DELTA: Data Extraction and Logging Tool for Android

Riccardo Spolaor, *Student Member, IEEE*, Elia Dal Santo, and Mauro Conti, *Senior Member, IEEE*

Abstract—In recent years, the use of smartphones has increased exponentially, and so have their capabilities. Together with an increase in processing power, smartphones are now equipped with a variety of sensors and provide an extensive set of API. These capabilities allow us to extract data related to environment, user habits, and operating system itself. This data is extremely valuable in many research fields such as user authentication, intrusion, and information leaks detection. For these reasons, researchers need a solid and reliable logging tool to collect data from mobile devices. In this paper, we first survey the existing logging tools available on the Android platform, comparing their features and their impact on the system. Then, we present DELTA - Data Extraction and Logging Tool for Android, which improves the existing Android logging solutions in terms of flexibility, fine-grained tuning capabilities, extensibility, and available set of logging features. We fully implement DELTA and we run a thorough performance evaluation. The results show that our tool has a low impact on the performance of the system, on battery consumption, and on user experience. Finally, we make the DELTA source code available to the research community.

Index Terms—Android, logging tool, data extraction, pervasive computing

1 Introduction

Thanks to their widespread adoption [38] and extensive data collection capabilities [32], modern mobile devices, such as smartphones and tables, became very useful research tools [19], [35]. The flexibility and complexity of mobile Operating Systems (OS) are also increasing very quickly. From simple embedded firmwares of early feature phones, modern mobile OSs now match the capabilities of traditional desktop systems. Mobile OSs such as Google's Android or Apple's iOS now support full-fledged multi-tasking, extensive APIs (Accessible Programming Interfaces), advanced connectivity, and a wide variety of hardware.

At the same time, mobile devices are equipped with a wide variety of sensors, which collect data from the surrounding environment. Built-in sensors such as gyroscopes, accelerometers, GPS receivers and digital compasses allow a mobile device to know its orientation, speed, and position. Similarly, light, pressure and temperature sensors can monitor the physical environment around the mobile device and apply on the fly corrections to system settings, such as adapting the display brightness to match the ambient one. As another example, built-in microphones and cameras collect audio and visual feedback. Nowadays, all the aforementioned components (and more) are commonly included in modern mobile devices, usually even in low and mid-range ones.

The above mentioned hardware and software characteristics make modern mobile devices excellent data gathering devices for research purposes. In fact, they allow us to explore whole new areas of research, spanning multiple fields [17], [18], [23], [26]. Another interesting avenue of research is using smartphones as portable monitoring stations, able to perform a variety of background monitoring tasks. Examples include collecting readings from personal health sensors [36], recording ambient data like air and sound pressure, monitoring and tracing people movements and habits [29]. Software development can also benefit from such data, from simply collecting logs to monitor apps performance and crashes [16], to analyzing user patterns and behavior [29] in order to perform detailed usability tests.

We categorized the types of data we can extract from mobile devices into three main categories:

- *Sensor data* is data we can gather directly by querying the many sensors embedded in modern mobile devices. This includes a wide variety of information about the device itself and its surrounding environment (e.g., the device's position, orientation and relative speed).
- *Device/OS context data* is the state of the device itself and its operating system (e.g., battery level, list of running processes, traffic statistics, and file system activity).
- *User interaction data* is related to the device's user and her actions and habits, such as how she interacts with the touchscreen, with the keyboard and with elements of the User Interface.

Given the high value of this data for research, a powerful and flexible multi-purpose logging tool would be of extreme value, as it would enables researchers to make data gathering for their projects easier, effective and efficient.

Contribution - The contribution of our paper is threefold:

1) We present a survey on the existing logging tools for mobile devices. In particular, we select the more

- *The authors are with the Department of Mathematics "Tulio Levi-Civita", SPRITZ Security and Privacy Research Group, Human Inspired Technology (HIT) Centre, University of Padua, Padova, PD 35122, Italy. E-mail: {rspolaor, conti}@math.unipd.it, elia.dalsanto@studenti.unipd.it.*

Manuscript received 29 Nov. 2016; revised 5 Sept. 2017; accepted 2 Oct. 2017. Date of publication 13 Oct. 2017; date of current version 3 May 2018. (Corresponding author: Riccardo Spolaor.)
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TMC.2017.2762692

prominent tools present in the literature and we thoroughly compare their data collection features and architecture, highlighting both their selling points and limitations.

2) We present DELTA - Data Extraction and Logging Tool for Android,[1] our own logging solution for the Android platform. We designed and implemented DELTA to cover the shortcomings most commonly found in other tools. Our tool logs as many information sources as possible, while at the same time allowing flexibility in what data is logged and at which frequency. Moreover, DELTA's architecture is designed to be modular and as non-invasive as possible in regards to user privacy and system security.
3) We evaluate DELTA's performance both in terms of security and energy consumption. We discuss in detail the security best practices we adopted against well known attacks in the literature. We run a thorough evaluation of DELTA's battery consumption within a broad range of settings, giving as references of comparison the battery consumption of several activities users carry out in daily life (e.g., music and video playing). Moreover, we perform an extensive comparison between the battery consumption of DELTA and some other competitors.

While DELTA does not introduce novel techniques in the way data is collected from the Android operating system, we believe that the real value of our tool is the way it makes easy for the user to collect and organize logs from many sources without having to write his own implementation.

We make the DELTA source code and toolset available to the research community and practitioners, so that interested people can leverage it to streamline the process of logging data for their experiments.

Organization - The rest of this paper is organized as follows. In Section 2, we present a thorough survey on the existing data collection tools for mobile devices. In Section 3 we introduce some basic concepts about the Android operating system and how its internal workings shaped the way we implemented our solution. In Section 4 we go in-depth about the design and inner workings of the DELTA system, while in Section 5 we sum up our results in terms of achieved goals, use cases, the security impact and energy requirements of our solution. Finally, in Section 6, we draw conclusions and introduce some ideas for future works.

2 Mobile Data Collection Tools

In this section, we open with a survey on existing mobile logging tools and other related work in the literature (Sections 2.1 and 2.2). We then sum up the main limitations common to existing tools (Section 2.3). Finally, we explain how our proposal outperforms existing solutions (Section 2.4).

2.1 Existing Tools

While several logging tools exist that are aimed at a mainstream public (e.g., mSpy,[2] MobiStealth),[3] they are generally unsuitable for research purposes. Indeed, such tools are designed and marketed as "spy apps" and they do not provide the precision and customization required for a research project. In addition to that, these apps are typically designed to send data to third-party remote servers, compromising user privacy and blocking access to the collected data behind a pay-wall. For these reasons, we did not consider any "spy apps" in this survey. Our solution is explicitly aimed at researchers, and we want it to be as open and easily accessible as possible.

Data collection is also an important aspect in the field of forensics. However, in forensic analysis the aim is to extract relevant information from a device at a certain point in time, usually in the context of a law enforcement operation. This is why forensic tools for mobile devices (e.g., ADEL [24]) are designed to perform a one-time data extraction, rather than to continuously monitor a device. This approach is very limiting for research purposes, where researchers typically want to collect usage data over time in order to find correlations and make predictions. In fact, a one-time extraction cannot provide a history of sensor readings and system events because it is usually more focused at making a snapshot of the current state of a device.

In our comparison, we focus on tools that can do continuous and/or periodic logging of data from more than a single source or sensor: SystemSens [20], DroidWatch [27], MobileSens [28], LiveLab (iOS) [37], PhoneLab [34] and DeviceAnalyzer [44]. Table 1 summarizes a feature comparison between the tools we tested and our solution. We grouped the logging features in Table 1 to highlight the various contexts from which we can gather data from (e.g., sensor readings, screen interactions, network features). In the following paragraphs, we analyze each of these tools in detail.

SystemSens is an open source logging application presented by Falaki et al. in [20] that collects data from sensors and OS context, and it is designed to offer some extensibility options. However, *SystemSens* has some shortcomings that limit its usefulness for research purposes. First, it has a monolithic design, with extensibility provided via Android Interface Definition Language (AIDL) [1]. as an optional feature. Second, *SystemSens* uses a fixed global polling interval, set at a frequency of two minutes, which is often not sufficient for fine grained data analysis (more on these issues in Section 2.3).

Another similar tool is *DroidWatch* [27], an enterprise monitoring system for Android mobile devices. *DroidWatch* focuses on data sources such as phone call logs, visited websites, text messages and the user location. On the other hand, it ignores system information and sensor readings. *DroidWatch* is also not natively extensible, and suffers from the same limitations as *SystemSens*: no fine-tuning or advanced customization of logging behavior and a monolithic design.

MobileSens [28] is another app for logging user behavior in Android. The main aim of this tool is to profile user actions in order to study their behavior. Thus, logging is geared toward user actions and how they affect the state of the device (e.g., when the screen turns on/off, when the user sends messages). Like *DroidWatch*, *MobileSens* is not extensible and no source code is provided.

Developed by Wagner et al. at the University of Cambridge, *DeviceAnalyzer* [44] is a comprehensive logging tool for Android. Among the Android tools we analyzed,

1. DELTA is open source and it is available at `http://spritz.math.unipd.it/projects/deltaloggingtool/`
2. mSpy - https://www.mspy.com
3. MobiStealth - http://www.mobistealth.com

TABLE 1
Feature Comparison Table

Group	Logged features	SystemSens [20]	DroidWatch [27]	MobileSens [28]	LiveLab (iOS) [37]	PhoneLab [34]	DeviceAnalyzer [44]	DELTA (our proposal)
Sensors	Gravity sensor	×	×	×	×	×	✓	✓
	Accelerometer sensor	×	×	×	✓	×	✓	✓
	Magnetic field sensor	×	×	×	×	×	✓	✓
	Proximity sensor	×	×	×	×	×	×	✓
	Pressure sensor	×	×	×	×	×	×	✓
	Light sensor	×	×	×	×	×	×	✓
	Humidity sensor	×	×	×	×	×	×	✓
	Log noise level around device	×	×	×	×	×	×	✓
	Record from microphone	×	×	×	×	×	×	✓
Screen	Screen state (off/on/unlock)	×	✓	✓	×	✓	✓	✓
	Touch events logging	×	×	×	×	×	×	✓
	Keyboard state (open/close)	×	×	×	×	×	×	✓
	Keylogging	×	×	×	×	×	×	✓
System & Power	CPU statistics	✓	×	×	×	×	×	✓
	Battery statistics	✓	×	×	✓	×	✓	✓
	Battery charging status	✓	×	✓	×	✓	✓	✓
	Memory statistics	✓	×	×	×	×	×	✓
	System volume change	×	×	×	×	×	×	✓
	Date / Time / Timezone changes	×	×	✓	×	×	✓	✓
	Device turning on/off	×	×	✓	×	×	✓	✓
	Storage space monitoring	×	×	×	×	×	✓	✓
	File system activity monitoring	×	×	×	×	×	×	✓
	Alarms ringing	×	×	×	×	×	✓	✓
Telephony	Phone calls	✓	✓	✓	✓	✓	✓	✓
	Incoming SMS messages	✓	✓	✓	✓	×	✓	✓
	Outgoing SMS messages	×	✓	×	✓	×	✓	✓
	Address book changes	×	×	✓	✓	×	×	×
Networking	Airplane mode on/off	×	×	×	×	×	✓	✓
	Cell tower ID	✓	×	×	✓	✓	✓	✓
	Cell signal strength	×	×	×	×	✓	✓	✓
	WiFi connection info	✓	×	✓	×	✓	✓	✓
	Scan of nearby WiFi HotSpots	✓	×	×	✓	✓	×	✓
	Network status (3G/WiFi/none)	✓	×	×	✓	✓	✓	✓
	Network traffic statistics	✓	×	✓	✓	×	✓	✓
	Network packet sniffing	×	×	×	✓	×	×	✓
	Opened URL logging	×	✓	✓	✓	×	×	×
	Bluetooth state changes	×	×	×	✓	×	✓	✓
	Bluetooth packet sniffing	×	×	×	×	×	×	×
	NFC device scanning	×	×	×	×	×	×	×
	NFC packet sniffing	×	×	×	×	×	×	×
Apps & events	Broadcast intents logging	×	×	×	×	×	×	✓
	Running services	✓	×	✓	✓	×	✓	✓
	Running applications and activities	✓	×	✓	✓	✓	✓	✓
	Foreground activity detection	×	×	×	×	✓	×	✓
	In-app UI interactions and changes	×	✓	✓	×	✓	×	✓
	App installs/uninstalls	×	✓	✓	✓	✓	✓	✓
Geoloc.	Location services status	✓	✓	×	×	✓	×	✓
	Coarse location	✓	✓	✓	✓	✓	✓	✓
	Precise location	×	✓	✓	✓	✓	×	✓
	Total logged features	15	10	16	18	15	17	**44**

DeviceAnalyzer is the one that logs the largest number of data sources. This tool is monolithic and not natively extensible, nor is it open-source, making it unsuitable for researchers that want to add their own custom logging plug-ins to the experiment. Moreover, *DeviceAnalyzer* collects all data and merges it into a global data set, controlled by its authors. Although the authors provide access to the data set (on request), this makes it impossible for a researcher to deploy a specific, customized experiment to a specific group of users.

PhoneLab [34] is a full Android OS distribution that integrates logging features into the original Android source code. The authors of PhoneLab review and accept third party change requests to the codebase and integrate them into the distribution, which is later deployed on a number of devices (their testbed) under their control. This is an interesting approach, as it allows to integrate logging at an OS level, bypassing API restrictions and overheads. However, we also feel that such approach has some shortcomings. First, creating an experiment with PhoneLab requires non-trivial development knowledge, as the contributors are required to branch, modify, rebuild and merge the Android source code. On the other hand, DELTA is designed to be usable for people that are not necessarily skilled in programming. Second, similarly to *DeviceAnalyzer* [44], *PhoneLab* is managed entirely by the PhoneLab team, and contributors have no control over deployment of their changes and over the data collection. Indeed, the PhoneLab team has to evaluate and approve every change made to the codebase, with no guarantees on how long the experiment will be kept running on the testbed. Finally, all gathered data is granted at discretion of the PhoneLab team. We believe our approach with DELTA is more flexible, as experiments can be distributed directly to participants and left running as long as the author requires. Moreover the gathered data using DELTA can be collected and accessed directly without intermediaries.

LiveLab [37] is a tool built for the iOS operating system. This tool allows logging of various sensor readings and context data, with support for uploading it to a remote server. Similarly to other tools we examined, *LiveLab* does not provide fine-grained tuning of polling intervals. In addition to this, since apps not approved by the manufacturer cannot be installed on iOS devices, it requires an unlocked ("jailbroken") copy of iOS. This requirement strongly limits the number of devices on which it can be deployed. For our implementation, we decided to target the Android operating system, to achieve maximum flexibility and guarantee a large potential user base. Contrarily to *LiveLab*, our tool is designed to require an unlocked ("rooted") device only for certain advanced logging features, not obtainable through the standard operating system API.

2.2 Additional Related Work

In this section, we briefly describe some additional tools and frameworks that perform logging operations on mobile devices. We did not include these solutions in our feature comparison, either because these solutions have a limited scope, or because they are not straightforward logging tools, but rather tools that log data to achieve a different primary goal.

One of the first works on device usage logging in the literature is *MyExperience*, targeted at the Windows Mobile platform and presented in [25]. It is a logging tool capable of collecting context data and performing actions in response to triggers configured by the experiment designer. *DroidTracer* [30] is a Linux kernel module that hooks into the Android system core at a low level. This tool aims to capture app interactions, logging data such as remote method calls between apps and Android API calls (e.g., access to disk or

telephony services). In our tool, we followed the more traditional approach of leveraging (when possible) the Android API to collect our data. Implementing an approach similar to *MyExperience*, *Ohmage* [36] is an Android tool designed to present the user with interactive surveys and self-monitoring tasks depending on triggers such as time and location. This tool is also capable of automatically collecting sensor data from the device, such as accelerometer, GPS, WiFi, microphone audio recordings and cell towers logs.

A novel solution is presented by Brouwers et al. with *Pogo* [13], a middleware for Android mobile phone sensing. *Pogo* provides a JavaScript API, which exposes a limited subset of the Android API. This feature allows researchers to design experiments without being familiar with Java or the Android development platform. While the idea of using a simpler programming language to design experiments is an interesting one, it also makes *Pogo* very limited in its logging capabilities. In fact, a lot of sensor and advanced contextual data can only be captured through calls to the native Android API.

Finally, *Dynamix* [15] is an open-source extensible context-sensing framework for Android. This tool is a plugin-based framework that collects sensor data, processes it to build a "context", and then makes this context accessible to other applications via a dedicated API. While not strictly similar to our tool its basic principles are similar to the ones present in DELTA.

2.3 Limitations

We encountered three limitations common to most existing tools. These limitations greatly reduce the usefulness of such tools in a research context. In fact, we found that researchers often opted for a custom solution, developed specifically for their project, instead of relying on existing logging tools [17], [18], [26], [40].

The first problem is finding a tool that single-handedly satisfies all the data collection requirements for a particular research project. Most existing tools concentrate on one particular area (e.g., logging data from the device's sensors, collecting network packets). While a combination of different tools can be used to cover all the logging requirements of an experiment, this approach introduces some drawbacks:

- *Time consistency* - different tools will operate independently, without synchronization, and will timestamp data based on their internal timings. This can cause inconsistencies in the timestamps, which in turn can render collected data difficult to correlate precisely and thus useless.
- *Sampling rate consistency* - different tools will most likely poll the sensors/APIs of the device at different frequencies. This causes potentially undesirable differences in the granularity of collected data.
- *Data format consistency* - different tools will use different formats to save the logged data. This means that a researcher would have to perform additional (and possibly non-trivial) post-processing on the data. This post-processing can cause inconsistencies when trying to correlate data together and is generally inefficient.
- *Data collection* - if different tools are used, there is no centralized mechanism to collect gathered data and send it back to the researchers. This means that an additional ad-hoc implementation must be developed and deployed for this purpose.

The second common problem that existing tools have is limited support for fine tuning the sampling rates, in all those scenarios that require periodic polling of data (e.g., sensors reading). Existing tools tend to set a predetermined (and often relatively long) polling interval. While this limitation is usually implemented with the intent of reducing energy consumption, it can negatively impact on usefulness of collected data. In our approach, we want the experiment designers to be in control of the sampling rates of each logging operation, so that usefulness of collected data is maximized.

Finally, the tools we examined were not modular, instead using a monolithic design which did not provide an easy way to extend their logging capabilities. Even the ones that were extensible (e.g., SystemSens [20]) still had a monolithic "core" that aggregated the basic logging features provided by the tool, with extensibility being an added extra. This approach leads to a lack of customizability and often violates the "principle of least privilege", i.e., the app will often require more permissions than are actually needed to gather data for a certain experiment.

2.4 What DELTA does Differently?

Given the above premises and limitations of the existing solutions, we decided to design a tool that would adhere to the following principles:

- *Feature-richness* - our tool aims to provide a large variety of logging features out of the box, focusing on features that are not implemented (or not found together) in other logging tools.
- *Modularity* - we are aware that it would be impossible to provide support for every single logging need a researcher might have. Consequently, our tool is designed to be easy to extend through a dedicated plug-in system. This modular approach allows a developer to implement data collection features that are not available out of the box.
- *Fine-tuning* - our tool can be fine-tuned so that every single source of data can be polled at a configurable interval (or not polled at all). This approach allows the experiment designer to decide exactly which data sources she wants to monitor and at which frequency. This brings three advantages: (i) it maximizes usefulness of collected data; (ii) it reduces overhead by only tracking data that the experiment needs at the required intervals; (iii) the logger only requires the minimum privileges necessary to run each experiment. This is important to encourage user participation, especially if experiments are deployed on a larger scale.
- *Data distribution* - our tool can be configured to send the collected data back to a central server to make it accessible to the researchers that are running the experiment. This feature is important, as it lets the researchers access the data as it is collected, allowing them to monitor the logging process while it progresses. Automatic uploading also removes the need to physically retrieve collected data from the devices

(although this is still an option, which might be more suitable when dealing with large amounts of data).

Thus, we designed and implemented *DELTA - Data Extraction and Logging Tool for Android*, an extensible Android logging framework that aims to satisfy the aforementioned needs.

3 Android Operating System

In this section, we briefly introduce background knowledge and terminology about the Android Operating System. In particular, we introduce terminology and key concepts (that we refer to in the remaining part of the paper) about apps and the Android execution model, permission system and APIs.

3.1 Android Apps and Execution Model

In the Android operating system, every application ("app" hereafter) is distributed as an `APK` file, which contains all the app's components: bytecode, resources and additional metadata. An `APK` file must be explicitly installed by the user, and the same goes when it comes to uninstalling an app. When running, the code of an Android app is executed in a sandbox. This means that an Android app runs in isolation from the rest of the system, and cannot directly share memory with other apps. Any communication between different apps must be mediated by one of the inter-process communication techniques Android supports [8].

3.2 System APIs and Permission System

Android OS provides an extensive set of APIs (Accessible Programming Interfaces), which mediate access to system resources and services. While some APIs can be invoked freely, the ones that give access to sensitive data or potentially privacy sensitive services (e.g., camera, microphone) cannot. Indeed, the access to these APIs is protected by the Android Permission System [7], [22]. In practice, the operating system defines a series of permissions that an app must obtain in order to access some of the more sensitive APIs. For an app to be granted a certain permission, the developer must declare the use of that permission in the app's Manifest, an `XML` file that contains metadata information about the app. The list of permissions that an app uses is shown to the user when installing the app. Additionally, in more recent versions of the OS, the user is also required to manually grant each permission upon first use by an app.

The permission mechanism is meant to reduce the damage that could be generated by an attack that manages to take control of an app. For this reason, the Android developer guide always recommends adhering to the *principle of the least privilege*. Unfortunately, permission over-provisioning is a common malpractice, so much so that research efforts have been spent in trying to detect this problem [11], [42]. This problem is even more relevant in our context, as the logging apps we analyzed all tend to over-provision on permissions, even though a researcher might only need a fraction of the functionality offered by a logging tool.

3.3 The "Plugin Problem"

As stated in Section 2.4, we wanted our system to be modular, so that researchers could easily implement additional logging capabilities besides the ones we provide out of the box. This also means that we wanted our logging app (from now on referred to as the *logger*) to only include the bare minimum code to fulfill the needs of each experiment. In essence, this consists of using techniques that allow the logger to dynamically load additional code (i.e., a plugin) at runtime. A typical technique is using a class loader, like the one provided by the Java Virtual Machine [33]. However, even though Android programming is done in Java, and the class loader is still usable to load code dynamically from an external file, the Android permission system can severely hinder its functionality. In practice, even if the logger can technically load additional code at runtime, this code will still run in the same process as the logger, and thus with its permission set. Consequently, if the *logger* does not hold the required permissions to run the dynamically loaded code, it will not be able to execute it correctly. Note that this problem is not mitigated by the new on-demand permission system introduced in Android 6.0 [2]: even if apps can now request permissions at runtime only when needed, these still need to be declared in the manifest, and thus known at compilation time. This means that creating a plugin system for applications such as DELTA is not straightforward. Our tool is designed to log a wide spectrum of data, and access to this data is often mediated by a specific permission. There are two ways to overcome this problem at runtime, but they both come with specific drawbacks:

- *Preemptive permission over-provisioning* consists in having the logger greedily declare the use of all existing permissions and be able to execute any dynamically-loaded code that is permission-protected. The advantage is that this way, the logger will be able to load any plugin from an external file. However, this solution violates the principle of least privilege, which as we have seen can be a security risk. Moreover, it is not an effective solution: it needs to be constantly updated to include new permissions and it does not cover other Manifest extensions like services or GUI components that a plugin might require.
- *Full decoupling* consists of having the plugins required for an experiment installed as distinct, separate apps on the device, alongside the logger. This way, the logger can be a minimal skeleton that delegates logging operations to the plugins. Then, the logger will communicate with them using one of Android's built-in inter-process communication mechanisms. This solution seems more reasonable than the former one, but it has flaws of its own. The first one is that plugins have to be installed on the device as separate apps, which can be extremely annoying to the user. Second, inter-process communication in Android relies on data serialization and message routing, which result in overhead and consequently in a higher impact on system performance and battery life.

4 Our Proposal: DELTA

In this section, we present the system architecture of DELTA, and explain how our implementation deals with the plugin problem. To better understand the specific needs and concerns of researchers and users, we set up two focus groups to help us define DELTA's architecture and user interface. The first group included researchers that had worked on projects

that used smartphone data as input, as well as Android developers. This group focused on system architecture, and helped us develop a design that would be easy to use for researchers and easy to extend for developers. The second group focused on usability, and included people not necessarily familiar with Android, such as researchers, students and other department staff. Input from this group helped us with suggestions about the design and functionality of the smartphone-side user interface of DELTA.

From the input received from these two focus groups, we set the following goals that the DELTA system design should fulfill:

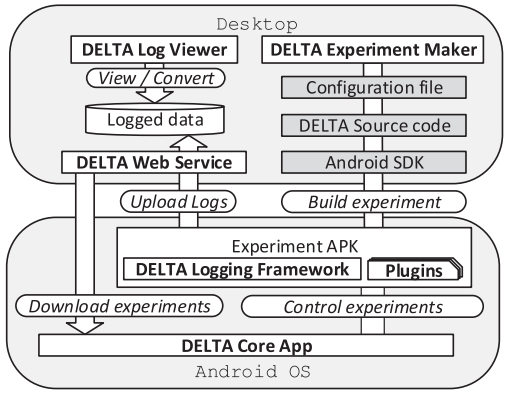
- *Minimal knowledge required by experiment designers* - we believe our tool should be accessible to everybody, even if they are not knowledgeable about the Android platform. DELTA provides dedicated graphical tools and an automated build system, which make creating new experiments straightforward.
- *Ease of extension for plugin developers* - DELTA uses a fully modular plugin-based architecture, allowing developers to easily extend its logging capabilities.
- *Simplicity and security for users* - DELTA was designed to be as easy-to-use and non-invasive of user privacy as possible, thus encouraging voluntary user participation.

4.1 System Model

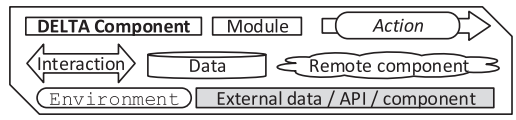
In this section, we illustrate the architecture of DELTA and show how each of the above mentioned goals (Section 4) is achieved. Fig. 1a shows a high-level view of the DELTA system architecture, highlighting the various components. Fig. 1b explains the symbology of all diagrams in this section.

DELTA allows users to configure, create, deploy and manage data logging experiments, in the form of standalone Android apps. From now on we will refer to these apps as *DELTA Experiments*, or simply *experiments*. Each DELTA Experiment is a specialized instance of DELTA Logging Framework component, plus a variable set of additional libraries, the DELTA plugins. Each plugin library contains one or more specialized modules that implement the actual data gathering operations. The DELTA Logging Framework is a generic framework that is able to instantiate and use the aforementioned plugins to schedule and perform the data logging and storage operations required by the author of the experiment.

Researchers who want to create new experiments can do so with the DELTA Experiment Maker tool. DELTA Experiment Maker is a desktop application that presents the users with a dedicated graphical interface, through which they are able to configure and build new DELTA Experiments. Depending on the chosen configuration, DELTA Experiment Maker is able to build a custom DELTA Experiment that will include the minimal set of plugins to log the required data. On the other hand, users that want to participate in experiments can use DELTA Core App. DELTA Core App is an Android app that is able to install, run, stop and remove DELTA Experiments on the user device. Optionally, DELTA Core App also allows users to download new experiments from the web or send the logged data back to the researchers. These features are made possible by



(a) Architecture of DELTA, highlighting its components and their interactions.



(b) Legend for all the system architecture diagrams.

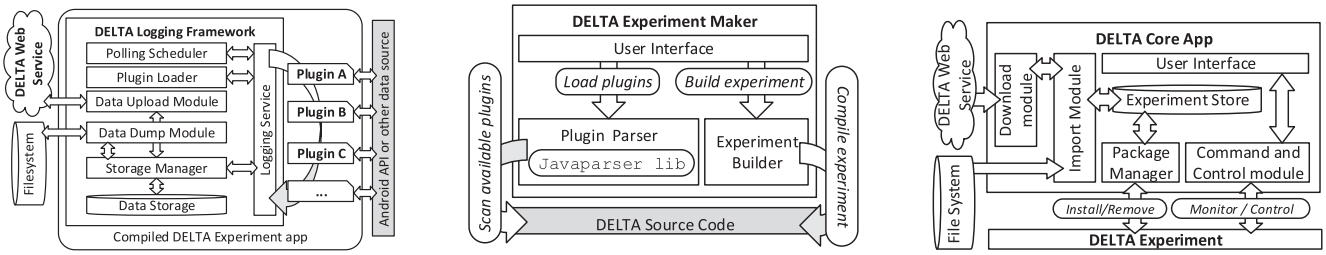
Fig. 1. DELTA's architecture and legend.

DELTA Web Service, a simple self hosting web server that researchers can run to allow remote collection of the logged data and remote deployment of new experiments. Once the logged data has been collected, DELTA Log Viewer desktop application can help researchers to merge it and convert it into different formats for better analysis.

4.2 Logging Framework

The DELTA Logging Framework is the core engine that runs DELTA's data gathering and storage operations. It is an Android app that implements a background service [9], the *Logging Service*, which is in charge of running the experiment. When a researcher creates a new experiment a copy of the DELTA Logging Framework, together with a configuration file and all the necessary plugins, is compiled into a single `APK` file. This file is then deployed to the user's device, meaning the user only has to install a single package to run an experiment. This architecture allows DELTA to reap the benefits of a plugin system, while at the same time being as non-invasive as possible. In particular, only the strictly required plugins are included with the Logging Framework when building an experiment. Consequently, the generated `APK` will only require the bare minimum permissions. This minimizes overhead and complies with the principle of least privileges. The anatomy and sub-components of a packaged DELTA Experiment are shown in Fig. 2a.

The *Logging Service* component is in charge of governing the experiment's life cycle. The framework itself does not implement any logging operations directly, delegating them to the various plugins instead. The service autonomously maintains wakelocks and manages the timers that trigger periodic logging operations. Its *Storage Manager* module implements facilities for timestamping, formatting and storing the logged data. This relieves plugin authors from having to manage such implementation details, so they only have to implement the routines that actually perform the logging of data.



(a) Components of a DELTA experiment package.

(b) Architecture of the DELTA Experiment Maker and its main interactions.

(c) Architecture of the DELTA Core App and its modules.

Fig. 2. DELTA's components.

Since the framework continuously runs in the background (if its experiment is running) it was important to minimize its impact on system resources, in order to save battery and not slow down the device. We designed the DELTA Logging Framework so it minimizes the time it keeps the device CPU awake. In particular, depending on the configuration of the plugins, we can have three cases:

- *Plugins that do not require periodic polling*. These are plugins that log data reactively. In this case, no wakelocks are used.
- *Plugins that need to be polled periodically, but at long intervals*. In case of plugins that do require polling, but are set to a polling frequency ≥ 10 seconds (i.e., the average time after which the CPU goes to sleep in Android devices), we use the energy-efficient alarms subsystem [3]. This facility allows us to schedule a periodic CPU wake-up without keeping the device constantly awake.
- *Plugins that need to be polled at fast intervals*. In these cases, it is necessary to resort to wakelocks to keep the device awake, as the Alarms subsystem does not guarantee sufficient precision to schedule low-latency events.

Even when using wakelocks, we aim to minimize DELTA Logging Framework's overhead, by limiting the number of timers and threads that perform polling. In particular, the *Polling Scheduler* module groups all plugins configured with an harmonic polling frequency into shared polling cycles, triggered by a single timer. This minimizes context switches and computational overhead. Nonetheless, the creator of an experiment can also decide to forgo wakelocks altogether, and only let the experiment run when the device is awake for other reasons. This is known as "piggyback sensing" [31] and is useful, for example, for experiments that only need to be running when the device screen is on.

Finally, the *Storage Manager* module implements an internal cache, invisible to the plugins, in order to minimize I/O operations. On disk, data is stored using lossless `ZIP` compression, to reduce occupation of the user's storage. Once logged and stored, data can be either uploaded remotely, via the *Data Upload* module, or dumped to a public portion of the user storage for easy manual extraction via the *Data Dump* module. For more information on data storage and format, see Section 4.6.

We also want to notice that our plugin architecture is designed in a way that makes it easy for developers to modify and extend an existing plugin. All of our plugin's source code is freely available and our interfaces provide entry points at various moments in an experiment life cycle (e.g., when the experiment starts, when the plugin posts data back to storage). This allows developers to quickly add functionality to an existing plugin with minimal effort, e.g.,: adding additional checks at the start of an experiment, changing the format in which data is stored or performing filtering/post processing before data is sent to storage.

4.3 Experiment Maker

The main components of DELTA Experiment Maker, and their interactions are shown in Fig. 2b. DELTA Experiment Maker is a graphical cross-platform desktop tool, written in Java, that allows a researcher to easily create and configure a new DELTA Experiment. The Experiment Maker is designed so that the user can create a customized experiment with minimal instructions, without any knowledge of Android or Java programming.

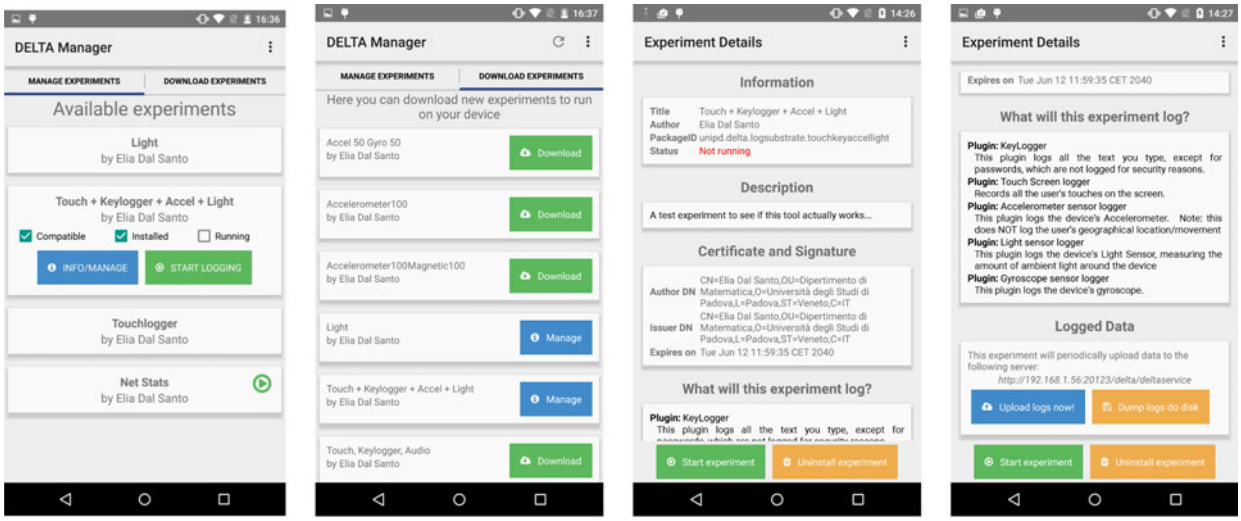
The *Plugin Parser* module leverages the `Javaparser` library to parse the DELTA source tree and detect all existing plugins. Using this information, the user interface presents the researcher with an automatically-populated list of available plugins. The researcher can then choose which plugins to include in the experiment, and is able to fine-tune the logging frequency of each one, plus any other advanced logging option. The Experiment Maker is also able to sign experiments after building, using a user-provided certificate. This protects experiments against tampering.

Once the configuration has been decided, the *Experiment Builder* module will invoke a series of custom build scripts to compile the experiment in a self-contained Android `APK` package. This package includes the main logging routines (the DELTA Logging Framework) and all (and only) the selected plugins. The generated experiment packages are lightweight (usually around 1MB) and thus can be easily distributed to participants without concern for download times.

4.4 Core App

The *DELTA Core App* is an Android app that manages DELTA Experiments, aimed at end users that want to participate in experiments. Its architecture and main modules are shown in Fig. 2c.

The DELTA Core App provides an intuitive graphical interface, from which users can manage all the experiments available on their device. In particular, the app allows them to:



(a) Browsing experiments. (b) Downloading experiments (c) Managing an experiment (1/2). (d) Managing an experiment (2/2).

Fig. 3. Screenshots depicting the DELTA Core App UI and functionality.

- Import new experiments, either directly from a file or by acquiring them through the DELTA Web Service (see Section 4.5 for details).
- Browse, install, remove and view details about stored experiments through the *Experiment Browser* and *Package Manager* modules. A detailed report is shown for each experiment, including information such as what the experiment logs, who is the author and the certificate used to sign it
- Monitor the status of an experiment and send commands to it, using the *Command and Control* module. These commands include the ability to start or stop the experiment at any time, plus additional commands to extract logged data or upload it to a remote server

The DELTA Core App maintains a list of running experiments, and is able to automatically restart them after the device is rebooted. This way, long-running experiments do not have to be manually restarted by the user in case they switch their phone off. The app is also able to automatically schedule periodic uploads of the logged data to a remote server, if the author of the experiment enabled this functionality when configuring the experiment. To avoid depleting the user's mobile data plan, this function only runs if a WiFi connection is available.

Fig. 3 shows the DELTA Core App's user interface. Fig. 3a shows a screenshot of the main activity of our app, which shows the list of installed experiments. In the main activity, the user can see at a glance what experiments she has available and whether they are compatible, installed and running. For the sake of usability, we included two quick access buttons: (i) *INFO/MANAGE* to view the details of an experiment and manage it (see Figs. 3c and 3d); and (ii) *START/STOP LOGGING* to start/stop an experiment. Moreover, a green icon (in "Net Stats" experiment in Fig. 3a) tells the user whether the experiment is running. Fig. 3b shows the list of experiments available for download from a DELTA Web Service (see Section 4.5) and allows a user to download them. The screenshots in Figs. 3c and 3d show the experiment manager activity. In this activity, the user can manage and read all the information about an experiment. In particular, information about the author is shown (including information on the digital certificate used to sign the experiment), together with a description of what the experiment does and a detailed list of what the experiment will log. This activity also allows users to easily manage every aspect of an experiment in a single place (i.e., start/stop, download/remove and install/uninstall it) and its logs (i.e., request a data dump to the local storage or trigger a data upload to the DELTA Web Service).

4.5 Web Service

The *DELTA Web Service* is a standalone self hosting web server, written in Java, that adds two key features to the system:

- It allows users to *download DELTA Experiments* directly from within the DELTA Core App. This greatly facilitates the distribution of experiments, as researchers can deploy them without having to manually distribute the required files to all users.
- It allows experiments to *send the logged data back to the researchers*. This means that researchers can collect the logged data automatically, without needing access to the user's device to perform a data dump. This also makes it possible for researchers to start analyzing incoming data while the experiment is still running, thus allowing them to speed up their analysis. It also has the added bonus of not clogging the user device with old data, since segments that are uploaded to the server are deleted from the local device cache.

DELTA Web Service is provided alongside the main components, so any experiment author can run a copy of it independently. This ensures that the experiment author has total control over the collected data, which is not sent to a third-party server.

Note that using the DELTA Web Service is entirely optional. Experiment creators can still distribute experiment packages through any other mean (e.g., email, USB side-loading), the DELTA Core App is able to import them directly from the file system. Similarly, logged data does not have to be uploaded to the web service: it can simply be dumped to the device's public storage to then be extracted manually.

4.6 Data Format and the Log Viewer

One of the advantages of having a multi-purpose logging tool like DELTA is that logged data has a consistent format, independent from the data source. In our implementation, we use JSON as our format for storing data. JSON is lightweight but at the same time, unlike the popular CSV format, it supports nested objects, so it can be used to represent complex data.

For added flexibility, our data-reporting interface also supports logging of raw binary data. This comes in handy, as some plugins may log data that is not suited to be represented efficiently through strings (e.g., the audio recording plugin).

To avoid potential data corruption or loss, the DELTA Logging Framework stores data in separate chunks, which are then uploaded to DELTA Web Service or otherwise retrieved by the interested party. To store data on disk, we use the DEFLATE algorithm. This gives good compression for textual data, without being too heavy on the device's CPU. This is important, since complex computations greatly affect the performance and battery life of mobile devices. Our tests have shown that the output from most of our plugins achieves an extremely high compression ratio, with compressed data typically ranging from 2 to 6 percent of the size of the original. All data is timestamped in milliseconds since Linux Epoch.[4]

DELTA Log Viewer is a utility that can preview, merge and convert the logs collected by a DELTA Experiment. In particular, the user can choose to export just a specific data chunk, all the data logged from a specific device or all the data collected from all devices involved in the experiment. The DELTA Log Viewer can also convert data, if requested, from its native JSON format to a more user-readable CSV file format. DELTA Log Viewer can also be used to preview the collected logs, and to merge together any binary (non-textual) data collected during the logging process.

4.7 Plugins and Extensibility Model

A *DELTA Plugin* is a specialized Java class that logs data on behalf of the DELTA Logging Framework. DELTA Plugins are contained in standard Android library packages (AAR), where each library package can contain one or more plugins. Contrary to what most other logging tools do, we designed DELTA to be completely modular, meaning that all the logging features we implemented are implemented as standard, optional DELTA plugins. No logging functionality is hard-coded in the DELTA Logging Framework which is, in fact, oblivious of the plugins. In our source code, we put plugins that require the same set of permissions together in the same library packages. This way we comply with the principle of the least privilege, as the final APK will only contain the strictly necessary libraries and thus will only require the minimum set of permissions to run the experiment.

From the point of view of a plugin developer, creating new plugins is a straightforward process. The implementation footprint consists of only a couple of standard interfaces and a Java annotation containing metadata (e.g., the author of the plugin, description of what it does, developer notes). This data is shown to experiment creators during the configuration phase. We also employ the Java annotation system to allow developers to easily define advanced options that experiment authors can modify at configuration time, so that plugins can be flexible in how they log data.

There are two distinct kinds of DELTA plugins that can be created, depending on how the data is gathered:

- *Event plugins* log data reactively, i.e., they do not need to be polled periodically, and are useful to subscribe to system events or other data sources that can actively notify a plugin of content changes.
- *Polling plugins* are polled periodically at a certain frequency, and are typically used to log data at a fixed rate. Examples include logging sensor readings or periodically collecting statistics about the system or the apps running on it.

While the DELTA codebase already includes several plugins, as discussed, we also implemented support for using plugins as standard precompiled Android Library AAR packages. This way, the DELTA Experiment Maker can load the plugins even if they are not merged into the official DELTA source tree. Plugin developers can thus easily share new plugins with experiment creators, without necessarily having to share the source code. Users of the DELTA Experiment Maker can add precompile plugins to their project by simply copying them to a specific folder, without any additional configuration needed.

5 Evaluation

In this section, we begin by summing up the current state of the project in Section 5.1. Then, we report how it impacts the user's devices in terms of security and privacy (Section 5.3) and in terms of performance and energy consumption (Section 5.4).

5.1 State of the Project

At the time of writing, the DELTA system is stable and fully functional. Table 1 compares the list of logging features between DELTA and the other apps we analyzed. As reported in Table 1, DELTA is the most feature-rich of the analyzed apps, with 44 data sources logged. This is more than double the amount of data sources handled by the most feature-rich app (after DELTA).

DELTA also covers several advanced logging features that are rarely supported by other apps (or not at all). Examples include, among others, the logging of network packets, of the device's touchscreen, a full keylogger and logging of all available sensors in the Android framework. We consider this data among the most precious for scientific research, as it allows researchers to monitor network data and profile user habits, two features that are often essential in security research.

It is worth noting that our implementation always strives to use official and documented APIs to collect data. This minimizes the risk of incompatibilities with future versions of Android and with devices we did not have a chance to test DELTA on. Our implementation is compatible with all Android 4.0.3 (and newer) devices, which covers more than 95 percent of all Android devices in use today [5]. Our implementation has also been tested with the newer versions of Android (6.x/7.x) and we confirmed it to be compatible with the new on-demand permission system

4. 00:00:00 UTC, 1 January 1970

Note that some of our plugins require root access (i.e., administration privileges) to work. These are plugins that log particularly sensitive data and have to rely on root privileges to bypass the standard Android API. In particular, affected plugins include *Touchscreen Logger*, *Packet Sniffer* and *Keyboard State logger*.

5.2 Use Cases

DELTA allows researchers to make data collection for their experiments easy, without the need to develop a custom logging app from scratch. This brings a significant benefit in terms of effort and spared time. Possible applications of DELTA on real-world researches may range from sensor and usage data collection for the development of new authentication techniques [18], [23], [26], user profiling studies [41], to new attacks on user privacy that exploit side-channel information [17], [21], [43].

In what follows, we provide a few examples of research projects that could have benefited from DELTA in the field of smartphone security and one that has successfully made use of DELTA for its data gathering. In their investigation on user actions recognition, Conti et al. in [17] had to deploy an entire framework for simulating actions on a mobile device (running a Python script that uses ADB commands on an external computer) and capture the resulting network traffic on a modified network access point. Since the gathered data was generated by two different sources, logs had to be merged and time-aligned using an additional post-processing script. This not only entailed a long development time, but the need for a custom access point and tethered devices meant that the experiment could only be run in a laboratory. With DELTA, it would have been possible to directly log Activities and Intents, as well as network traffic, and have them all already time-aligned and in a common format. Moreover, the experiment could have been run on real devices during day-to-day use, instead of a lab environment.

As another use case, DELTA may be used to ease the data collection for researchers that investigate user authentication methods based on behavioral biometrics. Giuffrida et al. in [26], [40] had to develop a custom keyboard app to collect keystrokes, accelerometer and gyroscope data. With DELTA, the development of such custom app would not be necessary since the necessary plugins are already available. Moreover, the participants could use their own keyboard apps, and not the one provided by the experimenters. A research work in this field that relies on DELTA has been recently published [39]. In this paper, the default out-of-the-box DELTA plugins were used to log data from the device sensors and to log the touches on the screen, while a custom DELTA plugin was developed by the researchers to log additional features specific to their project. Thanks to DELTA, the time required to develop and deploy the experiment was greatly reduced, leaving more time for researchers to focus on their data analysis. Moreover, DELTA could be a useful tool to provide data to "Events player" module in [12]. This module relies on real data collected from users' devices to make a sandbox for malware analysis robust against detection heuristics.

5.3 Security and User Privacy

The very nature of DELTA raises security and privacy concerns for the user. Being specifically designed to collect and store user data, it is imperative that such data is not leaked, and that other apps cannot interfere with the data collection process. When developing DELTA, we considered several possible threats in this regard, and designed it with appropriate countermeasures:

- *User trust*. When installing an experiment on their device, users are installing an app that has the ability to log (potentially) sensitive data. It is crucial that the user trusts the author of the experiment. While this is true in most scenarios (users are typically volunteers), we still provide mechanisms to protect/inform users before they install an experiment. First of all, all experiment APKs are signed with the authors private key. The Experiment Maker utility is able to perform this signing automatically, the author only needs to provide her keystore file as input. Information about the authors public certificate and the issuing Certification Authority is displayed by the DELTA Core App before the user installs an experiment. The Android system itself will downright refuse to install the experiment package if the signature is missing or invalid [4].
- *Data leak*. Considering the sensitive nature of collected data, it is important that this data is not easily accessible to external apps while still stored on the device. To achieve this, the DELTA Logging Framework will always keep logged data in its private storage space, where no other app can access it [10]. We used some commonly available file manager apps to try to extract data from DELTA's log storage and confirmed that we could not access it. Also, since every DELTA Experiment is bundled as a standalone app with its own sandbox, there is no way for an experiment to access the data gathered by other experiments. Even through the facilities provided by the DELTA Logging Framework, this is not possible since every experiment runs its own sandboxed instance of the framework. Nonetheless, logged data can be dumped to the public storage space only at a specific request from the user (typically at the end of an experiment, to collect the logged data and extract it from the device).
- *Experiment hijacking*. The DELTA Core App, as we have seen, is used by the user to start, monitor and stop experiments. It is also used to send additional commands to the experiment, like a request to dump all logged data to disk or upload it to the DELTA Web Service. It is important that other apps are inhibited from sending such commands to the Logging Framework. To achieve this, we protected the framework service entry point with a custom signature-level permission [6]. This means that a DELTA Experiment can only accept commands from an app that holds the aforementioned permission. In particular, this permission is declared by the DELTA Core App itself, and can only be acquired by apps that are signed with the same key. This effectively means that only the Core App can send commands to an experiment. We tested sending rogue commands to an experiment using an app without the proper permission, and we verified that it failed.

- *Confused Deputy attack.* This type of attack [14] is similar to the previous one, the difference being that the target of hijacking would not be an experiment but the DELTA Core App itself. In this case, an attacker would try to manipulate the DELTA Core App and trick it into sending rogue commands to experiments (for example by sending the command that dumps experiment data to the public storage). To prevent this, we designed the DELTA Core App without any API or entry point other than its GUI. This way, the only method to interact with the DELTA Core App is for the user to explicitly open it and use the app's GUI to issue commands to experiments.
- *Spoofing.* We have to make sure that an external app cannot "pass itself off" as a DELTA Experiment. In other words, we do not want an app that simply includes the DELTA Logging Framework to be displayed as an available experiment in the DELTA Core App. This could indeed confuse users, and potentially trick them into starting rogue experiments. For this reason, the DELTA Core App will only list experiments that the user has willingly imported from an external file or downloaded from the DELTA Web Service. It is not possible to add an experiment "silently" to the list of available ones, other than compromising the Android sandbox itself. In order to verify this, we tested this by manually installing several DELTA Experiment APKs on a test device instead of using the DELTA Core App. We confirmed that the DELTA Core App did not list any of those rogue experiments among the installed ones.

We feel that these measures should provide a reasonable level of security to the users of DELTA. We want to underline that, when it comes to a system like DELTA, the main security concern should be one of trust. It is essential that the user trusts the author of the experiments she is installing on her device. These are self contained apps and we cannot provide any guarantees that they have not been modified to act in a rogue fashion. The associated risks are the same as if the user installs an app from an unknown source.

We also want to point out that the security of DELTA can be compromised if the system itself is compromised, e.g., if a malicious app is running with root privileges. We feel, however, that this concern is irrelevant. If a malicious app is running on the device with root privileges, it can already collect user data directly and very easily, without the need to extract it from the DELTA databases or hijacking the logging service itself.

5.4 Energy and Performance

When it comes to performance, we designed DELTA to have a low overhead. The DELTA Logging Framework runs in a single process and, as we have seen previously, it employs various optimizations to reduce the load on the device's CPU and I/O interfaces.

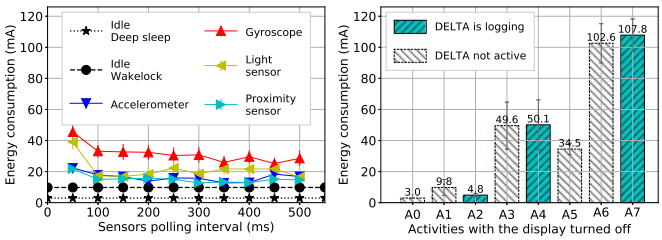
It must be noted that the performance impact of DELTA is entirely dependent on the experiment configuration, and in particular on the number of plugins used and the polling frequencies they operate at. A multi-plugin experiment with high-frequency polling times will obviously put more strain on the CPU than one that sporadically polls a single sensor. A lot of the other logging tools we examined use a fixed (and typically long) polling timer in an effort to save battery and CPU cycles. We wanted DELTA to offer researchers more flexibility when it comes to designing experiments, so we do not preemptively impose such restrictions. It is up to the researchers to balance the sample rates they need for their research against the impact such rates will have on the device's performance.

Another concern, always relevant in the field of mobile applications, is energy consumption. We used a Monsoon power monitor,[5] a high precision tool to measure DELTA's battery drain in a series of laboratory tests. Our test device was a LG Nexus 5 with only the stock set of apps installed (plus the DELTA Core App and several DELTA experiments).

We measured the battery consumption of our proposal running various DELTA experiments. We compared the battery consumption of our proposal with the one of day-to-day activities that a real life user would typically carry out on her device. We tested DELTA's battery drain under two scenarios: while the device's screen was turned off (also referred as *display off* scenario) and while it was turned on (also referred as *display on* scenario). From our preliminary tests, we noticed that most event based plugins have a trivial impact on energy consumption. In fact, such event based plugins operate reactively with a very low frequency, without scheduling dedicated poll cycles (e.g., the plugins related to system statistics and telephony). Nonetheless, other event-based plugins can log events with an high-frequency and they can have a impact on battery consumption. Two examples of high-frequency logging by event-based plugins are: (i) network packets sniffer during a streaming or a download; and (ii) touch event logger during a continuous user interaction with the touchscreen. For these reasons, we focus our evaluation on the plugins with a noticeable impact on battery consumption: polling-based plugins (e.g., sensors logging) and event-based plugins during high-frequency logging scenarios.

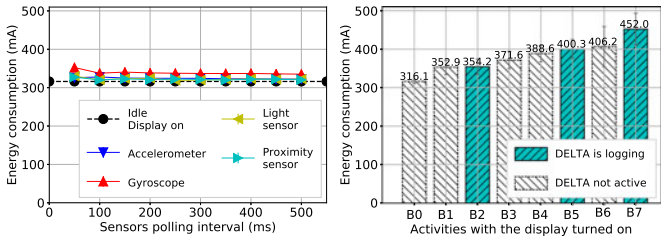
Battery consumption in display off scenario - In this first part of battery consumption evaluation, we report and discuss the batch of tests consisting of experiments in display off scenario. In Fig. 4a, we compare the energy consumption of an idle device in deep sleep, idle under wakelock (i.e., prevented artificially from entering deep sleep), and DELTA's polling based logging on four sensors: accelerometer, gyroscope, light and proximity sensors. We can notice that the energy consumption decreases as we increase the polling interval The highest consumption with a polling interval of $50ms$, while from $200ms$ onward it follows an almost constant trend. This suggests that the majority of energy consumption comes from the device's wakeful state, rather than from DELTA logging and data storage operations. On the other hand, the energy consumption of the DELTA experiment involving the gyroscope sensor is greater than the others. Unfortunately, we were not able to compare DELTA's sensors logging with other solutions because, to the best of our knowledge, the tools available on the Google Play Store only allowed us to log while the display was turned on. For this reason, we only compare them with our experiment that were running with the display turned on (as we discuss later in the second part of this evaluation).

5. www.msoon.com/LabEquipment/PowerMonitor/



(a) Polling based logging. (b) Event based logging.

Fig. 4. Energy consumption with the display off.



(a) Polling based logging. (b) Event based logging.

Fig. 5. Energy consumption with the display on.

Regarding event based logging in display off scenario, results are shown in Fig. 4b, while Table 2 explains the configuration of each measured activity and DELTA experiment. For the sake of comparison, we report once again the energy consumption of the device idle in deep sleep (A0) and under wakelock (A1). DELTA's network packet sniffing (A2) introduces a little overhead compared to deep sleep energy consumption. This because such a sniffer is limited to background network traffic, since it does not use any wakelock. Under the same quality settings (i.e., audio recorded at $8bit$ depth and $16kHz$), DELTA's audio recording plugin (A4) drains almost the same amount of energy as a normal Audio Recorder app found on Google Play (A3). Another test compares energy drain while playing the same song through headphones using the Google Music app, with and without DELTA running. In particular, we compare the energy consumption of playing music offline (A5) (i.e., the song was stored on the device's internal memory), streaming the same song through the Internet (A6) and streaming it while DELTA's network packet sniffer is running (A7). As we can notice, the difference between offline playback and streaming is considerable, while the overhead of having DELTA logging all traffic while streaming is very small, about $5mA$.

Battery consumption in display on scenario - In this second part of our evaluation, we report the results of some experiments that executed while the device's screen was turned on (display on scenario). When the screen is on, the device never enters deep sleep, allowing us to compare DELTA's performance against other logging tools that are not designed to operate with the screen off. In Fig. 5a, we can see that the display itself (more precisely, its backlight) is the main source of battery drain (i.e., around $318mA$). In comparison, the overhead caused by DELTA's polling-based logging is limited, ranging from around $6mA$ for the accelerometer ($324mA$ in total) and around $32mA$ for the gyroscope ($350mA$ in total), both polled every $100ms$.

TABLE 2
The Experiment Settings Presented in Fig. 4b

Label	Experiment
A0	Idle device, deep sleeping
A1	Idle device, NOT sleeping
A2	Network traffic sniffed thorough DELTA
A3	Audio recording using an app
A4	Audio recording thorough DELTA
A5	Music playing (offline)
A6	Music playing (streaming)
A7	Music playing (streaming) and network traffic sniffed through DELTA

Similarly to the display off scenario, polling the gyroscope is more expensive in terms of energy consumption. In the display on scenario, we were able to compare our sensors logging with some other apps that log the same sensors (i.e., *Sensor Logger* by i-Realitysoft, *Sensor Data* by Vipul Lugade, and *AndroSensor* by Fiv Asim) available on the Google Play Store. Under the same settings in terms of logged sensors and polling intervals, we did not observe any noticeable discrepancy in energy consumption between DELTA (reported in Fig. 5a) and other tested sensors logging tools. We also tested audio recording, as performed by DELTA and a competing app (B1 and B2, respectively) while the display was turned on. As reported in Fig. 5b and in Table 3, we measured a similar behavior as previously observed with the display turned off, but with an offset due to the display energy consumption. Moreover, we evaluated the impact of playing the same video (i.e., an HD movie trailer recorded at 720p) offline with the default media player (B3), streaming on YouTube (B4) and streaming while DELTA's network packet sniffer was running (B5). Similarly to the previous music playing scenario with the display turned off, DELTA's network packet sniffing introduces a modest overhead of around $12mA$.

From our experiments, we observed that the most expensive task in terms of battery draining is logging the user's interactions with the device's touch screen. In this case, running DELTA's touch logging plugin (B7) introduces a noticeable overhead compared to performing the same actions without logging (B6), with and increased drain of around $44mA$. This fact is due to the way touch logging is performed in DELTA.

Comparison with competitor tools - To compare the performance of DELTA against its Android competitors, we run each of the available competitor together with a DELTA experiment that mimicked its behavior (i.e., an experiment logging the same features on roughly the same schedule) on

TABLE 3
The Experiment Settings Presented in Fig. 5b

Label	Experiment
B0	Idle device, display on only
B1	Audio recording using an app
B2	Audio recording thorough DELTA
B3	Video playing (offline)
B4	Video playing (streaming)
B5	Video playing (streaming) and network traffic sniffed through DELTA
B6	Touch actions performed
B7	Touch actions performed and touch logged through DELTA

the same device. We could only retrieve publicly available and still working APKs for two of the four general-purpose Android logging tools we listed in Section 2.1: DroidWatch [27] and Device Analyzer [44]. Thus, we compare the battery consumption of both these tools with our DELTA. In our tests, we ran DELTA and its competitors for over three days on an LG Nexus 5x and a Xiaomi RedMi Note 3 PRO.

In order to simulate a realistic everyday life usage scenario, and thus asses the actual impact of these tools on the user's experience, we run the tools in the background while the device owners went through their day, using the device like normal. In order to measure the battery consumption of the three tools and to avoid any interference between them, we relied on the BetterBatteryStats[6] app v2.2.2, which is able to measure very accurate battery statistics, including measurements of wakelock times. From the result of our tests, we observed that the energy consumption of the three tools was comparable, accounting for around 1 percent of total CPU drain each. In particular, DroidWatch was slightly more energy intensive, since it tended to hold wakelocks for a longer time. DELTA was in the middle, while DeviceAnalyzer was the least energy-intensive. However, we underline that DELTA was polling at a slightly faster rate than DeviceAnalyzer. Since DeviceAnalyzer does not explicitly state its polling intervals, we had to approximate it relying on its wakelock frequency. In conclusion, we can state that DELTA's battery consumption is in line with its competitors.

6 Conclusions and Future Work

In this paper, we presented DELTA - Data Extraction and Logging Tool for Android, our implementation of a multi-purpose logging tool for Android. We started by comparing similar pre-existing tools, highlighting their target audience, features and common shortcomings. Then, we showed how DELTA improves on existing solutions in terms of flexibility, customization, extensibility and logging scope. Of the solutions we examined, our solution is the only one that is built from the ground-up to be fully modular. DELTA is also the only one to achieve this without either violating the principle of least privilege or relying on inter-process communication. DELTA is also, by far, the most complete of the examined tools, logging more than forty different data sources.

Seeing as the energy problem is very important for mobile devices, we designed DELTA to have a low overhead, employing various techniques to minimize its energy impact. Our tests show how the DELTA Logging Framework is in itself lightweight, and how the majority of consumed energy either comes from the device's wakeful state or the data logging operations themselves. In our evaluation, we also assessed that, under the same logging features, DELTA's battery consumption is in line with its competitors.

We believe that DELTA's feature-richness and simple extensibility model can make it a precious tool for researchers. Writing a custom logging tool for an experiment is often a non-trivial endeavor, requiring time and knowledge about Android development. DELTA's feature-richness and high level of customization is often enough to create an experiment suited to a lot of data logging needs. When the basic set of plugins does not suffice, DELTA's modular design makes extension easy, abstracting away most of the complexities involved in developing a full-fledged custom logging tool.

Although our implementation is stable and fully working, there are ample opportunities for future extensions. One possible extension we consider very interesting is adding contextual awareness to DELTA. This idea consists of extending the DELTA Logging Framework's architecture so that plugins can be dynamically started and stopped, depending on context provided by other plugins. We would also like to expand the functionality of the DELTA Web Service, in order to allow researchers to configure and build an experiment directly from a web interface, rather than relying on a desktop application. This would allow us to provide DELTA as a service, instead of forcing researchers to install the DELTA Experiment Maker on their machines in order to create experiments.

6. BetterBatteryStats - XDA forum thread - https://goo.gl/F2gZdz

Acknowledgments

Mauro Conti is supported by a Marie Curie Fellowship funded by the European Commission (agreement PCIG11-GA-2012-321980). This work is also partially supported by the EU TagItSmart! Project (agreement H2020-ICT30-2015-688061), the EU-India REACH Project (agreement ICI+/2014/342-896), and by the projects "Physical-Layer Security for Wireless Communication", and "Content Centric Networking: Security and Privacy Issues" funded by the University of Padua. This work is partially supported by the grant n. 2017-166478 (3696) from the Cisco University Research Program Fund and Silicon Valley Community Foundation. This work is also partially funded by the project CNR-MOST/Taiwan 2016-17 "Verifiable Data Structure Streaming".

References

[1] AIDL documentation. (2017). [Online]. Available: http://goo.gl/LdyUA1
[2] Android 6 - permissions at runtime. (2017). [Online]. Available: https://goo.gl/9FTnEL
[3] Android alarms. (2017). [Online]. Available: https://goo.gl/782nTv
[4] Android app signing documentation. (2017). [Online]. Available: https://goo.gl/Oj3G0E
[5] Android distribution. (2017). [Online]. Available: https://goo.gl/jZClHK
[6] Android permission usage. (2017). [Online]. Available: http://goo.gl/L0CLjI
[7] Android permissions. (2017). [Online]. Available: http://goo.gl/zkv1K3
[8] Android security. (2017). [Online]. Available: https://goo.gl/pg71dY
[9] Android services. (2017). [Online]. Available: http://goo.gl/UnjvA3
[10] Android storage options documentation. (2017). [Online]. Available: https://goo.gl/ROScqq
[11] A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus, "Automatically securing permission-based software by reducing the attack surface: An application to android," in *Proc. 27th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2012, pp. 274–277.
[12] L. Bordoni, M. Conti, and R. Spolaor, "Mirage: Toward a stealthier and modular malware analysis sandbox for android," in *Proc. 16th Eur. Conf. Res. Comput. Security*, 2017, pp. 278–296.
[13] N. Brouwers and K. Langendoen, "Pogo, a middleware for mobile phone sensing," in *Proc. 13th ACM Int. Middleware Conf.*, 2012, pp. 21–40.

[14] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry, "Towards taming privilege-escalation attacks on andriod," in *Proc. 2nd Netw. Distrib. Syst. Security Symp.*, 2012, pp. 19–37.
[15] D. Carlson and A. Schrader, "Dynamix: An open plug-and-play context framework for android," in *Proc. 3rd IEEE Int. Conf. Internet Things*, 2012, pp. 151–158.
[16] M. Cinque, D. Cotroneo, and A. Testa, "A logging framework for the on-line failure analysis of android smart phones," in *Proc. 1st Eur. Workshop AppRoaches MObiquiTous Resilience*, 2012, Art. no. 2.
[17] M. Conti, L. Mancini, R. Spolaor, and N. V. Verde, "Analyzing android encrypted network traffic to identify user actions," *IEEE Trans. Inf. Forensics Security*, vol. 11, no. 1, pp. 114–125, Jan. 2016.
[18] M. Conti, I. Zachia-Zlatea, and B. Crispo, "Mind how you answer me!: Transparently authenticating the user of a smartphone when answering or placing a call," in *Proc. 6th ACM Symp. Inf. Comput. Commun. Security*, 2011, pp. 248–259.
[19] S. Dufau, et al., "Smart phone, smart science: How the use of smartphones can revolutionize research in cognitive science," *PloS One*, vol. 6, no. 9, 2011, Art. no. e24974.
[20] H. Falaki, R. Mahajan, and D. Estrin, "SystemSens: A tool for monitoring usage in smartphone research deployments," in *Proc. 6th Int. Workshop MobiArch*, 2011, pp. 25–30.
[21] P. Faruki, et al., "Android security: A survey of issues, malware penetration and defenses," *IEEE Commun. Surv. Tutorials*, vol. 17, no. 2, pp. 998–1022, Apr.–Jun. 2015.
[22] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proc. 18th ACM Conf. Comput. Commun. Security*, 2011, pp. 627–638.
[23] M. Frank, R. Biedert, E. Ma, I. Martinovic, and D. Song, "Touchalytics: On the applicability of touchscreen input as a behavioral biometric for continuous authentication," *IEEE Trans. Inf. Forensics Security*, vol. 8, no. 1, pp. 136–148, Jan. 2013.
[24] F. Freiling, M. Spreitzenbarth, and S. Schmitt, "Forensic analysis of smartphones: The android data extractor lite (adel)," in *Proc. Conf. Dig. Forensics Security Law*, 2011, pp. 151–160.
[25] J. Froehlich, M. Y. Chen, S. Consolvo, B. Harrison, and J. A. Landay, "MyExperience: A system for in situ tracing and capturing of user feedback on mobile phones," in *Proc. 5th ACM Int. Conf. Mobile Syst. Appl. Serv.*, 2007, pp. 57–70.
[26] C. Giuffrida, K. Majdanik, M. Conti, and H. Bos, "I sensed it was you: Authenticating mobile users with sensor-enhanced keystroke dynamics," in *Proc. Int. Conf. Detection Intrusions Malware Vulnerability Assessment*, 2014, pp. 91–111.
[27] J. Grover, "Android forensics: Automated data collection and reporting from a mobile device," *Dig. Investigation*, vol. 10, pp. S12–S20, 2013.
[28] R. Guo, T. Zhu, Y. Wang, and X. Xu, "Mobilesens: A framework of behavior logger on andriod mobile device," in *Proc. IEEE 6th Int. Conf. Pervasive Comput. Appl.*, 2011, pp. 281–286.
[29] C.-K. Hsieh, et al., "Lifestreams: A modular sense-making toolset for identifying important patterns from everyday life," in *Proc. 11th ACM Conf. Embedded Netw. Sensor Syst.*, 2013, Art. no. 5.
[30] J.-C. Kuster and A. Bauer, "Platform-centric android monitoring-modular and efficient," *arXiv preprint arXiv:1406.2041*, 2014.
[31] N. D. Lane, et al., "Piggyback crowdsensing (PCS): Energy efficient crowdsourcing of mobile sensor data by exploiting smartphone app opportunities," in *Proc. 11th ACM Conf. Embedded Netw. Sensor Syst.*, 2013, Art. no. 7.
[32] N. D. Lane, E. Miluzzo, H. Lu, D. Peebles, T. Choudhury, and A. T. Campbell, "A survey of mobile phone sensing," *IEEE Commun. Mag.*, vol. 48, no. 9, pp. 140–150, Sep. 2010.
[33] S. Liang and G. Bracha, "Dynamic class loading in the java virtual machine," *ACM SIGPLAN Notices*, vol. 33, no. 10, pp. 36–44, Oct. 1998.
[34] A. Nandugudi, et al., "Phonelab: A large programmable smartphone testbed," in *Proc. 1st Int. Workshop Sensing Big Data Mining*, 2013, pp. 1–6.
[35] M. Raento, A. Oulasvirta, and N. Eagle, "Smartphones an emerging tool for social scientists," *Sociological Methods Res.*, vol. 37, pp. 426–454, 2009.
[36] N. Ramanathan, et al., "Ohmage: An open mobile system for activity and experience sampling," in *Proc. IEEE 6th Int. Conf. Pervasive Comput. Technol. Healthcare (Pervasive Health) Workshops*, 2012, pp. 203–204.
[37] C. Shepard, A. Rahmati, C. Tossell, L. Zhong, and P. Kortum, "Livelab: Measuring wireless networks and smartphone users in the field," *ACM SIGMETRICS Perform. Eval. Rev.*, 2011.
[38] A. Smith, "Smartphone ownership–2013 update," Pew Research Center: Washington DC, vol. 12, 2013.
[39] R. Spolaor, et al., "You are how you play: Authenticating mobile users via game playing," in *Proceeding of the 2nd Workshop on Communication Security*. Berlin, Germany: Springer, 2017.
[40] V.-D. Stanciu, R. Spolaor, M. Conti, and C. Giuffrida, "On the effectiveness of sensor-enhanced keystroke dynamics against statistical attacks," in *Proc. 6th ACM Conf. Data Appl. Security Privacy*, 2016, pp. 105–112.
[41] G. Suarez-Tangil, M. Conti, J. E. Tapiador, and P. Peris-Lopez, "Detecting targeted smartphone malware with behavior-triggering stochastic models," in *Proc. Eur. Symp. Res. Comput. Security*, 2014, pp. 183–201.
[42] V. F. Taylor and I. Martinovic, "SecuRank: Starving permission-hungry apps using contextual permission analysis," in *Proc. 5th ACM CCS Workshop Security Privacy Smartphones Mobile Devices*, 2016, pp. 43–52.
[43] V. F. Taylor, R. Spolaor, M. Conti, and I. Martinovich, "AppScanner: Automatic fingerprinting of smartphone apps from encrypted network traffic," in *Proc. IEEE Eur. Symp. Security Privacy*, 2016, pp. 439–454.
[44] D. T. Wagner, A. Rice, and A. R. Beresford, "Device analyzer: Understanding smartphone usage," in *Proc. 10th Int. Conf. Mobile Ubiquitous Syst.: Comput. Netw. Serv.* 2014, pp. 195–208.

Riccardo Spolaor reveived the master's degree in computer science from the University of Padua, Italy, with his thesis about smartphone privacy attack inferring user actions via traffic analysis. He is working toward the PhD degree in Brain, Mind, and Computer Science, University of Padua, Italy. In November 2014, he started his PhD degree under the supervision of Prof. Mauro Conti. He has been a visiting PhD student with Radboud University (2015), Ruhr-Universität Bochum (2016), and the University of Oxford (2016 and 2017). His main research interests include privacy and security issues on mobile devices. In particular, he applies machine learning techniques to infer user information relying on side-channel analysis. Most of the research that he carried out up to now is about the application of machine learning classifier on network traffic and energy consumption traces. He is a student member of the IEEE.

Elia Dal Santo received the MSc graduate degree in computer science from the University of Padua, Italy, in 2015, with his thesis which consisted of the project that is the subject of this paper. He now works as a senior software analyst and developer at e-project s.r.l.

Mauro Conti received the PhD degree from the Sapienza University of Rome, Italy, in 2009. After his PhD degree, he was a post-doc researcher with the Vrije Universiteit Amsterdam, The Netherlands. He is an associate professor with the University of Padua, Italy. In 2011, he joined as an assistant professor with the University of Padua, where he became an associate professor in 2015. In 2017, he obtained the national habilitation as a full professor of computer science and computer engineering. He has been a visiting researcher at GMU (2008 and 2016), UCLA (2010), UCI (2012, 2013, 2014 and 2017), TU Darmstadt (2013), UF (2015), and FIU (2015 and 2016). He has been awarded with a Marie Curie Fellowship (2012) by the European Commission, and with a Fellowship by the German DAAD (2013). His main research interest is in the area of security and privacy. In this area, he published more than 200 papers in the topmost international peer-reviewed journals and conference. He is an associate editor of several journals, including *IEEE Communications Surveys & Tutorials* and the *IEEE Transactions on Information Forensics and Security*. He was program chair of TRUST 2015, ICISS 2016, and WiSec 2017, and general chair of SecureComm 2012 and ACM SACMAT 2013. He is senior member of the IEEE.