collection or as a byproduct of routine operations.

We have developed Sens.io, a \$52 tracker platform built on Arduino Uno, of similar capabilities with current solutions. On Sens.io, SensCrypt (i) imposes a 6ms overhead on tracker writes, (ii) reduces the end-to-end overhead of data uploads to 50% of that of Fitbit, and (iii) enables a server to support large volumes of tracker communications.

While our defenses may not be immediately adopted by existing products [1], this paper provides a foundation upon which to create, implement and test new defensive mechanisms for future tracker designs.

II. System Model, Attacker and Background

A. System Model

We center our model on Fitbit Ultra [3] and Garmin Forerunner [4], two popular health centric social sensor networks (see Figures 1(a) and 1(b)). For simplicity, we will use "Fitbit" to refer the Fitbit Ultra and "Garmin" to denote the Garmin Forerunner 610 solution. The system consists of user tracker devices, USB base stations and an online social network. We now detail each system component.

The tracker. The *tracker* is a wearable device that records, stores and reports a variety of user fitness related metrics. The Fitbit tracker measures the daily steps taken, distance traveled, floors climbed, calories burned, the duration and intensity of the user exercise, and sleep patterns. The Garmin tracker records data at user set periodic intervals (1-9 seconds). The data includes a timestamp, exercise type, average speed, distance traveled, altitude, start and end position, heart rate and calories burned during the past interval. The tracker has a heart rate monitor (optional) and a 12 channel GPS receiver, enabling the user to tag activities with spatial coordinates. Both Fitbit and Garmin trackers have chips supporting the ANT protocol, with a 15ft transmission range for Fitbit and 33ft for Garmin. Each tracker has a unique id.

The base and agent module. The base connects with the user's main compute center (e.g., PC, laptop) and with trackers within transmission range (15ft for Fitbit and 33ft for Forerunner) over the ANT protocol. The user needs to install an "agent module", a software provided by the service provider (Fitbit, Garmin) to run on the base. The agent and base act as a bridge between trackers and the online social network. They upload information stored on the trackers to their users accounts on the webserver, see Figures 1(a) and 1(b) for system snapshots.

The webserver. The online social network webserver (e.g., fitbit.com, connect.garmin.com), allows users to create accounts from which they befriend and maintain contact with other users. Upon purchase of a tracker and base, the user binds the tracker to her social network account. In the following, we use the term *webserver* to denote the computing resources of the social network.

B. Attacker Model

We assume that the webserver is honest, and is trusted by all participants. We assume adversaries that are able to launch the following types of attacks:

Inspect attacks. The adversary listens on the communications of trackers, bases and the webserver.

Inject attacks. The adversary exploits solution vulnerabilities to modify and inject messages into the system, as well as to jam existing communications.

Capture attacks. The adversary is able to acquire trackers or bases of victims. The adversary can subject the captured hardware to a variety of other attacks (e.g., Inspect and Inject) but cannot access the memory of the hardware. We assume that in addition to captured devices, the adversary can control any number of trackers and bases (e.g., by purchasing them).

JTAG attacks. JTAG and boundary scan based attacks (e.g., [18]), extend the Capture attack with the ability to access the memory of captured devices. We focus here on "JTAG-Read" (JTAG-R) attacks, where the attacker reads the content of the *entire* tracker memory.

C. Crypto Tools

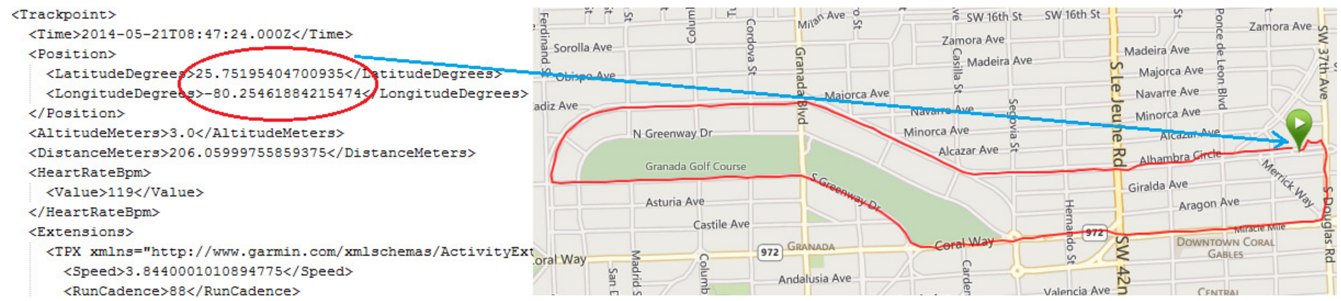
We use a symmetric key encryption system. We write $E_K(M)$ to denote the encryption of a message M with key K. We also use cryptographic hashes that are pre-image, second pre-image and collision resistant. We use $H(M)$ to denote the hash of message M. We also use hash based message authentication codes [19]: we write $Hmac(K, M)$ to denote the authentication code of message M with key K.

III. Fitbit and Garmin Attacks

A. Reverse Engineering Fitbit and Garmin

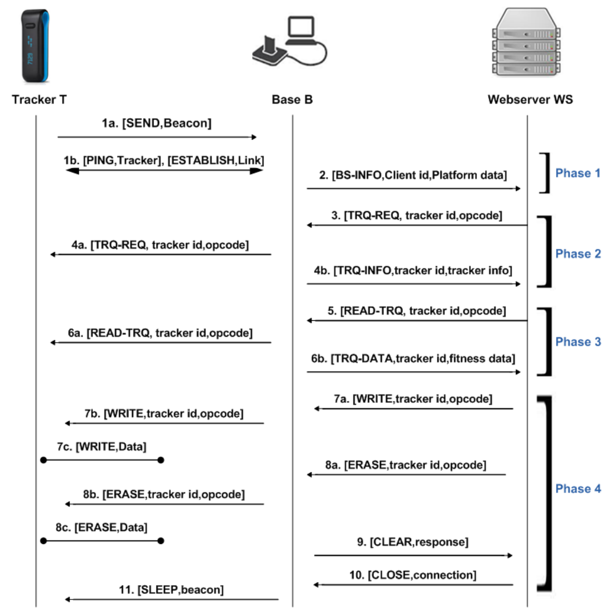
To log communications between trackers and webservers, we wrote USB based filter drivers and ran them on a base. We have used Wireshark to capture all Wi-Fi traffic between the agent software and the webserver. To reverse engineer Fitbit, we exploited (i) the lack of encryption in all its communications and (ii) libfitbit [20], a library built on ANT-FS [21] for accessing and transferring data from Fitbit trackers. Unlike Fitbit, Garmin uses HTTPS with TLS v1.1 to send

[1]We have contacted Fitbit and Garmin with our results. While interested in the security of their users, they have declined collaboration.



```
<Trackpoint>
  <Time>2014-05-21T08:47:24.000Z</Time>
  <Position>
    <LatitudeDegrees>25.75195404700935</LatitudeDegrees>
    <LongitudeDegrees>-80.25461884215474</LongitudeDegrees>
  </Position>
  <AltitudeMeters>3.0</AltitudeMeters>
  <DistanceMeters>206.05999755859375</DistanceMeters>
  <HeartRateBpm>
    <Value>119</Value>
  </HeartRateBpm>
  <Extensions>
    <TPX xmlns="http://www.garmin.com/xmlschemas/ActivityExt
      <Speed>3.8440001010894775</Speed>
      <RunCadence>88</RunCadence>
```

Fig. 3. TPDC outcome on Garmin: the attacker retrieves the user's exercise circuit on a map (shown in red on the right side), based on individual fitness data records (shown on the left in XML format). The data record on the left includes both GPS coordinates, heart rate, speed and cadence.



Tracker T
Base B
Webserver WS
1a. [SEND,Beacon]
1b. [PING,Tracker], [ESTABLISH,Link]
2. [BS-INFO,Client id,Platform data]
Phase 1
3. [TRQ-REQ, tracker id,opcode]
4a. [TRQ-REQ, tracker id,opcode]
4b. [TRQ-INFO,tracker id,tracker info]
Phase 2
5. [READ-TRQ, tracker id,opcode]
6a. [READ-TRQ, tracker id,opcode]
6b. [TRQ-DATA,tracker id,fitness data]
Phase 3
7a. [WRITE,tracker id,opcode]
7b. [WRITE,tracker id,opcode]
7c. [WRITE,Data]
8a. [ERASE,tracker id,opcode]
8b. [ERASE,tracker id,opcode]
8c. [ERASE,Data]
Phase 4
9. [CLEAR,response]
10. [CLOSE,connection]
11. [SLEEP,beacon]

Fig. 2. Fitbit *Upload* protocol. Enables the tracker to upload its collected sensor data to the user's social networking account on the webserver. SensCrypt's *Upload* protocol extends this protocol, see Section IV.

user login credentials. However, similar to Fitbit, all other communications are sent over plaintext HTTP.

Due to space limitations, we now focus on Fitbit's data upload protocol (see Figure 2). The communication between the webserver and the tracker through the base contains opcodes, commands for the tracker, e.g., TRQ-REQ, READ-TRQ, WRITE, ERASE, CLEAR. In a nutshell, the protocol works as follows. Upon receiving a beacon from the tracker, the base establishes a connection with the tracker. In Phase 1, the base contacts the webserver and sends basic client and platform information. In Phase 2, the webserver sends the tracker id and the opcode for retrieving tracker information (TRQ-REQ). The base contacts the tracker, retrieves its information TRQ-INFO (serial number, firmware version, etc.) and sends it to the webserver. In Phase 3, the webserver retrieves the associated tracker and user ids, then sends them to the base along with the opcodes for retrieving fitness data from the tracker (READ-TRQ). The base forwards this message, retrieves the fitness data from the tracker (TRQ-DATA) and sends it to the webserver. In the last phase, the webserver sends to the tracker through the base, opcodes to WRITE updates provided by the user. The webserver sends opcodes to ERASE the fitness data from the tracker. The base forwards the ERASE request to the tracker, who then erases the contents of the corresponding read memory banks. The webserver replies with the opcode to CLOSE the tracker.

B. Vulnerabilities

During the reverse engineering process, we discovered several fundamental vulnerabilities:

Fitbit: cleartext login information. During the initial user login via the Fitbit client software, user passwords are passed to the webserver in cleartext and then stored in log files on the base. Garmin uses encryption only during the login step.

Fitbit and Garmin: cleartext HTTP data processing. For both Fitbit and Garmin, the tracker's data upload operation uses no encryption or authentication. All the tracker-to-webserver communications take place in cleartext.

Garmin: faulty authentication during Pairing. The authentication in the Pairing procedure of Garmin assumes that the base follows the protocol and has not been compromised by an attacker. The authentication process is not mutual: the tracker does not authenticate the base.

C. The FitBite and GarMax Tools

We have built FitBite and GarMax, tools that exploit the above vulnerabilities to attack Fitbit Ultra and Garmin Forerunner. FitBite and GarMax consist of separate modules for (i) discovering and binding to a nearby tracker, (ii) retrieving data from a nearby tracker, (iii) injecting data into a nearby tracker and (iv) injecting data into the social networking account of a tracker owner. We have built FitBite and GarMax over ANT-FS, in order to connect to and issue (ANT-FS) commands to nearby trackers. The attacker needs to run FitBite or GarMax on a base he controls.

D. Attacks and Results

Tracker Private Data Capture (TPDC). FitBite discovers tracker devices within transmission range and captures their fitness information: Fitbit performs no authentication during tracker data uploads. We exploit Garmin's assumption of an honest base to use GarMax, running on a corrupt base, to capture data from nearby trackers. On average, the TPDC attack on both Fitbit and Garmin took less than 13s. Table I

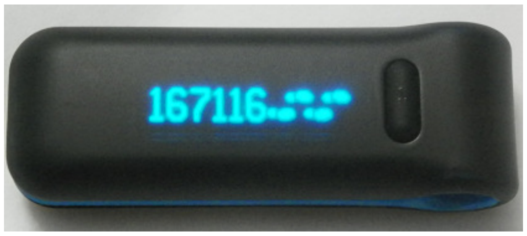


Fig. 4. Outcome of Tracker Injection (TI) attack on Fitbit tracker: The daily step count is unreasonably high (167,116 steps).

summarizes the information captured by FitBite and GarMax. Figure 3 shows the reconstructed exercise circuit of a victim, with data we recovered from a TPDC attack on Garmin. The GPS location history can be used to infer the user's home, locations of interest, exercise and travel patterns.

Tracker Injection (TI) Attack. FitBite and GarMax use the reverse engineered knowledge of the communication packet format, opcode instructions and memory banks, to modify and inject fitness data on neighboring trackers. On average, this attack takes less than 18s, for both FitBite and GarMax. Figure 4 shows a sample outcome of the TI attack on Fitbit.

User Account Injection (UAI) Attack. We used FitBite and GarMax to report fabricated fitness information into our social networking accounts. We have successfully injected unreasonable daily step counts, e.g., 12.58 million in Fitbit. Fitbit did not report any inconsistency, especially as the corresponding distance we reported was 0.02 miles! The UAI attack takes only 6s on average. Furthermore, by injecting fraudulent fitness information into Earndit [22], an associated site, we were able to accumulate undeserved rewards, including 200 Earndit points, redeemable for a $20 gift card.

IV. A Protocol for Lightweight Security

We now propose SensCrypt, a lightweight protocol for providing secure data storage and communication in fitness centric social sensor networks. Section VI also describes our evaluation of a solution based on public key cryptosystems.

Protocol overview. Let U denote a user, T denote her tracker, B a base and W the webserver. T's memory is divided into records, each storing one snapshot of sensor data. The memory is organized using a circular buffer structure, to ensure an even wear. T shares a symmetric key K_T with W. W also maintains a unique secret key K_W for each tracker T.

To prevent Inject attacks, all communications between T and W are authenticated with K_T. To prevent Inspect, Capture and JTAG-R attacks, we encode each tracker record using two

Type of data	FitBite	GarMax
Device info	✓	✓
User profile, schedules, goals	✓	✓
Fitness data	✓	✓
(GPS) Location history	✗	✓

TABLE I
Types of data harvested by FitBite and GarMax from Fitbit and Garmin. Garmin provides GPS tagged fitness information, which GarMax is able to collect.

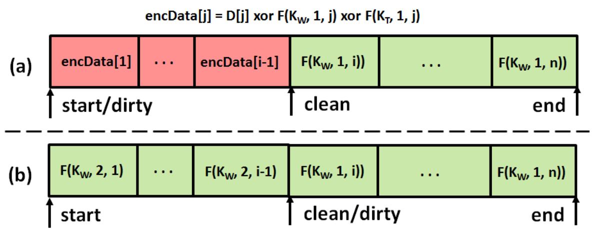


Fig. 5. Example SensCrypt tracker memory (mem). Light green denotes "clean", unwritten areas. Red denotes areas that encode tracker sensor data. (a) After (i-1) records have been written. The ctr is 1. (b) After $Upload$ occurs at state in (a). The ctr becomes 2, to enable the creation of fresh PRNs, overwritten on the former red area.

pseudo-random numbers (PRNs). One PRN is generated by W using K_W and written on T during data sync protocols. The other PRN is generated by T using K_T at the time when the record is written on its memory. Both PRNs can later be reconstructed by W. This approach significantly increases the complexity of an attack: the attacker needs to capture the encoded data and both PRNs to recover the cleartext data.

A. The SensCrypt Protocol

Let id_U, id_B, and id_T denote the public unique identities of U, B, and T. U has an account with W. W manages a database Map that has an entry for each user and tracker pair: $Map[id_U, id_T]$ = $[id_U,\ id_T,\ K_T,\ K_W,\ ctr]$. Each tracker is factory initialized with a symmetric key K_T and a counter ctr initialized to 1. K_T and ctr are also stored in $Map[id_U, id_T]$. K_W is a per-tracker symmetric key, kept secret by W.

SensCrypt consists of 2 procedures, $RecordData$ and $Upload$. $RecordData$ is invoked by T to record new sensor data; $Upload$ allows it to sync its data with W. We now describe the organization of the tracker memory.

Tracker Memory Organization. Let mem denote the memory of T. mem is divided in "records" of fixed length (e.g., 64 bytes for Fitbit, 80 bytes for Garmin). Each record stores one report from the tracker's sensors (see Section II-A). We organize time into fixed length "epochs" (e.g., 2s long for Fitbit, 1-9s long for Garmin). $RecordData$ records sensor data once per epoch. mem is organized using a circular buffer. The *dirty* pointer is to the location of the first written record, and the *clean* pointer is to the location of the first record available for writing. When reaching the end of mem, both records "circle" over to the *start* pointer. Figure 5 illustrates the SensCrypt tracker storage organization, after the execution of various $RecordData$ and $Upload$ procedures.

During $Upload$, each tracker record is reset by W to store a pseudo-random value. That is, the i-th record of the tracker's memory is set to hold $E_{K_W}(ctr, i)$, where K_W is the secret key W stores for T. The index i ensures that each record contains a different value. ctr counts the number of times mem has been completely overwritten; it ensures that a memory record is overwritten with a different encrypted value.

The RecordData Procedure. Commit newly recorded sensor data D to mem, in the next available record, pointed to by $clean$. T generates a new pseudo-random value, $E_{K_T}(ctr, clean)$, and xors it into place with $mem[clean]$ =

$E_{K_W}(ctr, clean)$ and D:

$$mem[clean] = D \oplus E_{K_T}(ctr, clean) \oplus E_{K_W}(ctr, clean).$$

The *clean* pointer is then incremented. When reaching the *end* of mem, *clean* circles back to *start*. We call "red" the written records and "green" the records available for write. *dirty* and *clean* enable us to reduce the communication overhead of $Upload$ (see next): instead of sending the entire mem, T sends to W only the red records.

The Upload Procedure. We present the SensCrypt $Upload$ as an extension of the corresponding Fitbit protocol illustrated in Figure 2. In the following, each message M sent between T and W is accompanied by an authentication value $Hmac(K_T, M)$, where Hmac is a hash based message authentication code [19]. The receiver of the message uses K_T to verify the authenticity of the sender and of the message. For simplicity of exposition, in the following we do not show the Hmac value.

$Upload$ extends steps 6b and 7 of the Fitbit $Upload$. Specifically, when T receives the READ-TRQ command (step 6a), it compares the *dirty* and *clean* pointers. If *dirty* < *clean* (see Figure 5(a)), T sends to W, through B,

$$\mathtt{T \rightarrow B \rightarrow W : TRQ - DATA, id_T, mem[dirty..clean]},$$

where $mem[dirty..clean]$ denotes T's red memory area. For each record i between *dirty* and *clean*, W uses keys K_T and K_W and the current value of *ctr* to recover the sensor data: $D[i] = mem[i] \oplus E_{K_T}(ctr, i) \oplus E_{K_W}(ctr, i)$. Then, in step 7 of Upload (see Figure 2), W sends to T:

$$\mathtt{W \rightarrow B \rightarrow T : WRITE, id_T, E_{K_T}(ctr + 1, E_{K_W}(ctr + 1, i))},$$

$\forall i$=*dirty*..*clean*. T uses K_T to decrypt each $E_{K_T}(ctr + 1, E_{K_W}(ctr + 1, i))$ value. If the first field of the result equals ctr+1, T overwrites $mem[dirty{+}i]$ with $E_{K_W}(ctr{+}1, i)$, then sets *dirty*=*clean*. Thus, mem[*dirty*.. *clean*] becomes green. The case where *clean* < *dirty*, occurring when *clean* circles over, past the memory end, is handled similarly, and is omitted here for brevity. We eliminate the ERASE communication (steps 8 and 9 in Figure 2) from the Fitbit protocol.

B. SensCrypt Properties

Due to space limitations, we only briefly state the properties of SensCrypt. We will include the proofs in the full paper version. We note that a JTAG-R attack is not able to recover encoded sensor data on T, since it is xored to a PRN previously generated by W. An Inspect attack fails, as communications between T and W are encrypted, or xor-ed with secret PRNs.

In SensCrypt, the base does not contribute to the messages it forwards between T and W. Hence, the base does not need to be authenticated. The use of the $ctr + 1$ value in communications through the base ensures message freshness.

SensCrypt ensures an even wear of tracker memory: the most overwritten memory record has at most 2 overwrites more than the least overwritten record. It imposes no storage overhead on trackers: sensor data is xor-ed in-place in mem. SensCrypt is user friendly, as the user is never involved.

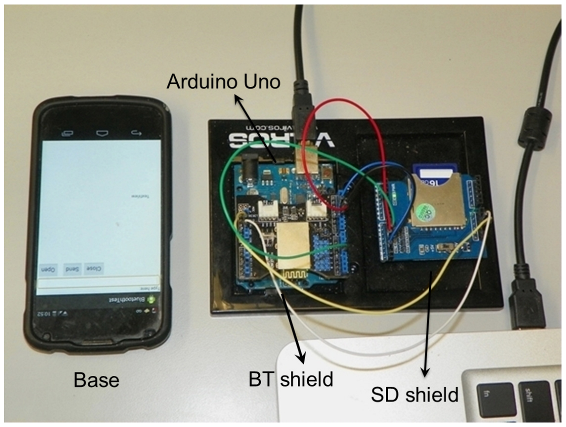


Fig. 6. Testbed for SensCrypt. Sens.io is the Arduino Uno device equipped with Bluetooth shield and SD card is the tracker. Nexus 4 is the base.

V. Sens.io: The Platform

We have built Sens.io, a prototype tracker, from off-the-shelves components. It consists of an Arduino Uno Rev3 [23] and external Bluetooth (Seeeduino V3.0) and SanDisk card shields. The Arduino platform is a good model of resource constrained trackers: its ATmega328 micro-controller has a 16MHz clock, 32 KB Flash memory, 2 KB SRAM and 1KB EEPROM.

The cost of Sens.io is \$52 (\$25 Arduino card, \$20 Bluetooth shield, \$2.5 SD Card shield, \$4 SD card, see Figure 6), a fraction of Fitbit's (\$99) and Garmin's (\$299) trackers.

SensCrypt. We have implemented a general, end-to-end SensCrypt architecture, as illustrated in Figure 7. We have implemented the tracker both in Arduino's programming language (a Wiring implementation [24]), and, for generality, in Android. The base component (written exclusively in Android) is a simple communication relay. We implemented the webserver using Apache Tomcat 7.0.52 and Apache Axis2 Web services engine. We used the MongoDB 2.4.9 database to store the Map structure. We implemented a Bluetooth [25] serial communication protocol between the tracker and the base.

The testbed. We used Sens.io for the tracker, an Android Nexus 4 with 1.512 GHz CPU for the base, and a 2.4GHz Intel Core i5 Dell laptop with 4GB of RAM for the webserver. We used Bluetooth for tracker to base communications and Wi-Fi for the connectivity between the base and the webserver. Figure 6 illustrates our testbed.

VI. Evaluation

We evaluate system performance on the above testbed. All results are averages over at least 10 independent protocol runs.

Public key crypto. We have explored the feasibility of public key cryptosystems to secure tracker storage and communications. We implemented FitCrypt, a solution where each tracker stores a public key. The corresponding private key is only known by the webserver. Each sensor data record is encrypted with the public key before being stored on the tracker. FitCrypt with RSA (FitCrypt-RSA) with a 2048 bit key takes 2.3s to encode a single record, when run on Sens.io. We also implemented FitCrypt-ECC, based on ECIES (Elliptic Curve Integrated Encryption Scheme), an elliptic curve crypto (ECC)

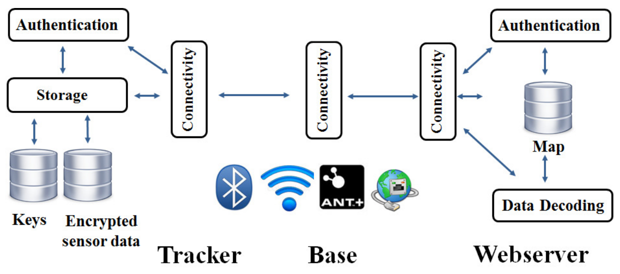


Fig. 7. SensCrypt architecture. The tracker relies on locally stored key K_T to authenticate webserver messages and encode sensor data. The webserver manages the Map structure, to authenticate and decrypt tracker reports.

solution that uses a 224 bit key size, the security equivalent of RSA with 2048 bit modulus. When run on Sens.io, FitCrypt-ECC takes 2.5s to encode a single sensor record. When the data recording frequency is lower than once in 3s, FitCrypt (RSA and ECC) cannot complete the operation on time.

A. *Sens.io Performance*

Tracker: RecordData overhead. On Sens.io, SensCrypt's $RecordData$ takes 6.02ms to encode Fitbit data and 6.06ms for Garmin data. Unlike FitCrypt (RSA and ECC), SensCrypt is a viable alternative for resource constrained trackers.

Webserver: Storage overhead. The structure Map stores a record for each user and tracker pair, consisting of user and tracker ids (8 byte long each), a salt (16B), password hash (28B), 2 symmetric keys (32B each) and a counter (1B). Thus, a Map entry stores 133 bytes. For a 1 million user base, the Map size is 127MB. The average time to retrieve a record from the MongoDB representation of this Map is 158ms.

End-to-end Upload overhead. We consider a "Fitbit" scenario where the $Upload$ procedure runs once every 15 minutes. With a $RecordData$ frequency of once every 2s (usual in Garmin), and a record size of 64B, SensCrypt uploads and overwrites 71 blocks of 512B each. The tracker side of SensCrypt's $Upload$ procedure takes 502ms. The 200ms server overhead is dominated by the 158ms of retrieving a record from a 1 million entry Map. Due to Arduino RAM limitations, the communication cost of SensCrypt's $Upload$ is 153ms. SensCrypt's total $Upload$ time is thus 846ms, 12 times faster than FitCrypt (whose total exceeds 10s) and twice faster than Fitbit's $Upload$ (1481ms on average). This gain is partly due to SensCrypt's optimization of only uploading the written blocks, instead of the entire memory.

VII. Comparison with State of the Art

In the context of implantable medical devices (IMDs) Halperin et al. [10] propose zero power notification, authentication and key exchange solutions, while Rasmussen et al. [11] propose proximity based access control solutions. The different mission of fitness trackers creates different design constraints. First, unlike IMD security, where the focus is on authentication and key exchange, SensCrypt's focus is on the secure storage and communication of tracker data. This is further emphasized by our need to also consider attackers that can perform Capture and JTAG-R attacks, for both trackers and bases (readers in the IMD context). While such attacks may not be possible for IMDs, and IMD readers may be expensive enough to afford tamper proof memory, these assumptions do not hold for most existing fitness social sensor network solutions. Furthermore, user interaction is natural for several IMD solutions. To avoid annoying users, fitness security solutions should minimize user involvement.

VIII. Conclusions

We identified and exploited vulnerabilities in the design of Fitbit and Garmin, to launch inspection and injection attacks. We presented SensCrypt, a secure and efficient solution for storing and communicating tracker sensor data. SensCrypt imposes minimal computation and communication overhead on trackers, and is resilient even to attackers able to probe the memory of captured trackers.

IX. Acknowledgments

This work has been supported by the US ARMY through award W911NF-13-1-0142.

References

[1] Holter Monitor. https://en.wikipedia.org/wiki/Holter_monitor.
[2] Nike+. http://nikeplus.nike.com/plus/.
[3] Fitbit. http://fitbit.com/.
[4] Garmin Forerunner. http://sites.garmin.com/forerunner610/.
[5] Jawbone UP24. https://jawbone.com/up.
[6] Body Media. http://www.bodymedia.com/.
[7] Jawbone takes a big bite out of health tech: acquires BodyMedia, launches Up app platform. http://venturebeat.com/2013/04/30/jawbone-takes-a-big-bite-out-of-health-tech-acquires-bodymedia-launches-up-app-platform.
[8] Please Rob Me. http://www.http://pleaserobme.com/.
[9] Kota Tsubouchi, Ryoma Kawajiri, and Masamichi Shimosaka. Working-relationship detection from fitbit sensor data. In *Proceedings of the 2013 ACM conference on Pervasive and ubiquitous computing adjunct publication*, UbiComp '13 Adjunct, pages 115–118, 2013.
[10] D. Halperin, T. Heydt-Benjamin, B. Ransford, S. Clark, B. Defend, W. Morgan, K. Fu, T. Kohno, and W. Maisel. Pacemakers and implantable cardiac defibrillators: Software radio attacks and zero-power defenses. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 129–142, 2008.
[11] K. B. Rasmussen, C. Castelluccia, T. S. Heydt-Benjamin, and S. Capkun. Proximity-based access control for implantable medical devices. In *ACM Conference on Computer and Communications Security*, 2009.
[12] Chunxiao Li, A. Raghunathan, and N.K. Jha. Hijacking an insulin pump: Security attacks and defenses for a diabetes therapy system. In *Proceedings of the IEEE International Conference on e-Health Networking Applications and Services (Healthcom)*, 2011.
[13] Mototola MotoActv. http://www.motorola.com/us/MOTOACTV-16GB-Golf-Edition/121481.html.
[14] Basis B1. http://www.mybasis.com/.
[15] Nest Thermostat. https://nest.com/thermostat/life-with-nest-thermostat/.
[16] WeMo Switch. http://www.belkin.com/us/p/P-F7C027/.
[17] Sense: The meaning of life. https://sen.se/store/mother/.
[18] Ing Breeuwsma. Forensic imaging of embedded systems using JTAG (boundary-scan). *Digital Investigation*, 3, 2006.
[19] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying hash functions for message authentication. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '96, pages 1–15, 1996.
[20] Libfitbit: Library for accessing and transfering data from the fitbit health device. https://github.com/qdot/libfitbit.
[21] ANT-FS and FIT. http://www.thisisant.com/developer/ant/ant-fs-and-fit.
[22] Earndit: We reward you for exercising. http://earndit.com/.
[23] Arduino Uno. http://arduino.cc/en/Main/arduinoBoardUno.
[24] Arduino Guide. http://arduino.cc/en/Guide/Introduction.
[25] Bluetooth SIG. Specification of the bluetooth system, 2001.