Journal of Network and Computer Applications 152 (2020) 102509

Contents lists available at ScienceDirect

Journal of Network and Computer Applications

journal homepage: www.elsevier.com/locate/jnca

PGFIT: Static permission analysis of health and fitness apps in IoT programming frameworks

Mehdi Nobakht[a,*], Yulei Sui[b], Aruna Seneviratne[c], Wen Hu[d]

[a] School of Engineering and Information Technologies, UNSW Canberra, Australia
[b] Faculty of Engineering and Information Technology, University of Technology Sydney, Australia
[c] School of Electrical Engineering and Telecommunications, UNSW Sydney, Australia
[d] School of Computer Science and Engineering, UNSW Sydney, Australia

A R T I C L E I N F O

Keywords:
Permissions
Principle of least privilege
Static program analysis
IoT programming frameworks
Google Fit
Health and fitness data

A B S T R A C T

Popular Internet of Things (IoT) programming frameworks, such as Google Fit, enable third-party developers to build apps that store and retrieve user data from a variety of data sources such as wearable devices. Most of these apps, particularly those that are health and fitness-related, collect potentially sensitive personal data and send it to cloud servers. Analogous to Android OS, IoT programming frameworks often follow similar permission model; third-party apps on IoT platforms prompt users to grant the apps the access to their private data stored on cloud servers of IoT programming frameworks. Most users have a poor understanding of why these permissions are being asked. This can often lead to unnecessary permissions being granted, which in turn grant these apps with excessive privileges. Over-privileged apps might not be harmful to users when they are used as designed, however, they can potentially be exploited by a malicious actor in a cyber security attack. This is of particular concern with health and fitness apps, which may be exploited to leak highly sensitive personal information. This paper presents PGFIT, a static permission analysis tool that precisely and efficiently identifies privilege escalation in third-party apps built on top of a popular IoT programming framework, Google Fit. PGFIT extracts the set of requested permission scopes and the set of used data types in Google Fit-enabled apps to determine whether the requested permission scopes are actually necessary. PGFIT performs graph reachability analysis on inter-procedural control flow graph. PGFIT serves as a quality assurance tool for developers and a privacy checker for app users. We evaluated PGFIT using a set of 20 popular Google Fit-enabled apps downloaded from Google Play. Our tool successfully identified the unnecessary permission scopes granted in our data set apps and found that 6 (30%) of the 20 apps are over-privileged.

1. Introduction

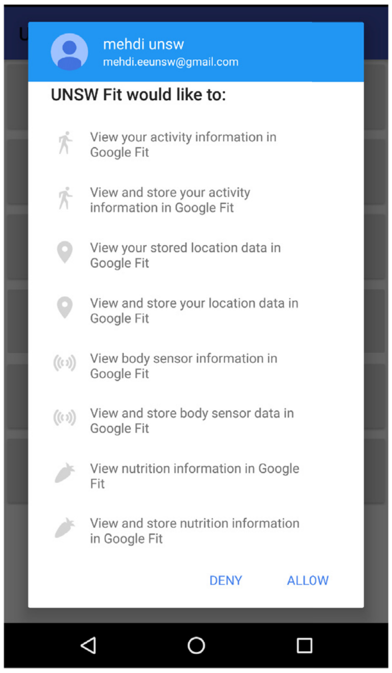
The Internet of Things (IoT) largely consists of embedded devices such as wearable accessories that generate data, and end applications (e.g. mobile apps) that consume this data and optionally take actions. Recently, programming frameworks have emerged which enable developers to create third-party apps that can process this data for a variety of purposes. In particular, IoT programming frameworks for health and fitness-tracking are receiving more attention due to the popularity of wearables, smart watches and the proliferation of third-party IoT apps. Google Fit (Google, 2017a), Apple's HealthKit (Apple, 2017), Samsung Digital Health Platform (Samsung, 2017), and Microsoft's HealthVault (Microsoft, 2017) are a few examples of such platforms.

As a representative IoT programming framework, Google Fit paves the way for third-party app programmers to efficiently write health and fitness apps by providing high-level centralised APIs, without the need to understand low-level implementation details. Google Fit also has a central cloud-based repository that allows a user to store and retrieve health and fitness-related data from multiple apps and devices such as activity trackers and smart watches.

Google Fit APIs provide access to data with specified permissions associated with a Google user account. Google Fit uses the OAuth protocol (The oauth 1.0 protocol, 2010; The oauth 2.0 protocol, 2012) to authorise third-party apps by obtaining consent from users to access fit-

* Corresponding author.
E-mail address: mehdi.nobakht@unsw.edu.au (M. Nobakht).
URL: https://mehdi-nobakht.github.io/ (M. Nobakht).

https://doi.org/10.1016/j.jnca.2019.102509
Received 29 April 2019; Received in revised form 6 November 2019; Accepted 14 December 2019
Available online 23 December 2019
1084-8045/Crown Copyright © 2020 Published by Elsevier Ltd. All rights reserved.


mehdi unsw
mehdi.eeunsw@gmail.com

UNSW Fit would like to:

- View your activity information in Google Fit
- View and store your activity information in Google Fit
- View your stored location data in Google Fit
- View and store your location data in Google Fit
- View body sensor information in Google Fit
- View and store body sensor data in Google Fit
- View nutrition information in Google Fit
- View and store nutrition information in Google Fit

DENY ALLOW

Fig. 1. A typical google fit consent screen.

ness information. Authorised apps are then allowed to store or read the user's fitness information. Google Fit blends the stored data from a variety of apps and makes them available to the user and authorised apps on behalf of the user. As far as we know, apart from Google Fit's own app, 43 health and fitness apps based on the Google Fit programming framework are available on the market (e.g., exercise activities trackers, heart rate monitors and calorie counters (Google, 2017b)). These apps are developed by, among others, Motorola, Intel, Sony, Adidas and Nike.

IoT programming frameworks often follow similar permission models to smartphone OS permissions, and IoT apps must be authorised by the end-user to access health and fitness data. In Android's case, users are asked to grant permissions to a third-party app via a dialogue screen that granularly lists the permission scopes with "deny" and "allow" buttons (see Fig. 1). Providing proper permission scopes is important to avoid privilege escalation or permission misuse, as it exposes user to the risk of leaking private user information. Yet our studies indicate, however, that many apps request access to more data than they actually need to perform their core functions.

Nowadays, insurance companies offer incentives to encourage people to be active (Assure, 2017; Medibank, 2017; AIA, 2017). The rewards can be in the form of premium discounts or credits in a joint loyalty program. In such voluntary plans, users agree to be monitored by fitness devices, and whether biometric data (e.g., daily steps and heart rates) can be shared privately with the insurer. Using data collected from the user, an insurer can determine whether the user is eligible for a reward. However, the user may only be willing to share a subset of their health data with their insurer, and might not wish for other parties to access certain information such as hospital appointment records.

Granting the permissions in any app requires that third-party developers understand the usage of APIs within the context of IoT frameworks. Giving unnecessary permissions to an app may expose users to privacy risks such as leakage of sensitive user information such as blood pressure levels or physical location data to an unnecessary wide audience.

Let us consider an over-privilege scenario in a third-party app built upon Google Fit, whose cloud contains both user fitness data and health data. The user wishes only to share their fitness data with this app, not their health data. Instead, the user's heath data will be accessed via another medical app connecting to the same Google Fit cloud. The medical app allows the user to organise their appointments and other health needs. However, if the fitness app developer intentionally or unintentionally requests to access both types of data on Google Fit cloud, the user has to grant the blanket permission in order to use this third-party app (even he/she may not realise this is an unnecessary authorisation). Consequently, the over-privileged fitness app causes sensitive health data to be leaked to this third party (e.g. the insurer).

The increasing deployment of various third-party apps on top of IoT platforms raises security and privacy risks. Thus, it is important to ensure that permissions granted will not be abused. However, detecting excessive privileges is challenging due to the sheer size of modern IoT apps and the complications of API usage in IoT programming frameworks.

This paper focuses on discovering over-privilege permissions to access user data within IoT platforms. In particular, we analyse a dataset of apps working with the Google Fit framework to determine how they adhere to the least-privilege principle to protect sensitive user data. To this end, we perform comprehensive studies in the Google Fit access control mechanism to gain insights into their structure and key requirements. We have summarised the scopes of permission access and data types used in existing Google Fit APIs.

We present PGFit, a static analysis tool to identify overprivileged apps developed upon the Google Fit programming framework. We formulate the permission analysis as a source-sink graph reachability problem. PGFit introduces context-sensitive reachability analysis into over-privilege detection in IoT apps. By considering event-driven callbacks in Android apps, PGFit performs forward control-flow reachability analysis to obtain the program slices from a source node (a Google Fit API call which grants a permission scope) to a sink node (an API call which consumes data types). On top of the slices of a given source-sink pair, PGFit performs backward reachability analysis to compute the values of the data type variables from the sink node. An over-privilege is reported in an app if a permission granted at a source node is never used (based on the data types) in any of its sink nodes. We have evaluated PGFit on a set of 20 third-party apps developed upon Google Fit. Our analysis found that six (30%) of them are over-privileged.

The paper makes the following key contributions:

- We have identified privilege escalation in IoT apps built upon the existing popular IoT programming framework Google Fit. We have performed comprehensive studies of the Google Fit permission control mechanism by summarising the existing Google Fit APIs, permission scopes and data types.
- We present PGFit, a static analysis tool that introduces context-sensitive reachability analysis into over-privilege detection in apps on top of IoT programming frameworks.
- We have implemented PGFit as a software tool and evaluated it over a set of 20 IoT Apps built on Google Fit. Our tool successfully detected that six (30%) of them are over-privileged. PGFit reduces false alarm rates by 10% by eliminating "dead codes" compared to a naive approach that does not employ control flow reachability analysis.

The rest of this paper is organised as follows. Section 2 describes our studies in Google Fit and ASM, a static analysis tool on which PGFit is built. Section 3 gives a representative threat model to highlight how an app can be compromised to disclose private user information. Section 4 presents PGFit and its main components. The tool aims to perform static analysis on Google Fit-enabled apps to check privilege escalations. Section 5 reports the result of our analysis on a set of 20 Google Fit-enabled apps. Related work is summarised in Section 6 with conclusions in Section 7. In order to assist readers, we provide in Table 1 a list of acronyms along with brief definitions as used throughout this article.

Table 1
Acronyms and definitions.

Acronym	Definition
API	Application Programming Interface
APK	Android Application Package
App	Application
ASM	ASM[a]Library
BLE	Bluetooth Low Energy
DEX	Dalvik EXecutable
HTTPS	Hypertext Transfer Protocol Secure
ICC	inter-component communication
ICFG	Inter-procedural Control Flow Graph
IoT	Internet of Things
JAR	Java ARchive
JVM	Java Virtual Machine
OAuth	Open Authorisation
OS	Operating System
REST	Representational State Transfer
SDK	Software Development Kit
SDN	Software-defined Networking
SSA	Static Single Assignment
UI	User Interface
URI	Uniform Resource Identifier
XML	eXtensible Markup Language

[a] The ASM name does not mean anything; it is just a reference to the *__asm__* keyword in *C*, which allows some functions to be implemented in assembly language (Bruneton et al., 2002).

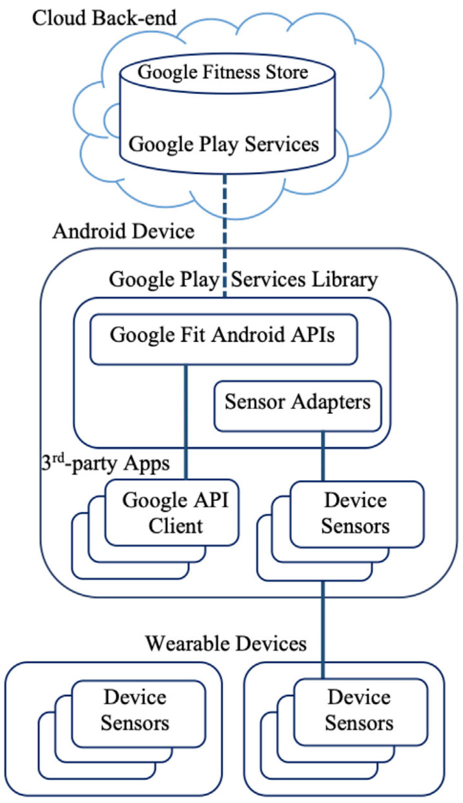


Fig. 2. An overview of Google Fit on Android.

2. Background and our studies of Google Fit

This section describes our studies of the Google Fit framework including (1) its overall structure and basic types of fitness related APIs, (2) its data storage and representation, and (3) its permission and user control mechanism. We also provide background information on ASM, a static analysis library, on which PGFIT was developed to perform static permission analysis for Google Fit-enabled apps.

2.1. Google Fit: IoT programming framework

Google Fit is a health and fitness-tracking platform developed by Google which uses sensors in a user's activity tracker or mobile device to record physical fitness activities such as walking or cycling. It also enables the user to measure stored information against their fitness goals in order to provide a comprehensive view of fitness activities. Google Fit, with its open programming framework, enables third-party developers to build health and fitness apps to collect, insert, or query user fitness-related data. Google Fit-enabled apps upload data to a central repository, where it remains owned by the user and is associated with their Google account. Google Fit blends a user's data collected from various sources and makes it available to the user and also to authorised third-party apps from a single location. In this way, users are able to query the stored fitness data and to track their progress.

2.1.1. Overview

The central repository of Google Fit can be accessed by either Android or Web apps. Google Fit thus provides two sets of APIs: Android APIs and Web REST API. Google Fit APIs for Android devices have two main functionalities; **(i)** to provide access to data streams from sensors embedded in Android devices and sensors available in companion devices such as wearables, and **(ii)** to provide access to data history and to allow apps to obtain the stored data. Android apps in these two scenarios can be seen as data sources and data sinks respectively. Google Fit REST API, however, is not supposed to connect to any sensor. Thus, it is intended to access user data in the fitness store. The REST API can be used by any Web browser on any platform. Since our study involves permission analysis on Android fitness apps, we focus on the Google Fit Android APIs interface.

The Google Fit ecosystem for Android devices consists of four major components: *Google Fitness Store*, *Google Fit APIs*, *Sensor Adapters* and *Third-party Android Apps*. Fig. 2 provides a general overview of the Google Fit architecture. *Google Fitness Store* is a backend cloud storage managed by Google which stores fitness data from various apps and devices. Users track fitness activities and access fitness data on Google Fit platform from *Third-party Android Apps* which can insert data and also query the fitness store on behalf of users. *Sensor Adapters* interface physical sensors on the mobile device and companion devices to Google Fit. *Google Fit APIs* link the Google Fitness Store to third-party apps and sensors.

Google Fit Android APIs consist of various APIs to access the fitness store. A third-party fitness app interfaces to a fitness service via an appropriate API. Below, we briefly describe these APIs in order to provide the context for the permission analysis that will be presented later.

- *Sensors API* provides access to raw sensor data streams and easily discovers available sensor data sources.
- *Recording API* enables automatic collection of sensor data in a power-efficient manner.
- *History API* allows apps to perform bulk operations such as inserting, updating, and deleting fitness data.
- *Sessions API* enables apps to create an activity time interval (sessions) in the fitness store in real time or after a fitness activity has finished.
- *BLE API* provides access to Bluetooth Low Energy sensors by enabling the app to look for available BLE devices and use embedded sensors in them.
- *Config API* provides methods to create custom data types and also methods for disconnection from Google Fit.

2.1.2. Connecting to Google Play

To use Google APIs such as authentication, Map, and Google Fit, a Google-enabled app must first connect to Google Play services using Google common API. Once connected to Google Play services, the app can call Google API methods. The Google Play services system is composed of two components: **(i)** a client library that resides in a third-party app, and **(ii)** an implementation of Google Play services that runs as a background service in the Android OS. The client library interfaces third-party apps to Google Play services and allows apps to obtain authorisation from users to gain access to Google services with their credentials. To connect to Google APIs, a third-party app must create an instance of GoogleAPIClient which provides a common entry point to all Google Play services. The GoogleAPIClient provides methods that allow the app to specify what Google APIs are required and the desired authorisation scopes too.

To analyse a Google Fit-enabled app, we used the publicly available Google Fit API (Google, 2017c) and collected all information about Google Fit API interface methods. We additionally used Google developer documents (Google, 2017d) and collected information regarding Google common API methods which are typically used in fitness apps. The collected specification from Google common APIs and Google Fit APIs includes method names, descriptions, and target classes. We saved the collected specification in a list to be used later to identify Google Fit API calls and Google common API calls.

More specifically, a Google-enabled app should invoke the addAPI method and pass the required API token as an input parameter to specify what Google API is required. The required Google APIs appear as string literals in the field of the Api class and the values of Google APIs strings are documented by Google references. For example, an app that requires access to raw sensor data streams must add the *Sensors API* to enable this part of Google Play services.

2.1.3. Fitness data representation

Google Fit defines high-level representations for fitness data stored in its repository, in order to make it easier for apps to interact with the fitness store on any platform and to extract the required information. We briefly elaborate on Google Fit representations for embedded sensors and fitness data types and explain how Google Fit abstracts away unnecessary information.

Google Fit defines *Data Sources* to represent unique sources of sensor data. Data Sources contain information which can be used to uniquely identify the hardware device and the app collecting the data. Google Fit uses the notion of *Data Types* to represent the format of streams of fitness data such as step count or heart rate. A single value of data in a data type's stream from a particular data source is represented by a *Data Point* in Google Fit and includes a timestamp. A Data Point can hold a value for either an instantaneous observation or aggregate data over a time interval. For example, com.google.step_count.delta is an instantaneous data type to represent the number of steps since the last reading, while com.google.heart_rate.summary is an aggregate data type which holds statistics for beats per minute during a time interval.

The Data Types representation in Google Fit abstracts away details for apps wishing to access fitness data. In this way Google Fit removes unnecessary details such as how the data is being collected or what sensors, hardware or even apps are being used. To illustrate how this device-independent abstraction works, consider a case where a user uses two different Google Fit-enabled apps to record their activities. The first app tracks cycling activities by using sensors in a wearable device. The other app records walking activities by utilising embedded sensors in the user's smart phone. Both apps expose raw sensor data from hardware sensors to Google Fit. Each value in such streams of data contains information about the user's activity. The user can later use a third app to extract total calories expended over a time interval. For this purpose, the third app can use the com.google.calories.expended data type to query such information from Google Fit Store and deliver it to the user. In this way, Google Fit abstracts away any details from available data points in the fitness store.

2.1.4. Permissions and user controls

Google Fit requires user consent before apps can access user fitness data. Apps must obtain authorisation by specifying the *scope* of access to fitness data and the level of access. Google Fit classifies fitness data into four different data types: *activity*, *biometric*, *location*, and *nutrition*. The variation of fitness data types with read and write privileges creates a set of 8 different authorisation scopes. For instance, the FITNESS_ACTIVITY_READ scope provides read-only access to all data related to a user's activities. Table 2 shows string literals representing each permission scope $ps \in PS$ in Google Fit along with its corresponding set of data types D_{ps}. Note that for any two permission scopes $ps1$ and $ps2$, their data types D_{ps1} and D_{ps2} are always disjointed, i.e., $D_{ps1} \cap D_{ps2} = \varphi$

Google Fit provides an OAuth-based authentication service for apps to obtain required authorisation scopes. OAuth service involves a multi-step authorisation dialogue over HTTPS between three entities: Google Fit cloud backend, the app wishing to access fitness data, and the user who owns the fitness data. The app first must specify one or more scopes of access. Once Google Fit receives the app request, the user is prompted to grant the app the required permissions. The user must approve or deny the request at once. Fig. 1 shows a consent screen for the Google Fit app developed by authors.

Once the user approves the app request to access the user's fitness data, Google Fit sends the authorisation code to the app and upon app acknowledgement sends an access token. Having acquired a *scoped* OAuth bearer token, the app can make Google Fit API calls to access all fitness data types defined by that scope.

More specifically, a Google-enabled app should invoke the addScope method with the required OAuth scope as an input parameter to specify the required authorisation scope. Google Fit defines authorisation scopes as string literals. In most cases, these strings are passed directly to the addScope method. However, in some cases, an instance of the Scope class from Google common API is created to return Google Fit scope strings. In these cases, the constructor method of this class accepts a Uniform Resource Identifier (URI) string to indicate the intended scope. The values of URI strings are documented in Google Fit API.

As seen in Fig. 1, the authorisation of Google Fit-enabled apps is coarse-grained; multiple permission scopes to access fitness data types are granted at once. Thus, the user should grant permission to the app to access all requested scopes or deny all. While this coarse-grained authorisation can improve the simplicity and stability of Google Fit platform, it also leaves users with no option to grant or deny permission scopes separately.

2.2. ASM library

There exist various publicly-available frameworks for analysis of Java apps when their source code are not accessible. Such frameworks perform analysis either statically or dynamically. Static analysis is conducted on some form of the source code or the object codes and involves binary analysis and re-writing, whereas dynamic analysis is performed after the app has been loaded into memory and just prior to execution. Examples of static analysis frameworks are Soot (Vallée-Rai et al., 1999) and WALA (Watson, 2017). There also exist analysis frameworks which can provide both static (offline) and dynamic (runtime) Java bytecode manipulation and analysis such as ASM API (Bruneton et al., 2002). We developed a static analysis tool using ASM API to perform analysis on compiled Java classes of Google Fit apps. We chose the ASM API for its well-designed architecture and modular API that is easy to use. The ASM API is also fast, robust, well-documented and has an open source license.

ASM API uses the visitor pattern (Palsberg and Jay, 1998), which allows one to define a new operation without changing the class of elements on which it operates. ASM library provides two set of APIs; the

Table 2
Scopes for Google Fit and related data types.

Permission	Scope	Data Types
Activity	SCOPE_ACTIVITY_READ SCOPE_ACTIVITY_READ_WRITE	TYPE_ACTIVITY_SAMPLES TYPE_ACTIVITY_SEGMENT AGGREGATE_ACTIVITY_SUMMARY TYPE_CALORIES_CONSUMED TYPE_CALORIES_EXPENDED AGGREGATE_BASAL_METABOLIC_RATE_SUMMARY TYPE_CYCLING_PEDALING_CADENCE TYPE_CYCLING_PEDALING_CUMULATIVE TYPE_CYCLING_WHEEL_REVOLUTION TYPE_CYCLING_WHEEL_RPM TYPE_POWER_SAMPLE TYPE_STEP_COUNT_CADENCE TYPE_STEP_COUNT_DELTA
Body	SCOPE_BODY_READ SCOPE_BODY_READ_WRITE	TYPE_BODY_FAT_PERCENTAGE AGGREGATE_BODY_FAT_PERCENTAGE_SUMMARY TYPE_HEART_RATE_BPM TYPE_HEIGHT TYPE_WEIGHT
Location	SCOPE_LOCATION_READ SCOPE_LOCATION_READ_WRITE	TYPE_DISTANCE_DELTA TYPE_LOCATION_SAMPLE TYPE_SPEED
Nutrition	SCOPE_NUTRITION_READ SCOPE_NUTRITION_READ_WRITE	TYPE_NUTRITION TYPE_HYDRATION

Core API with an event-based representation of classes and the *Tree API* providing an object-based representation. To build a class hierarchy of a Java project, ASM provides the ClassVisitor abstract class from Core API. This abstract class has two main components: the ClassReader class for parsing an existing compiled class, and the ClassWriter class for generating compiled classes directly in bytecode form. The MethodVisitor abstract class is another visitor interface to visit all methods of a compiled class. ASM API thus allows one to visit a compiled Java class and retrieve all kinds of information from the class such as fields, methods and inner classes.

ASM API provides MethodVisitor abstract class to visit a Java method. This class provides various methods to retrieve the Java method contents such as modifiers, name, parameter and return types and values. There is a method called visitParameter for extracting a parameter name. However, this method will only return a parameter name if a special compiler customisation has been set to include the parameter name. Typically, most Java tools that produce and consume compiled class files may not expect the larger static and dynamic footprint of class files that contain parameter names. In addition, some parameter names such as secret keys or passwords may expose information about security-sensitive methods.

While the process of obtaining names of parameters may at first appear straightforward, it is not simple to obtain the names of method parameters using ASM API. In order to address this, it is important to know how a JVM executes Java methods. We briefly describe this process here to be sufficient to infer names of method parameters (for a complete description, see the JVM Specification (Oracle, 2017)). A Java method consists of various elements such as its name, descriptors, exceptions and instructions, where instructions represent the code of the method. Inside a Java method, a method invocation is represented by a *Frame*. A Frame is used to store data and partial results and contains two parts: a local variable part and an operand stack. A JVM uses local variables to pass parameters on method invocations. The JVM supplies *bytecode instructions* to load constants or values (from local variables or fields) onto the operand stack. By examining these bytecode instructions for transferring values between the local variable and the operand stack, it is possible to infer parameters on method invocations.

3. Threat model

Health and fitness IoT programming frameworks store and maintain sensitive user data produced by embedded sensors in connected devices. Sensitive data can be activity-related such as calories consumed and step counts, or biometric data such as heart rate. While these frameworks provide tangible benefits to users, there is also a potential possibility to put users at risk of leaking their sensitive data. Denning et al. survey the security and privacy in IoT-based smart homes and warn that leaking of sensitive data could result in reputation and financial damage (Denning et al., 2013).

When an app connects to IoT programming frameworks, it specifies the scope of access to user data. The app may request permissions that are not necessary to what it really needs. In such cases, an attacker could take advantage of these escalated privileges. In this study we assume that an adversary compromises over-privileged fitness and health apps working with IoT frameworks, in order to acquire private user information. We thus aim to focus on app-level attacks that are launched through malware or vulnerable apps. In the former, malicious logic is disguised within the app at install time, while in the latter, the app may contain design or implementation flaws.

We assume that fitness and health apps run on smart phones or wearable devices whose hardware and software are trusted. Therefore, attacks from an untrustworthy baseline are outside the scope of this paper.

4. PGFit - design and implementation

Scope and Assumptions. We developed a static analysis tool called PGFit, which analyses Google Fit-enabled Android apps in order to determine whether the requested authorisation scopes are indeed needed. PGFit takes a given Android Application Package (APK) file as input and performs analysis on it to compute Google Fit scope over-

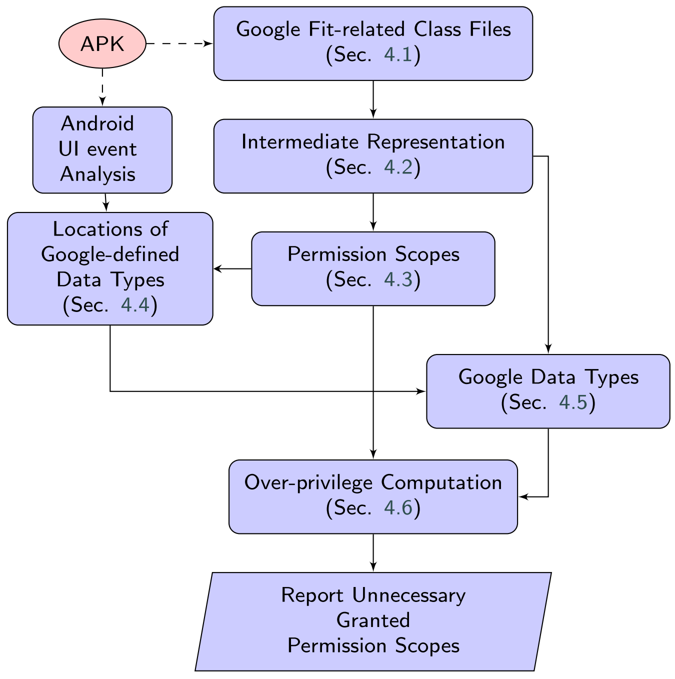

APK
Google Fit-related Class Files (Sec. 4.1)
Android UI event Analysis
Intermediate Representation (Sec. 4.2)
Locations of Google-defined Data Types (Sec. 4.4)
Permission Scopes (Sec. 4.3)
Google Data Types (Sec. 4.5)
Over-privilege Computation (Sec. 4.6)
Report Unnecessary Granted Permission Scopes

Fig. 3. An overview of PGFIT.

privilege. We assume apps are not obfuscated and can be analysed by ASM API.

Overview. An overview of PGFIT is shown in Fig. 3. PGFIT takes a given APK file of a third-party fitness app as input and carries out static analysis. Under the hood, there are six phases of analysis in PGFIT. In the first phase, Google-related class files will be identified among a large number of class files in a typical Google Fit-enabled app. This helps to avoid performing analysis on irrelevant class files. In the next phase, PGFIT uses ASM API to extract all method calls within the identified Google-related class files. This will build an intermediate representation which contains Google Fit method calls. Next, PGFIT will extract requested permission scopes. To determine whether the requested permissions are indeed necessary, PGFIT first locates potential procedures which may consume fitness data types. The procedures invoked by user interactions or containing permission requests may use fitness data types. PGFIT then creates the call graphs of such procedures and forward traverses the graphs to visit all program statements to identify Google method calls. It then performs a backward traversal to extract used Google Fit data types within the app. Finally, in the last phase, two sets of requested scope permissions and used Google Fit data types are compared to compute over-privilege in the app.

4.1. Identifying google-related class files

Android apps compiled to Dalvik Executable (DEX) bytecode that can be run on Android's Virtual Machine. Due to the similarity of DEX bytecode to Java bytecode and the availability of tools providing Java bytecode analysis, we decided to convert DEX bytecode to the corresponding Java bytecodes. To this end, PGFIT employs the publicly available *dex2jar* tool (dex2jar, 2017). In the first phase, PGFIT takes a given APK as input and employs *dex2jar* to generate an equivalent JAR file containing Java class files.

One of the challenges in analysing obtained JAR files from Android Google Fit-enabled apps is the sheer size of the JAR files. We unpacked app APKs in our study dataset (See Section 5.1) and observed that they contain roughly 12,600 class files, on average. Note that many of these classes originate from Java and Android native libraries or other app libraries which are irrelevant to Google Fit Analysis. Thus, performing resource intensive static analysis on all classes of an app is actually unnecessary.

To overcome this challenge, we first collected all information about Google Fit API method names and Google Fit-defined data type names from the publicly-available Google Fit APIs (Google, 2017c) and stored the Google Fit-related string names in a set. Then, a bash script was developed and embedded in PGFIT. The script searches for entries in the aforementioned set among all compiled class files of the JAR file. The output of this phase is a list of compiled class files related to Google Fit. In this way class files with no Google Fit API calls are filtered out.

4.2. Intermediate representation

We need to convert the compiled class files to some intermediate representation suitable for our analysis. PGFIT, in this phase, takes every class file in the list of Google Fit-related classes and then loads each from the JAR file using Java internal libraries. The compiled class files retain the structural information and many of symbols from the source code. Note that unlike a Java source file which can contain several classes, a compiled class file describes only one class. If a source file includes a *nested class*, then the main and inner class are compiled in two separate class files.

PGFIT employs ASM API to parse the compiled main class and nested classes (if there is any inner class) and to visit all of their methods to extract all method invocations. This list of method invocations also includes Google API invocations which are called in the fitness app. PGFIT stores this list of method invocations and other related information, including caller and callee classes and target methods along with their descriptors. A method descriptor itself is a list of type descriptors that describe the parameter types and return types of the method. The stored information is used in the next phase.

PGFIT examines every method invocation in the above list to determine whether it is a Google Fit API call or belongs to Google common APIs. It takes every method invocation and compares **(i)** the method name, **(ii)** method description, and **(iii)** the target class of method with the Google Fit specification obtained from Google API references, as explained in Section 2.1.2. If the method invocation matches Google Fit API method, PGFIT, it then labels it with the corresponding Google Fit API interfaces. The list of method invocations is then refined to contain only Google common API calls or Google Fit APIs.

4.3. Extracting permission scopes

In this phase PGFIT runs multiple threads to analyse every method invocation in the list of Google Fit-related method invocations. It aims to discover **(i)** what Google Fit APIs have been requested to connect and **(ii)** what authorisation scopes have been requested. PGFIT obtains this information from Google common API method calls.

Further to the explanation in Section 2.1.2 and 2.1.4, Google Fit-enabled apps should use addAPI and addScope methods to connect to the required APIs and specify the scope of access. Table 3 shows these two methods.

PGFIT searches in the list of Google Fit-related method invocations for the above two methods. The method invocation which adds a scope must satisfy two conditions: **(i)** the addScope method belonging to the Builder class must be used to configure an instance of GoogleAPIClient at the main entry point of Google Play services, and **(ii)** it must have either a string parameter whose value is a scope permission value in Table 2 or an instance of Scope which in turn returns Google Fit scope strings.

PGFIT leverages the MethodNode class from ASM API and extracts the requested APIs and permission scopes in a given Google Fit-enabled app. The obtained information is stored in a list to be used in later phases of analysis.

Table 3
Google fit APIs.

Google Fit APIs	Methods
Google Authorisation	public GoogleApiClient.Builder **addScope** (Scope scope)
Google Fitness Service	public GoogleApiClient.Builder **addApi** (Api api)

4.4. *Identifying the location of google-defined data types*

In order to discover all Google-defined data types that a Google Fit-enabled app uses, we first need to identify the procedures that may consume such data types. Analysing all procedures of the app with a large number of class files is an inefficient and resource-intensive task. Thus, in this phase PGFit aims to identify potential procedures containing Google Fit method invocations which may use Google-defined data types. Table 4 shows Google Fit methods extracted from publicly available Google Fit API (Google, 2017c).

Obviously, the procedure in which the app requests the required permission scopes is more likely to contain method invocations which use Google-defined data types. Thus, all the following statements and called procedures immediately after the point where the permissions are requested should be analysed. However, unlike this procedure there might be other procedures containing Google Fit method invocations. These procedures can be invoked via user interaction such as inter-component communication or inter-application communication (through another app). For example, consider an app that when launched requests read-write access to nutrition data types in Google Fit. The app is designed so that it does not use any nutrition data types until the user enters the consumed food (to record expended calories). All possible interactions with the app must therefore be analysed for any usage of Google-defined data types.

To this end, PGFit searches for Google-defined data types in a given app by analysing all statements and procedures **(i)** after a permission scope is requested, and **(ii)** from the beginning of all entry points of the app initiated by inter- and intra-application communications and which may contain Google Fit methods. For the former, PGFit is aware of the starting point of analysis, as explained in the previous section; for the latter, it is necessary to inspect the app and determine first all entry points of the app. Below, we briefly describe the steps that PGFit takes to locate those entry points that may lead to Google Fit methods.

PGFit first uses a publicly available tool named *apktool* (apktool, 2017) to decode the APK file of a given app. The tool reconstructs the resource folder of the app including its manifest file and layout files. After decoding the app, PGFit analyses the metadata specification in the manifest file to extract all declared activities. In Android mobile apps an activity serves as the entry point for an app's interaction with its user by providing a window as the activity's user interface (UI). Among extracted activities, PGFit selects Google Fit-related activity class files and parses them to obtain the associated UI layout ID by analysing the setContentView() method, which is a reference to the layout resource passed to it. In Android apps, UI layouts are recommended to be defined with XML files to provide a human-readable structure. All layouts in an app are stored in the *res/layout/*folder of its project. In the next step, PGFit finds the layout XML file name from its associated ID by parsing the R class, which contains resource IDs of all resources in the app *res/*directory. Having found the layout XML files, PGFit analyses them and extract a list of procedures in the app that can be called by the user. For instance, a user can interact with the app and input a consumed food item by clicking on a button provided on the screen. This action invokes a method in the app that makes use of a defined Google Fit data types such as TYPE_NUTRITION.

4.5. *Extracting google-defined data types*

Once potential procedures with Google-defined data type consumption are identified, PGFit analyses every statement of these procedures. It first builds a call graph over the procedures under examination and performs a forward traversal of the graph looking for any Google API method invocation which consumes a Google-defined data type. Arguments of a method can be passed by value or by reference. In cases where a reference to the data type is passed to the method invocation, PGFit backward traverses the graph to find the value of the reference and obtain the actual Google-defined data type.

4.5.1. *Call graph generation*

Typically, tasks and functionalities in mobile apps are spread across several procedures and end classes. Thus, it is necessary to perform *inter-procedural analysis* (Reps, 1997) which operates across all class files within the app. Since the information flows both from the caller procedure to its callee and in the opposite direction, we use *call graphs* to inform which procedure calls which. A call graph is a set of nodes (vertices) and edges such that each node represents either a *call site* (a place where a procedure is invoked) or a procedure and an edge represents the connection or relationship between the call site and the procedure. More specifically, PGFit uses *Inter-procedural Control Flow Graph* (ICFG) consisting of a set of nodes and a set of edges denoted by $< N, E >$. In ICFG, a node $n \in N$ represents the program statement and an edge $e \in E$ represents the control flow. Every function has a unique entry node and a unique exit node, with each call site being split into a call node and a return node. We use $outEdges(e)$ to represent the outgoing edges of node n. Also, $dst(e)$ denotes the destination node of edge e.

4.5.2. *Forward reachability analysis*

Algorithm 1 performs a forward reachability analysis to compute the data types of every source node $n_{src} \in SRC$, which is either **(i)** a program statement where permission scopes are requested, or **(ii)** an entry statement of a procedure containing Google Fit APIs as listed in Table 4, and the procedure which is invoked through inter-component communication (ICC). SNK denotes the set of program statements consuming Google-defined data types. Every element $n_{snk} \in SNK$ needs to satisfy two conditions: **(i)** n_{snk} is a Google Fit method call from the list in Table 4, and **(ii)** n_{snk} has a string parameter whose value is a data type in Table 2.

PGFit performs a forward traversal on the ICFG of the analysed program from every source node n_{src} to find its sinks (lines 2–29 in Algorithm 1). A standard worklist algorithm is applied to compute the nodes reachable from n_{src}. The worklist W contains a set of parameterised nodes (statements) with each element $(c,\ n_s)$ representing a context-sensitive statement, where c is a call stack $c = [c_1,\ c_2,\ \dots c_m]$ denoting a sequence of call sites from the entry of a program to the method containing n_s.

To achieve precise results for our reachability analysis, PGFit employs context-sensitive analysis to distinguish different program contexts by solving a balanced-parentheses problem (Reps et al., 1995) with calls and returns being matched (lines 11–22). The algorithm pushes the context-sensitive nodes which have not been visited previously into worklist (lines 23–25) for further processing until a fixed-point is reached.

Table 4
Google fit APIs.

Google Fit APIs	Public Method Summary
Google Fit History API	Result<Status>**deleteData**(GoogleApiClient client, DataDeleteRequest request) Result<Status>**insertData**(GoogleApiClient client, DataSet dataSet) Result<DailyTotalResult> **readDailyTotal**(GoogleApiClient client, DataType dataType) Result<DailyTotalResult> **readDailyTotalFromLocalDevice**(GoogleApiClient client, DataType dataType) Result<DataReadResult>**readData**(GoogleApiClient client, DataReadRequest request) Result<Status>**registerDataUpdateListener**(GoogleApiClient client, DataUpdateListenerRegistration request) Result<Status>**unregisterDataUpdateListener**(GoogleApiClient client, PendingIntent pendingIntent) Result<Status>**updateData**(GoogleApiClient client, DataUpdateRequest request)
Google Fit Recording API	Result<ListSubscriptionsResult> **listSubscriptions**(GoogleApiClient client) Result<ListSubscriptionsResult> **listSubscriptions**(GoogleApiClient client, DataType dataType) Result<Status>**subscribe**(GoogleApiClient client, DataType dataType) Result<Status>**subscribe**(GoogleApiClient client, DataSource dataSource) Result<Status>**unsubscribe**(GoogleApiClient client, DataSource dataSource) Result<Status>**unsubscribe**(GoogleApiClient client, DataType dataType) Result<Status>**unsubscribe**(GoogleApiClient client, Subscription subscription)
Google Fit Sensors API	Result<Status>**add**(GoogleApiClient client, SensorRequest request, OnDataPointListener listener) Result<Status>**add**(GoogleApiClient client, SensorRequest request, PendingIntent intent) Result<DataSourcesResult> **findDataSources**(GoogleApiClient client, DataSourcesRequest request) Result<Status>**remove**(GoogleApiClient client, OnDataPointListener listener) Result<Status>**remove**(GoogleApiClient client, PendingIntent pendingIntent)
Google Fit Sessions API	Result<Status>**insertSession**(GoogleApiClient client, SessionInsertRequest request) Result<SessionReadResult> **readSession**(GoogleApiClient client, SessionReadRequest request) Result<Status>**registerForSessions**(GoogleApiClient client, PendingIntent intent) Result<Status>**startSession**(GoogleApiClient client, Session session) Result<SessionStopResult> **stopSession**(GoogleApiClient client, String identifier) Result<Status>**unregisterForSessions**(GoogleApiClient client, PendingIntent intent)

4.5.3. *Backward analysis algorithm*

For every sink node n_{snk} on ICFG, we extract its Google-defined data types through Algorithm 2. It evaluates the values of the parameter *dtv* at n_{snk}, which is a Google API call (line 2). The value is extracted directly if *dtv* is a constant variable (lines 4–6), otherwise the algorithm starts backward data dependence analysis from the definition node of *dtv* on ICFG (lines 7–25), where *def*(*dtv*) returns the definition statement of *dtv* on the Static Single Assignment (SSA) form (Cytron et al., 1991) of the analysed program. Note that we only analyse the statements which are visited in the forward analysis (lines 11–13) since the data type extraction for every call path is context sensitive.

A standard worklist algorithm is applied for def-use analysis (Cytron et al., 1991) of variable *dtv* until a fixed point is reached (lines 9–24). Our def-use analysis considers three types of statement that define the value of *dtv*, i.e., assignment $dtv = var$ (line 15), function call $f(var)$; $f(dtv)\{\ldots\}$ (line 17), and function return $dtv = f(\ldots)$; $f(\ldots)\{\ldots$ *return-var*$\}$ (line 19). Note that loads and stores, which require pointer analysis, are handled conservatively (line 22). For each of these three cases, variable *var* is passed into *dtv*, which then is pushed into a worklist and analysed recursively if *var* is not a constant (lines 29–35). In most real apps, the values of data types usually flow via the above three types of statement.

4.6. *Over-privilege computation*

Algorithm 3 computes whether a source node n_{src} is over-privileged by comparing the two sets of data types D and D', where D is computed through ReachabilityAnalysis in Algorithm 1 and D' is the associated data types of n_{src} immediate available from Table 2. An over-privilege is reported if $D \cap D' = \varphi$ since an authorisation permission scope in D is not permitted in D'. For example, through Algorithm 1, if PGFit does not retrieve any data type (i.e., $D = \varphi$) for a permission scope whose corresponding data types are D' (obtained from Table 2), it will report an over-privilege warning because $D \cap D' = \varphi$. The time complexity of the algorithm is O(N^3), where N is the number of nodes on the ICFG (Reps, 1997).

5. Analysis results

We applied PGFit to a set of 20 fitness applications built upon Google Fit to identify the occurrence of privilege escalation regarding authorisation scopes. In cases of escalated privilege, fitness applications expose users to the risk of accepting unnecessary scope permissions in their apps. This exposes users to attacks where adversaries take advantage of escalated privilege to leak private user information.

5.1. *Dataset collection*

The input to PGFit are Google Fit-enabled Android applications. We used a publicly available tool and downloaded applications that are free and have no region restriction imposed by Google Play. In addition, since PGFit performs permission analysis by inspecting compiled class files of a fitness application, As of November 2017, there are 43 Google Fit-enabled applications available on Google Play Store. From this set of apps, we selected all the apps that are publicly available and not obfuscated as PGFit is not intended for obfuscated code. To this end, we used *dex2jar* in combination with JD tool (Java Decompiler, 2017) and discarded applications with string-obfuscated source codes. Overall, our dataset consisted of 20 Google Fit-enabled fitness applications.

5.2. *Result*

We have run PGFit on our dataset of 20 fitness applications. During the first phase PGFit discovers that 14 applications contained Google Fit API calls in one compiled class file, while in six applications Google Fit API calls and data types were distributed in more than one class. This shows that performing static analysis over all compiled class files of an application is unnecessary. For example, *Runtastic* is a health and fitness application to help users measure their activities (walking, running, jogging or biking) against goals they set. The APK file of the application contains 3507 compiled class files, while Google Fit API methods

Algorithm 1 Context-sensitive Forward Reachability Analysis.

```
Input  : $n_{src}$
Output: $D$ - A set of Google-defined Data Types

Procedure ReachabilityAnalysis($n_{src}$)
    $W \leftarrow W \cup \{(\emptyset, n_{src})\}$;
    $V \leftarrow \emptyset$;
    $D \leftarrow \emptyset$;
    while $(!W.empty())$ do
        $(c, n) \leftarrow W.pop()$;
        if $n \in SNK$ then
            $D \leftarrow D \cup$ ExtractDataTypes $(n_{snk})$
        end
        foreach $e \in outEdges(n)$ do
            if $e$ is a call edge with call site id $c_i$ then
                $c = c.push(c_i)$
            end
            else if $e$ is a return edge with call site id $c_i$
             then
                if $c.peek(c_i)$ then
                    $c = c.pop()$
                end
                else
                    continue;
                end
            end
            $n_s = dst(e)$;
            if $(c, n_s) \notin V$ then
                $V \leftarrow V \cup (c, n_s)$;
                $W.push(c, n_s)$
            end
        end
    end
    Return $D$;
end
```

and procedures are being employed among 3 class files.

Meanwhile, PGFit reports the fitness services that an application requested to connect to. The statistics of the extracted connection request to Google APIs are presented in Table 5. As shown in the table, History API and Sessions API are at the top of the list of most prevalent in the set of 20 apps evaluated. The former enables an application to access the fitness data history that was inserted or recorded using other applications or itself. The latter provides a functionality to create sessions when a user performs a fitness activity. The Config API is another popular fitness service, employed mainly to disconnect from Google Fit. Other fitness services, which provide functionality to store fitness data, are the Recording API and Sensors API. However, these fitness services are less prevalent, meaning most fitness applications read fitness data from Google Fit rather than writing the fitness data to Google Fit, which is against Google Fit principles.

Out of 20 fitness applications in our dataset, PGFit found six applications (30%) that request at least one authorisation scope but never use any data types corresponding to that scope. This is undesirable, because it allows an adversary to abuse this vulnerability to leak sensitive information from a victim's fitness data. Table 6 reports the unnecessary permissions among these six applications. For example, five (83%) out of 20 apps grant permission scope SCOPE_ACTIVITY_READ_WRITE, but the permission is never used in these apps. One example of Google Fit-enabled applications that exhibits over-privilege permission is *8fit*; a personal trainer providing workout routines and healthy meal plans tailored to a user. This application requests access to three authorisation scopes but does not use any data type related to any of the requested scopes.

For comparison, we implemented a naive approach by searching for Google API method invocations that consume Google-define data types directly without considering the reachability property of program flows via call graph traversals introduced in Section 4.5. To validate the warnings, we used JD tool to decompile the Google Fit-related compiled class files in the 20 apps. We then manually investigated the reconstructed source codes and compared the results obtained from PGFit and the naive approach with reverse-engineered codes. We did not find any inconsistencies between the report of PGFit and the manual analysis of source codes, while the naive approach produced two *false alarms*, which represents 10% overall false alarm rates. A careful inspection discovered that the apps requested unnecessary scopes, and the developers of the apps used *return* statements before the Google API method invocations that made them unrealisable. Both cases were discovered successfully by PGFit since it analysed control and data dependences of an app context-sensitively. By identifying these area of dead codes, spurious value-flows are eliminated. The earlier version of PGFit (Nobakht et al., 2018) reported that seven of the 20 Google-defined apps (35%) have unnecessary permissions. Our manual inspections determined that one app was not actually over-privileged, resulting in a 14% false positive rate. The false alarm in Nobakht et al. (2018) is due to the conservative analysis without considering context-sensitivity and ICC calls via user interaction events. The improvement proposed in this paper successfully reduces this false alarm.

In Android 6.0 (Marshmallow), Google redesigned its long-criticised permission model to prompt the user during runtime, allowing them to dynamically revoke granted permissions. Prior to this update, Android systems implemented static permissions, requested upon app installation mechanism to request permission. The updated runtime permission model of Android enhances security related to app permissions. However, as shown in Gasparis et al. (2018) most apps either have not yet migrated to this new model because it cannot be enforced due to potential backwards compatibility issues or do not follow the recommended guidelines. From an end-user perspective, these changes are rendered ineffective by the fact that most of mobile users do not have sufficient expertise to weigh the permissions requested against those that are nec-

Algorithm 2 Backward Reachability Analysis to Extract Data Types.

```
Input  : $n_{snk}$ - A Google Fit method using a Google
         Fit Data type as argument
Output: $d$ - Data type

Procedure ExtractDataTypes($c, n_{snk}$)
    Let $dtv$ be the variable at sink node ($n_{snk}$), a
      Google Fit API, which consumes a data type;
    Let $\sigma$ be a function evaluating the value of $dtv$;
    if $\sigma(dtv)$ is constant then
        $D \leftarrow \sigma(dtv)$;
    end
    else
        $W \leftarrow W \cup \{c, def(dtv)\}$;
        while $(!W.empty())$ do
            $(c, n_s) \leftarrow W.pop()$;
            if $(c, n_s) \notin V$ then
                continue;
                // only nodes visited in the forward
                   analysis are handled
            end
            switch $n_s$ do
                case Assignment: $dtv = var$
                  PushToWorkList($var$);
                break;
                case Function call: $f(var)$;
                  $f(dtv)\{\dots\}$
                  PushToWorkList($var$);
                break;
                case Function return: $dtv = f(\dots)$;
                  $f(\dots)\{\dots$ return var$\}$ ;
                PushToWorkList($var$);
                break;
                default:
                // stores and loads are handled
                   conservatively
            end
        end
    end
    Return $D$;
end

Procedure PushToWorkList($var$)
    if $\sigma(var)$ is const then
        $D \leftarrow D \cup \sigma(var)$;
    end
    else
        $W \leftarrow W.push(var)$;
    end
end
```

Algorithm 3 Overprivilege Computation.

```
Input  : $n_{src}$
Output: Over-privilege reports for each source node

Procedure Over-priviledgeComputation($n_{src}$)
    foreach $n_{src} \in SRC$ do
        $D \leftarrow$ ReachabilityAnalysis($n_{src}$) ;
        Let $D'$ be a set of Google-defined data types
          of $n_{src}$ in Table 2;
        if $D \cap D' = \emptyset$ then
            report over-privilege $n_{src}$
        end
    end
end
```

Table 5
Statistics of Connection request to Fitness Services in a set of 20 Google Fit-enabled Applications.

Fitness Service	# of Apps
HISTORY_API	14 (70%)
SESSIONS_API	14 (70%)
CONFIG_API	12 (60%)
RECORDING_API	8 (40%)
SENSORS_API	5 (25%)

Table 6
Unnecessary scope permissions in 20 apps.

Authorisation Scope	# of Apps
SCOPE_ACTIVITY_READ_WRITE	5 (83%)
SCOPE_BODY_READ_WRITE	4 (66%)
SCOPE_LOCATION_READ_WRITE	2 (33%)
SCOPE_BODY_READ	1 (16%)

essary for the core functionality of the app. The new permission model of Android limits over-privileged apps yet still makes it possible for an app to obtain escalated privilege to app permissions.

6. Related work

There are two lines of research most closely related to ours: IoT security and least-privilege principle.

IoT Security. In recent years, much research has been conducted in the context of IoT security. Research in this domain mainly focused on three aspects: *Protocols*, *Devices* and *Platforms*. In IoT protocol security, researchers warned how security flaws in IoT-specific protocols such as ZigBee (Jun, 2017) and Zwave (Behrang Fouladi, 2017) make devices vulnerable to compromise.

In IoT device security, Ronen et al. classified attacks on IoT devices based on how the attacker deviates from the designed functionality to achieve a different effect and demonstrated potential attacks on smart lighting systems (Ronen and Shamir, 2016). In Nobakht et al. (2016), authors proposed a network-level solution using Software-defined Networking (SDN) to monitor smart home IoT network traffic and machine learning detection mechanisms to identify malicious activities. In another attempt, authors in Nobakht et al. (2019) proposed a security mechanism, called IoT-NetSec, to ensure IoT device availability requirement is met. IoT-NetSec enables network operators to ensure network service attacks such as heavy hitters are not occurring against IoT devices and also these devices are not being compromised to perform such attacks.

Research into IoT platform security is still in its early stages. There has been effort in analysing security concerns on programming frameworks, in particular hub-based platforms (Fernandes et al., 2016a; Jia et al., 2017). More recently Fernandes et al. (2016b) performed analysis of the Samsung-owned SmartThings programming framework for smart home applications. Although SmartThings is a closed systems and third-party applications are run on a proprietary cloud backend, the authors managed to access the source code of applications and discovered security design flaws in SmartThings platforms and other common vulnerabilities such as revealing sensitive information caused by the lack of sufficient protection on protocols operating between the cloud backend and the client-side hub. In comparison, our work can be applied on both cloud-based and hub-based configuration and is not limited to closed source applications. Instead, this paper focused on analysing the mechanism operating between IoT programming framework backends and the IoT devices.

Least-privilege Principle. Limiting applications privilege can lower potential security and privacy risks, however, there is a trade-off between the complexity of the permission control model and enforcing least-privilege. This is evidenced in a large body of prior research work (Enck et al., 2009; Rahmati and Madhyastha, 2015; Felt et al., 2012a, 2012b; Roesner and Kohno, 2013; Roesner et al., 2012).

The popularity and open-source nature of Android have attracted a large number of research work to evaluate and propose enhancements to the security of Android OS itself, apps built on top of Android and Software Development Kit (SDK) tools to build such apps. In a recent attempt, Acar et al. have systematically categorised research related to Android security and privacy (Acar et al., 2016). Android employs the concept of permission-based access control for privileged resources. This area has received a lot of attention by the security research community with different motivations ranging from preventing unauthorised information disclosure (Jin et al., 2018; Wei et al., 2018; Nadkarni et al., 2016; Jia et al., 2013; Arzt et al., 2014) to detecting over-privileged apps (Vidas et al., 2011). Additionally, a large body of research focuses on the modification Android's permission enforcement (Felt et al., 2011a; Nauman et al., 2010; Fragkaki et al., 2012; Dietz et al., 2011).

Recently, a number of static analysis tools for analysing Android OS have been proposed and implemented (Felt et al., 2011b; Au et al., 2012) with the primary focus of creating a permission map for Android OS. Felt et al. proposed Stowaway (Felt et al., 2011b) which used unit testing and feedback directed API to observe the required permissions for each API call. They determined that about one third of Android apps in a set of 940 apps were over-privileged. In another attempt, the PScout tool proposed by Wain Yee Au et al. (2012), extended Stowaway and used static reachability analysis between permission checks and API calls to extract permission specifications from Android OS source code. More recently, Backes et al. built the AXPLORER (Backes et al., 2016) tool to conduct an Android permission analysis that establishes a more precise permission mapping. To evaluate the precision and performance of such application analysis tools, Bonett et al. examined a set of prominent Android static analysis tools that have been proposed in response to malicious, curious or vulnerable apps (Bonett et al., 2018).

Our work is similarly motivated, however, these previous investigations analysed permissions granted to third-party applications to access hardware and software resources on physical devices. In contrast to prior work, we focus on permissions granted by the user to applications to access their private data through health and fitness apps.

Another line of research focuses on the over-privilege permission in IoT programming frameworks. Fernandes et al. (2016b) performed a market-scale over-privilege analysis of third-party applications on SmartThings and discovered that over 55% of applications are over-privileged, citing the framework design flaw as the main factor of the problem. In contrast, our work could be applied to many IoT programming platforms using public cloud services with open standards and is not limited to proprietary systems.

7. Conclusions

Emerging IoT programming frameworks for health and fitness applications provide tangible benefits to users. Typically, health and activity related information in such frameworks is collected from various sources, unnecessary details are abstracted away using data analytic techniques, and finally the user or authorised third-party applications can access data from one central location.

The confidentiality of user data in such platforms is critical as leaked sensitive user data poses security and privacy risks to users and could results in financial and psychological harms. Over-privileged apps on these platforms request end-users for permissions which are not necessarily required for the core functionality of the apps. Over-privileged apps might not be harmful to users and of themselves, however, they are prime targets for abuse by other security vulnerabilities. Over-privileged apps make it possible for adversaries to abuse escalated privileges and leak user data for malicious purposes. By identifying over-privileges, we can reduce the attack surface of apps using IoT programming frameworks to ensure the confidentiality of user data stored on such platforms.

This paper aims to address an issue that affects the security and privacy of user data in IoT programming frameworks. To demonstrate the concept, we have examined Google Fit, a popular IoT programming framework, to determine how well Google Fit-enabled third-party apps adhere to the least-privilege principle when requesting user consent to access sensitive data. We first studied Google Fit and its permission control mechanism to gain insights into its structure and key requirements.

However, permission analysis of Google Fit-enabled third-party IoT apps is challenging due to the closed-source system, the sheer size of compiled class files and the complications of API usage in them. To overcome these challenges, we have developed a static permission analysis tool, called PGFIT, to ensure third-party apps on top of Google Fit adhere to the least-privilege principle. PGFIT takes a given Google Fit-enabled app as input and extracts the set of authorisation scopes along with the used Google-defined data types in the app, by performing forward and backward reachability analysis to compute over-privilege. We applied PGFIT to 20 Android Google Fit-enabled apps that were available on the market at the time of the study. Results from this study showed that 30% of those Google Fit-enabled apps are over-privileged and requested at least one unnecessary authorisation scope to access user data. This problem in Google Fit stems from its coarse-grained permission access control. PGFit, thus, guides third-party app developers to assure their apps do not open an attack vector putting user at risk of leaking sensitive data and also helps app users to check the privacy requirement of app at install time.

Declaration of competing interest

There is no conflict of interest among editorial board.

Acknowledgments

We would like to thank the anonymous reviwers for helpful feedback. We are grateful to Andrew Noble for providing feedback on earlier drafts.

References

Acar, Y., Backes, M., Bugiel, S., Fahl, S., McDaniel, P., Smith, M., 2016. SoK: lessons learned from android security research for appified software platforms. In: 2016 IEEE Symposium on Security and Privacy (SP), pp. 433–451, https://doi.org/10.1109/SP.2016.33.
AIA, November 2017. AIA Vitality. https://www.aiavitality.com.au/vmp-au/.
apktool, November 2017. https://ibotpeaches.github.io/Apktool/.
Apple, November 2017. HealthKit. https://developer.apple.com/healthkit/.
Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., McDaniel, P., 2014. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14. ACM, New York, NY, USA, pp. 259–269, https://doi.org/10.1145/2594291.2594299.
Assure, Q., November 2017. Qantas Assure. https://www.qantasassure.com/health-insurance/offer.
Au, K.W.Y., Zhou, Y.F., Huang, Z., Lie, D., 2012. PScout: analyzing the android permission specification. In: CCS '12, pp. 217–228.
Backes, M., Bugiel, S., Derr, E., McDaniel, P., Octeau, D., Weisgerber, S., 2016. On demystifying the android application framework: Re-visiting android permission specification analysis. In: USENIX Security '16, pp. 1101–1118.
Behrang Fouladi, S.G., November 2017. Security Evaluation of the Z-Wave Wireless Protocol. https://sensepost.com/cms/resources/conferences/2013/bh_zwave/Security20Evaluation20of20Z-Wave_WP.pdf.
Bonett, R., Kafle, K., Moran, K., Nadkarni, A., Poshyvanyk, D., 2018. Discovering flaws in security-focused static analysis tools for android using systematic mutation. In: Proceedings of the 27th USENIX Conference on Security Symposium, SEC'18. USENIX Association, Berkeley, CA, USA, pp. 1263–1280. http://dl.acm.org/citation.cfm?id3277203.3277298.
Bruneton, E., Lenglet, R., Coupaye, T., 2002. ASM: a code manipulation tool to implement adaptable systems. Adaptable Extensible Compon. Syst. 30 (19).
Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K., 1991. Efficiently computing static single assignment form and the control dependence graph. TOPLAS '91 13 (4), 451–490.
Denning, T., Kohno, T., Levy, H.M., 2013. Computer security and the modern home. Commun. ACM 56 (1), 94–103.
dex2jar, November 2017. https://github.com/pxb1988/dex2jar.
Dietz, M., Shekhar, S., Pisetsky, Y., Shu, A., Wallach, D.S., 2011. Quire: lightweight provenance for smart phone operating systems. In: Proceedings of the 20th USENIX Conference on Security, SEC'11. USENIX Association, Berkeley, CA, USA, p. 23. http://dl.acm.org/citation.cfm?id2028067.2028090.
Enck, W., Ongtang, M., McDaniel, P., 2009. On lightweight mobile phone application certification. In: Proceedings of the 16th ACM Conference on Computer and Communications Security, CCS '09. ACM, New York, NY, USA, pp. 235–245, https://doi.org/10.1145/1653662.1653691.
Felt, A.P., Wang, H.J., Moshchuk, A., Hanna, S., Chin, E., 2011a. Permission Re-delegation: attacks and defenses. In: Proceedings of the 20th USENIX Conference on Security, SEC'11. USENIX Association, Berkeley, CA, USA, p. 22. http://dl.acm.org/citation.cfm?id2028067.2028089.
Felt, A.P., Chin, E., Hanna, S., Song, D., Wagner, D., 2011b. Android permissions demystified. In: CCS '11, pp. 627–638.
Felt, A.P., Egelman, S., Wagner, D., 2012a. I've got 99 problems, but vibration Ain'T one: a survey of smartphone users' concerns. In: Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices, SPSM '12. ACM, New York, NY, USA, pp. 33–44, https://doi.org/10.1145/2381934.2381943.
Felt, A.P., Egelman, S., Finifter, M., Akhawe, D., Wagner, D., 2012. How to ask for permission. In: HotSec'12, p. 7.
Fernandes, E., Paupore, J., Rahmati, A., Simionato, D., Conti, M., Prakash, A., 2016. FlowFence: practical data protection for emerging IoT application frameworks. In: USENIX Security '16, pp. 531–548.
Fernandes, E., Jung, J., Prakash, A., 2016. Security analysis of emerging smart home applications. In: S&P '16, pp. 636–654, https://doi.org/10.1109/SP.2016.44.
Fragkaki, E., Bauer, L., Jia, L., Swasey, D., 2012. Modeling and enhancing android's permission system. In: Foresti, S., Yung, M., Martinelli, F. (Eds.), Computer Security ESORICS 2012. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 1–18.
Gasparis, I., Aqil, A., Qian, Z., Song, C., Krishnamurthy, S.V., Gupta, R., Colbert, E., 2018. Droid m: developer support for imbibing android's new permission model. In: Proceedings of the 2018 on Asia Conference on Computer and Communications Security, ASIACCS '18. ACM, New York, NY, USA, pp. 765–776, https://doi.org/10.1145/3196494.3196533.
Google, November 2017a. Google Fit. https://www.google.com/fit/.
Google, November 2017b. Apps Work with Google Fit. https://play.google.com/store/apps/collection/promotion_3000e6f_googlefit_all.
Google, November 2017c. Google Fit Developer Reference. https://developers.google.com/android/reference/com/google/android/gms/fitness/Fitness.
Google, November 2017d. Google Developer Reference. https://developers.google.com/android/reference/com/google/android/gms/common/package-summary.
Java Decompiler, November 2017. http://jd.benow.ca/.
Jia, L., Aljuraidan, J., Fragkaki, E., Bauer, L., Stroucken, M., Fukushima, K., Kiyomoto, S., Miyake, Y., 2013. Run-time enforcement of information-flow properties on android. In: Crampton, J., Jajodia, S., Mayes, K. (Eds.), Computer Security ESORICS 2013. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 775–792.
Jia, Y.J., Chen, Q.A., Wang, S., Rahmati, A., Fernandes, E., Mao, Z.M., Prakash, A., 2017. ContexIoT: towards Providing Contextual Integrity to Appified IoT Platforms. San Diego, CA. .
Jin, H., Liu, M., Dodhia, K., Li, Y., Srivastava, G., Fredrikson, M., Agarwal, Y., Hong, J.I., 2018. Why are they collecting my data?: inferring the purposes of network traffic in mobile apps. Proc. ACM Interact., Mob., Wearable Ubiquitous Technol. 2 (4), 173:1–173:27, https://doi.org/10.1145/3287051.
Jun, L., November 2017. I'm A Newbie yet I Can Hack ZigBee - Take Unauthorized Control over ZigBee Devices. https://www.defcon.org/html/defcon-23/dc-23-speakers.html#Li.
Medibank, November 2017. Medibank & Flybuys. https://flybuys.medibank.com.au/.
Microsoft, November 2017. HealthVault. https://www.healthvault.com.
Nadkarni, A., Andow, B., Enck, W., Jha, S., 2016. Practical DIFC enforcement on android. In: 25th USENIX Security Symposium (USENIX Security 16). USENIX Association, Austin, TX, pp. 1119–1136. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/nadkarni.
Nauman, M., Khan, S., Zhang, X., 2010. Apex: extending android permission model and enforcement with user-defined runtime constraints. In: Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security, ASIACCS '10. ACM, New York, NY, USA, pp. 328–332, https://doi.org/10.1145/1755688.1755732.
Nobakht, M., Sivaraman, V., Boreli, R., 2016. A host-based intrusion detection and mitigation framework for smart home IoT using OpenFlow. In: 2016 11th International Conference on Availability, Reliability and Security, IEEE ARES '16, pp. 147–156, https://doi.org/10.1109/ARES.2016.64.
Nobakht, M., Sui, Y., Seneviratne, A., Hu, W., 2018. Permission analysis of health and fitness apps in IoT programming frameworks. In: 2018 17th IEEE International Conference on Trust, Security and Privacy in Computing and Communications/12th IEEE International Conference on Big Data Science and Engineering (TrustCom/BigDataSE), IEEE TrustCom '18, pp. 533–538, https://doi.org/10.1109/TrustCom/BigDataSE.2018.00081.
Nobakht, M., Russell, C., Hu, W., Seneviratne, A., 2019. IoT-NetSec: policy-based IoT network security using OpenFlow. In: 2019 IEEE International Conference on Pervasive Computing and Communications Workshops, IEEE PerCom Workshops '19, pp. 955–960, https://doi.org/10.1109/PERCOMW.2019.8730724.

Oracle, November 2017. Java Virtual Machine Specification. https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-2.html.

Palsberg, J., Jay, C.B., 1998. The essence of the visitor pattern. In: COMPSAC '98, pp. 9–15.

Rahmati, A., Madhyastha, H.V., 2015. Context-specific access control: conforming permissions with user expectations. In: Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices, SPSM '15. ACM, New York, NY, USA, pp. 75–80, https://doi.org/10.1145/2808117.2808121.

Reps, T., 1997. Program analysis via graph reachability. In: ILPS '97, pp. 5–19.

Reps, T., Horwitz, S., Sagiv, M., 1995. Precise interprocedural dataflow analysis via graph reachability. In: POPL '95, pp. 49–61.

Roesner, F., Kohno, T., 2013. Securing embedded user interfaces: android and beyond. In: Presented as Part of the 22nd USENIX Security Symposium, USENIX Security '13. USENIX, Washington, D.C., pp. 97–112. https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/roesner.

Roesner, F., Kohno, T., Moshchuk, A., Parno, B., Wang, H.J., Cowan, C., 2012. User-driven access control: rethinking permission granting in modern operating systems. In: 2012 IEEE Symposium on Security and Privacy, IEEE S&P '12, pp. 224–238, https://doi.org/10.1109/SP.2012.24.

Ronen, E., Shamir, A., 2016. Extended functionality attacks on IoT devices: the case of smart lights. In: EuroS&P '16, pp. 3–12.

Samsung, November 2017. Samsung Digital Health. http://developer.samsung.com/health.

The oauth 1.0 protocol, April 2010. RFC 5849, Internet Engineering Task Force (IETF). https://tools.ietf.org/html/rfc5849.

The oauth 2.0 protocol, October 2012. RFC 6749, Internet Engineering Task Force (IETF). https://tools.ietf.org/html/rfc6749.

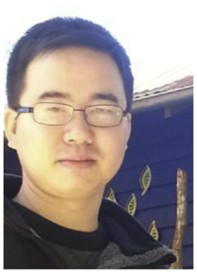
Valle-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., Sundaresan, V., 1999. Soot - a Java bytecode optimization framework. In: CASCON '99, CASCON '99. IBM Press.

Vidas, T., Christin, N., Cranor, L.F., 2011. Curbing android permission creep. In: Proceedings of the Workshop on Web 2.0 Security and Privacy, W2SP '11.

Watson, T.J., November 2017. Libraries for Analysis (WALA). http://wala.sf.net/.

Wei, F., Roy, S., Ou, X., Robby, 2018. Amandroid: a precise and general inter-component data flow analysis framework for security vetting of android apps. ACM Trans. Priv. Secur. (TOPS) 21 (3), 14:1–14:32, https://doi.org/10.1145/3183575.

Mehdi Nobakht is currently a Postdoctoral Research Fellow in the School of Engineering and Information Technology (SEIT) at University of New South Wales (UNSW) Canberra Australia, where he is affiliated with UNSW Canberra Cyber Research Centre. His research centres around Cyber-physical Systems/Internet of Things (IoT) Security and Computer Networks; spans from Adversarial Machine Learning and Data Security to adopting Software Defined-networking (SDN) and Distributed Ledger Technology (DLT) to address security and privacy concerns in IoT systems. Mehdi received his Ph.D. degree in Computer Science and Engineering (CSE) at UNSW Sydney, Australia in 2018 and his M.Sc. degree in Information Technology specialising in Electronics and Communication Systems at the University of Turku, Finland in 2013.



Yulei Sui is a Lecturer (Assistant Professor) and an ARC DECRA at Faculty of Engineering and Information Technology, University of Technology Sydney (UTS). He obtained his Ph.D from University of New South Wales (UNSW), where he also holds an Adjunct Lecturer position. He is broadly interested in the research field of software engineering and programming languages, particularly interested in static and dynamic program analysis for software bug detection and compiler optimizations. He worked as a software engineer in Program Analysis Group for Memory Safe C project in Oracle Lab Australia. He was an Australian IPRS scholarship holder, a keynote speaker at EuroLLVM and a Best Paper Award winner at CGO, and has been awarded an Australian Discovery Early Career Researcher Award (DECRA) 2017–2019.

Aruna Senviratne is a Professor at the School of Electrical Engineering and Telecommunication (EET), UNSW Sydney. He received the PhD degree in electrical engineering from the University of Bath, United Kingdom. He is the foundation chair in telecommunications and holds the Mahanakorn chair of telecommunications in EET. He was the research director of the Cyber Physical Systems Research Program, Data61, CSIRO. His current research interests include mobile content distributions and preservation of privacy. He has held academic appointments with the University of Bradford, United Kingdom, Curtin University, and UTS.

Wen Hu is an Associate Professor at the School of Computer Science and Engineering (CSE) at UNSW Sydney. Much of his research career has focused on the novel applications, low-power communications, security and compressive sensing in sensor network systems and Internet of Things (IoT). Hu published regularly in the top rated sensor network and mobile computing venues such as ACM/IEEE IPSN, ACM SenSys, ACM transactions on Sensor Networks (TOSN), IEEE Transactions on Mobile Computing (TMC), and Proceedings of the IEEE. Hu received his Ph.D. from UNSW Sydney. He is a recipient of multiple research grants from Australian Research Council, CSIRO and industries. He is a senior member of ACM and IEEE, and is an associate editor of ACM TOSN, as well as serves on technical advisory board (IoT) of ACS, Standards Australia and the organising and program committees of networking conferences including ACM/IEEE IPSN, ACM SenSys, ACM MobiCOM, ACM MobiSys, ACM/IEEE IOTDI, IEEE ICDCS, IEEE LCN, IEEE ICC, IEEE GlobeCom. Hu works as the Chief Scientist (part time) in WBS Tech to commercialise his research results in smart buildings and IoT since 2017.