SOUL: An Edge-cloud System for Mobile Applications in a Sensor-rich World

Minsung Jang[*], HyunJong Lee[†], Karsten Schwan[‡], Ketan Bhardwaj[‡]

[*]AT&T Labs - Research, minsung@research.att.com
[†]University of Michigan, hyunjong@umich.edu
[‡]Georgia Institute of Technology, {karsten, ketanbj}@gatech.edu

Abstract—**With the Internet of Things, sensors are becoming ever more ubiquitous, but interacting with them continues to present numerous challenges, particularly for applications running on resource-constrained devices like smartphones. The SOUL abstractions in this paper address two issues faced by such applications: (1) access to sensors with the levels of convenience needed for their ubiquitous, dynamic use, and only by parties authorized to do so, and (2) scalability in sensor access, given today's multitude of sensors. Toward this end, SOUL, first, introduces a new abstraction for the applications to transparently and uniformly access both on-device and ambient sensors with associated actuators. Second, potentially expensive sensor-related processing needs not just occur on smartphones, but can also leverage edge- and remote-cloud resources. Finally, SOUL provides access control methods that permit users to easily define access permissions for sensors, which leverages users' social ties and captures the context in which access requests are made. SOUL demonstrates that the applications on Android platforms can scale the use of 100s of sensors with performance and energy efficiency.**

I. Introduction

Recent high-end smartphones have more than 10 embedded sensors, and there are already 6 sensors on average in roughly 30 to 40% of today's mobile phones [1], [2]. A similar trend is seen for emergent wearable devices. Mirroring the growth in device-level sensors, there is also an increasing presence of sensors in users' external environments like homes, cars, or entertainment. Tesla's Model S, for example, has 12 sensors, and Google's driverless car is known to use at least 6 sensors just for its obstacle detection unit. Smart homes can have 1000s of sensors for providing home security, automation, and entertainment [3].

This paper addresses the challenges faced by mobile applications (apps) that seek to leverage and use the dynamic sets of sensors present on mobile devices and in the environments where they operate. The issues faced by such apps are (i) the diverse nature of sensors, reflected in the need to use per-sensor protocols for interacting with them; (ii) the computational and data management challenges in interacting with sensors, particularly for applications running on resource-constrained end devices like smartphones; (iii) the dynamic nature of sensor presence because users move in and out of their proximity and run applications requiring their dynamic access; and (iv) the access privilege of sensor-collected data that possibly include sensitive data of users.

In order for such apps to efficiently interact with and manage the dynamic sets of currently accessible sensors with the associated actuators and software services, SOUL (Sensors Of Ubiquitous Life)

- **externalizes sensor & actuator interactions and processing** from the resource-constrained device to edge- and remote-cloud resources, to leverage their computational and storage abilities for running the complex sensor processing functionality;
- **automates reconfiguration of these interactions** when better-matched sensors and actuators become physically available;
- **supports existing sensor-based applications** allowing them to use SOUL's capabilities without requiring modifications to their code; and
- **authorizes sensor access at runtime** to gain protected and dynamic access for applications to sensors controlled by certain end users.

The functionalities listed above are obtained via the *SOUL aggregate* abstraction, which is a single point of access to sensors as well as actuators, and software services for the apps. This abstraction is realized by leveraging *edge cloud* infrastructure–in our case, the PCloud [4] system–to efficiently run SOUL aggregate functionality. The outcome is that with SOUL, computationally or storage-intensive data management and processing tasks for sensors can be externalized from the smartphones to run *anywhere* in the edge or remote cloud. For such actions, sensor and resource accesses are guided by dynamic access permissions.

Key to SOUL's *aggregate* abstraction is the insight that sensors along with actuators and services can efficiently be virtualized by the capabilities from edge clouds to create a new high-level abstraction so as to provide apps with a consistent and convenient access to them. Apps interact with such an abstraction presented as a single point of access for various sensor-related resources. SOUL manages the diverse nature and dynamic presence of current physical sensors and virtualizes them in the exactly same way that Android provides apps with sensors. In doing so, SOUL can supports the existing applications without requiring them to be reprogrammed. While SOUL transparently supports existing apps, new apps with the SOUL API can fully utilize SOUL's features. For example, SOUL's automated reconfiguration actions can shield

978-1-5090-3322-5/16 $31.00 © 2016 IEEE
DOI 10.1109/SEC.2016.9

apps from the need to understand what physical sensors are currently accessible. Further, the apps do not need to run processing raw data of sensors on resource- and energy-constrained smartphones. Instead, resources from edge and remote clouds can be leveraged to carry out these computationally expensive tasks, and SOUL can run the potentially costly tasks required for sensor interaction on behalf of smartphones. Finally, with the enormous number of sensors with which the apps can interact, SOUL helps users easily set up access privileges for their own sensors when users share their sensors with others. Ensuring safe and secure sharing in SOUL can be achieved by the different access privileges granted to individual sensors.

The evaluations in this paper show with SOUL, a single app can interact with 100s of physical sensors alongside associated actuators and services while minimizing the impact on a device's battery life and performance. An app from Google Play Store see reductions of up to 95.4% in access latencies for on-device sensors compared with Android sensor framework.

The remainder of the paper is organized as follows. The current usage of sensors in mobile apps is explained in Section II. SOUL-enabled use cases are shown in Section III. The design and implementation of SOUL appear in Sections IV, and V, respectively. Experimental results are in Section VI. Related work is in Sections VII. Section VIII describes conclusions and future work.

II. Sensor Use in Mobile Apps

With the advent of a world of a trillion sensors and mobile devices, a new class of applications has been predicted to emerge, providing end users with personalized services based on the precise capture of their current contexts and intents. Those apps, however, must actively interact with the numerous sensors present on mobile devices and in their current environments. Unfortunately, today's reality is that most mobile apps use only a few physical sensors, despite the fact that the devices hosting such apps are themselves sensor-rich. Estimates [1], [2] are that apps using at least one sensor in Android devices are just 0.5% of all available apps in 2012. This section describes more precisely the current status of how apps interact with sensors, enhanced with our own comprehensive study of current apps' sensor use.

A. Background

Android provides apps with the Android sensor framework in the *android.hardware* package, which is a principal means for apps to access raw data from on-device sensors. With the Android sensor API, applications must manually interact and explicitly deal with individual sensor's operations including availability check. What aggravates the situation is that in Android, sensor availability and operation methods quietly vary, depending upon manufacturers, device models from the same manufactures, and even on Android versions installed in the same device. Thus, complexity in sensor use goes beyond apps' innate functionality, which is one of our motivations. We next delve more deeply into the mobile app ecosystem and its current use of on-device and ambient sensors.

TABLE I: The most commonly used sensors.

Sensor Permission	Counts(top100, 5K)	Counts(750K)
accelerometer	66	13192
compass	15	2391
proximity	8	432
gyroscope	6	1211

B. Mobile App Analysis

1) Methodology

We created a set of tools inspired by recent studies([5], [6], [7]) to automatically download and analyze nearly one million apps from the Google Play Store. Those tools scrutinize apps' bytecodes and manifest files to see how such apps behave on the Android platform. Our initial study used the top 100 apps in each category on the Google Play Store, resulting in a total of 5,000 apps (on May 20, 2015). We then expanded it to almost all free apps in the Store (750K out of the entire 1.2 million apps on May 20, 2015).

2) Sensor Usage in Current Apps

For the top 100 apps (in each category of Google Play Store), 81 out of 5000 (1.62%) use at least one sensor. This constitutes only a small increase over the previous estimate of 0.5% reported in 2012. Our more extensive survey of 750K apps does not show any notable differences from the top 100 apps, with 1.92% of those apps using at least one sensor. Further, for apps using sensors, most of them (84%) only use a single sensor despite the multiple on-device sensors available to apps. In addition, for both cases (top100, 750K), the accelerometer is the most commonly used sensor, followed by the compass (see Tables I).

3) Discussion of Study Outcomes

Evident from the statistics reported above is the fallacy of recent predictions that mobile devices will naturally become hubs in a sensor-rich world. While it remains unclear why today's apps do not actively leverage even the potential utility of the sensors on their own devices, the industry have raised issues ([1], [2], [8], [9]) regarding this matter as follows: (i) device manufacturers may define different ways of accessing the same sensor, sometimes even for different generations of the same products, and (ii) app developers seeking to use sensors have to handle different sensor vendors and their diverse products. Consequently, (i) and (ii) cause apps that seek backward compatibility to forgo using such sensors.

SOUL reacts to those issues by improving ease of use for on-device and nearby sensors as follows: (i) tackling fragmentation via a common sensor API (Section V), with backward compatibility, (ii) providing dynamic, protected access to the sensors present in a device's current external environment (Section IV-B1).

III. SOUL Sensor Applications

This section describes (1) how SOUL supports and augments existing apps as well as Android internal services in their sensor use, and (2) how it presents opportunities for new kinds of apps that easily interact with sensors and use nearby and cloud resources to process their data.

A. Supporting Existing Apps

New and additional sensors can be used without modifying existing app, whether those sensors are embedded in the Android device or are accessible remotely.

1) Augmenting Existing Apps & Services

SOUL assists apps by automatic reconfiguration of the mapping between new physical sensors and actuators in a certain SOUL aggregate used by these apps. In fact, this functionality is available even to Android system services like its notification service as well as apps. We demonstrate the utility of this functionality with a the novel SOUL service termed 'Everything Follows Me' as an Android system service (a daemon in Unix-like system), which hooks up all interactions between Android system services and apps to create a SOUL aggregate. Along with SOUL's reconfiguration feature, this service can offer a continuous media app experience even in the case that a user's context (i.e., location) is changed without the app having to keep track of it.The actuators of the media app consists of an adjacent loudspeaker and screen like those present in a home media system and an LED indicator built via an Intel Galileo board to forward all notifications emitted by the Android notification service.

The Spotify [10] app, with the 'Everything Follows Me' service, interacts with end users via whatever display screen and speakers are close to the user's current location. In addition, the user need not be concerned about missing out on other important Android notifications since the 'Everything Follow me' service serves for Android internal notification service, e.g., to notify the user about an incoming text messages via the LED also present in each room. SOUL enables Spotify to interact with edge-cloud-controlled sensors and actuators without requiring the app to be modified. In other words, forwarding is done both for the display/sound used by Spotify and for notifications.

B. Prototype SOUL Applications

We now presents how SOUL creates opportunities for new kinds of apps that easily interact with sensors and use edge- and remote-cloud resources to efficiently process their data.

1) PoD: Processing on Demand

An important property of the SOUL is its ability to use nearby and cloud resources for potentially expensive sensor processing activities. Raw sensor data must typically be processed for meaningful use by apps, but such processing can be expensive, quickly draining a device's battery or exceeding its processing abilities. To address this, the SOUL API permits apps to encapsulate their sensor processing code into Javascript as a part of SOUL aggregates. Since it is the SOUL aggregate running these code, they can be run anywhere. i.e., not just as the code embedded in the app, but as the code running on any of the edge-cloud resources available to SOUL. We demonstrate this functionality with an app deploying a Kalman filter [11] to produce a statistically best estimate of sensor data: the app creates its SOUL aggregate with the filter code and executes the aggregate on edge-cloud resources.

2) Composing SOUL Aggregates

A common app need is to combine and make use of multiple sensors/actuators in a uniform way to realize some desired app-level functionality. This motivates the 'sensor/actuator groups' in SOUL aggregates, where each aggregate can group and operate on multiple such groups. We demonstrate this functionality with an app that permits end users to check the current time on their smartphone, but without turning on the smartphone's battery-consuming screen. This 'Don't turn on the screen' app uses a SOUL aggregate with access to a home and smartphone camera , and a phone's speaker, along with a finger-gesture recognition software service running on the edge cloud: if the home camera sees the user approaching the smartphone, the camera triggers a camera on the smartphone to check if the user does with two fingers. Two-finger gesture is interpreted as a desire to check time vs. the user grasping the entire phone, and the response is the phone's speaker stating the current time, without unlocking and activating the screen, thus conserving phone power. Once the app defines this SOUL aggregate via the SOUL Activity class, SOUL operates the aggregate without app's interventions.

3) Comprehensive Health Aggregate

A future SOUL app could implement a 'health aggregate'. This app would allow mobile devices to become hubs for health-related information about the device owner [12], [13]. For example, it could collect data such as blood sugar level measured by wearable devices (e.g., *health bracelets*), or from an exercise bike used by the owner in a gym. Analytic services, part of the SOUL aggregate, would immediately raise alarms if unusual readings are detected from the SOUL aggregate's virtual sensors. In addition, they can interact with a cloud-resident service to compute long term health statistics, and implement a dashboard. We offer this application example to show the capabilities of multiple SOUL aggregates, and exercising SOUL's dynamic authorization service (e.g., when accessing the health club bicycle sensors); all driven by the SOUL engine.

IV. SOUL Design

SOUL consists of two main building blocks: (1) **the SOUL Core** built on the edge clouds, which processes sensor-related operations requested by the SOUL Engine, and (2) **the SOUL Engine** on user's device managing all sensor-related operations required by apps as a part of the Android platform. Connections between the two are enabled by SOUL Streams. The current SOUL is built on the PCloud edge-cloud infrastructure, and includes SOUL's sensor datastore, access control methods, and resource management.

Apps use SOUL functionality to access sensor data, control sensors and actuators, and to run software services via the SOUL's *aggregate* abstraction. Specifically, with the aggregate abstraction, the apps can create consistent points of access regardless of where components encapsulated in a certain aggregate are physically located. Figure 1 overviews SOUL's design, described in more detail next.

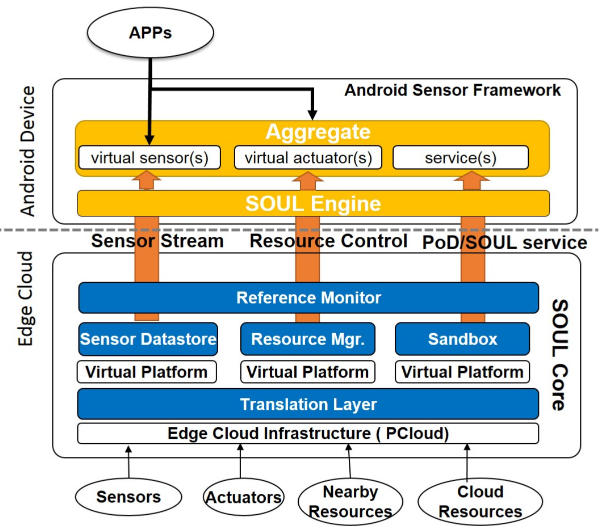


Fig. 1: SOUL Design

A. Design Principles

This section presents design principles for the SOUL middleware. The SOUL Core and Engine continue to guide more detailed architecture design and implementation on Android and PCLOUD .

1) Logical Decoupling

SOUL virtualizes the physical sensors, actuators and services to combine them into a single aggregate. Uniform APIs for such an aggregate can replace the custom APIs offered by specific physical sensors, actuators, and services. SOUL aggregate also provides flexibility for exactly crafting the abstractions desired by apps. For instance, for sensor processing requiring both current and past sensor data [14], with SOUL, sensor data can first be placed into a sensor storage service, the *Sensor Datastore*, and SOUL aggregates obtain data from that storage service rather than from physical sensors. Such intermediate sensor data storage permits data regularization and/or time series-based data use [15] and construction of entirely new sensor types from several physical sensors [16].

2) Flexible Service Execution

Ample previous work demonstrating the need for flexibility in sensor data processing ranges from early results on runtime adaptation for sensor processing([17], [18]) to recent work on novel edge cloud functionality [14] and on cloud offloading [19]. Leveraging such results and akin to first storing sensor data in a storage service before delivering it to apps, SOUL offers flexibility in where an aggregate performs its sensor processing: on the sensor itself, on the platform running the app, or on cloud processing resources. SOUL obtains such functionality by interacting with edge cloud infrastructures([4], [20], [21], [22]), via its *translation layer* shown in Figure 1. We currently use PCLOUD [4] as an edge-cloud infrastructure, but translation layers for others are straightforward to implement.

3) Compatibility with Existing Android Apps

Apps currently using device-level sensors should be able to continue to run, interacting with SOUL without the need to modify their codes. SOUL addresses this issue by exporting *illusions* of the physical sensors with which the apps interact. The apps should be able to access such illusions by Android Sensor Framework. This differentiates SOUL from previous work supporting only new applications written for their new APIs such as GSN [15] and RTDroid [23].

B. SOUL Core

The SOUL Core is comprised of three modules interfaced with the underlying edge cloud. The first is the reference monitor enables dynamic permissions to sensor data based on fine-grained access control policies by collaborating with the policy generator. Second, the Sensor Datastore interacts with physical sensors to collect and store their data into an underlying time-series database. Lastly, the resource manager makes decisions concerning the offloading of sensor management and processing from resource-poor devices to the edge or remote cloud. As PCLOUD natively supports sandboxing methods, it can better control the execution of app-provided, potentially complex and time-consuming sensor processing codes on local or remote resources in a secure way.

1) Reference Monitor

SOUL enforces access permissions to ambient or nearby sensors when apps use sensors. For instance, a ‘digital neighborhood watch’ app like the one described in [4], which will need access not only to a single home’s sensors like home cameras, smoke detectors, and intrusion sensors, but also to those in or around other homes in the neighborhood, to implement safety and notification functions for homeowners and other authorized personnel.

Mitigating privacy and security risks should be prioritized when apps access sensors because sensors often collect very sensitive and private data. On the other hand, sharing sensor data is also inevitable [24] to maximize benefit to users. To reconcile those conflicts, SOUL assists users to easily set up their own access control policy via its reference monitor along with policy generator. The reference monitor enforces fine-grained access policy for each invoked sensor while the policy generator provides the sensor owners with an easy way to construct such policies. The access control policies are realized with following design goals in mind. (i) SOUL aggregates can export only and precisely the data needed by an app ([25], [26]); (ii) sensor owners can define fine-grain permissions for apps to use certain SOUL aggregates [27]; and (iii) SOUL assists owners in creating access policies with automation support that leverages their social network services; and finally, (iv) a sandboxing mechanism is used to safely execute app-provided codes on remote resources that process sensor data [28].

Access controls driven by the reference monitor begin with mutual authentication activities between two principals, i.e., a user running an app and the owner of sensors that the app tries to access. In SOUL, those activities involve a Facebook-based app installed by the user on her Facebook account and a trust key server. After authentication, for every request to a sensor, the reference monitor should check its access permissions and provide the sensor access only if it is permitted. To do so, SOUL uses discretionary access control [29], implemented via cryptographically protected capabilities for all sensors. The access rules, i.e., policies, realized in this fashion are formulated by resource owners to control who (i.e., some principal) is authorized to access certain operations associated with the sensors in question. There are explicit operations for creating policies, granting and revoking access rights, and restricting delegation. An example is the *glance* operation in SOUL as dynamic methods for access control, thus enabling such apps to address the security and privacy issues arising for shared sensing (and sensor processing).

Glance. Access controls can be used to permit or disable sensor access, but by controlling which app can use which operations on resources managed by SOUL, it becomes possible to make finer grain decisions that can mitigate the risks to privacy inherent in permitting others to view data from personal sensors. SOUL handles this via the two distinct operations, *read* vs. *glance*, which export different granularity of data to different invokers. That is, if an app without proper privileges invokes a *read* on a sensor, that access will not be granted, but the app may succeed with its *glance* requests. The same app with *glance* from different users will see different set of data from the same sensor (e.g., only overall trends vs. detailed sensor data).

2) Policy Generator

While fine-grain protection is important, it is difficult to formulate and express such protection policies in environments targeted by SOUL [27]. We address this issue by providing to resource owners a runtime policy generator for access permissions, leveraging the wealth of information about potential principals available on social networks and data about the current context in which the request is made [30]. SOUL assumes that the owners are willing to share their resources with those who are closed to them in the real world. In Sociology, such people are referred to as having a strong social tie with the owners [31]. This tie, however, is hard to measure in the real world to construct access policies. Hence, the policy generator leverages recent studies proposing certain models [32] for predicting such social ties from the interactions observed in an Social Network Service (SNS) like Facebook. For the relationships that are unable to be captured by the SNS, it refers to the context in which a request is made. The SNS information helps the service predict the real strength of social ties, and the context information captures the situation that goes beyond social ties, which means that their complementary nature can lead to a more accurate policy. The owners can accept or customize given polices to make their own ones.

Social Network Service. Recent studies [32], [33] present models that predict actual social relationships, *social ties*, from the interactions between participants observed in SNS. Such models derive predictive variables from SNS and then use them to estimate the strength of social ties in the real world. SOUL adopts this approach by (i) periodically inspecting users' SNS interactions and then (ii) using these observations to predict their social ties to other individuals with which they interacts. This prediction, then, is the basis for constructing a set of templates for access permissions to the their resources. Users can use these templates for making final decisions about granting access permissions.

Context Information. An SNS can capture many, but not all social relationships relevant to access to SOUL aggregates. Additional information of value for deciding on access permissions include the context in which access requests are made and the intent behind making those requests [34]. A user may be visiting a gym, for instance, wishing a trainer to have temporary access to her/his personal health sensor. In SOUL, such context information [35] is captured with SOUL-specific data that can include access to the user's online calendar (e.g., gym appointments), SNS events, and physical sensors like the user's GPS location via a smartphone [30], [36].

3) Sensor Datastore

As stated earlier, sensor data is first placed into the store, then pre-processed to provide apps with different ways to access and use that data. In particular, upon an app's access to some sensor, the sensor datastore constructs a SOUL stream to transfer sensor data (in the form of SOUL aggregates) from the SOUL Engine to the SOUL Core that ultimately, provides sensor data to apps. By doing so, the SOUL allows apps to combine multiple sensors as well as actuators into a higher-level abstraction, to make it easy for apps to scale in terms of the numbers of sensors with which they interact and in terms of the degrees of required sensor processing. For example, if an application desires a time-series sensor, the store manager reads the corresponding sensor data and *bundles* time stamps with that data. It can then make available to the app an appropriate new kind of a sensor (e.g., sensor data along with time stamps). The datastore includes the following operations.

GroupBy. The datastore offers the app with the *groupBy* operation to group individual sensors based on app-desired properties. Typical properties used in *groupBy* are those based on sensor location, type, etc. The outcome is that apps interact with a single point of control, for any such set of sensors with associated actuators and services, thus making it easy for an app to see and control them.

Filter. With *filter*, apps can receive only the data that meets their criteria, expressed e.g., as time windows or sampling rates.

Bundle. The *bundle* operation can provide additional metadata like time stamps and sensor locations, to enable apps to utilize the sensor data without additional, extraneous sensor or datastore interactions. Applications are also able to define a new type of metadata *bundling* the existing ones to reduce effort on creating and managing them by the applications.

Reconfigure. If an initial configuration of a SOUL aggregate must be changed because, say, a user moves to a new location, newly available sensors may be seamlessly added to the aggregate and others may be removed. The *reconfigure* operation shields apps from such dynamics in the environments in which they operate, resulting in seamless SOUL use across changes in time, location, or even with physical sensor failure.

4) *Remote Sensor Management & Processing*

The descriptions above make clear that SOUL aggregates may require a wide array of services that implement the rich sensor data processing methods needed by applications. There are two ways to implement such services: (i) by utilizing services already present in the edge cloud [4], or (ii) by dynamically extending a SOUL aggregate via app-defined methods in JavasScript for processing the data from sensors in the aggregate. Section III-B1 describes it in more details. To enrich sensor processing by leveraging edge cloud's capabilities, SOUL permits dynamic extension to allow apps to run their own post processing algorithms on top of PCLOUD resources, termed *Processing on Demand*, and at its execution time, it sandboxes such codes for safe execution [28] where the resources allocated for those services and sandboxes are controlled by PCloud's underlying mechanism.

C. *SOUL Engine*

SOUL's realization in Android, the SOUL Engine, interacts with apps via SOUL APIs. Invocations of Engine APIs trigger additional important functionality in cooperation with the SOUL Core on edge clouds as follows: (i) Access control–the Engine initiates a process for each app's degrees of access to desired sensors. (ii) Orchestrated data movement at the granularity of a SOUL aggregate, it creates *SOUL Streams* that ultimately link SOUL to apps. (iii) Externalization–it runs the Processing-on-Demand (PoD) functions needed to interact with edge clouds, when present. If no edge cloud is available, the Engine runs the app's SOUL aggregates on available local resources on the device. (iv) Runtime mapping–SOUL permits automated methods to map physical sensors, actuators and services to SOUL aggregates with runtime remapping based on changes in user context. This eliminates app' burden required to explicitly track and response such changes in real time.

1) *Discovering Edge Clouds for SOUL Engine*

For access to edge cloud resources, SOUL seeks to discover an edge cloud whenever the user unlocks the screen on her Android device. To reduce overheads, our current implementation limits the frequency of such discovery actions to once every five minutes. Unlocking the screen triggers an interaction with a directory service located on a remote cloud (currently, an Amazon EC2 node) that returns to the device a set of edge clouds available to the user in her current environment. Which resources are returned depends on user context and her social relationships or more generally, on the access controls associated with the user requesting an access to a sensor. The reference monitor implements these access controls.

2) *SOUL Streams*

Apps should be able to use SOUL aggregates for sensors much like current Android apps use ones on a device, thus preserving the Android sensor programming model. To do so, SOUL streams are placed below the layer implementing the Android sensor programming model. Upon a request from an app, a SOUL stream is created to connect the Engine with the SOUL core. It attempts to meet Android-defined constraints on desired sensor data rates and delays, and within those constraints, it also seeks to obtain improved performance by optimizing this stream using sensor data batching. The outcome is that battery-operated mobile devices are shielded from some of the potential overheads of using physical sensors (e.g., battery drain discussed in Section VI-B); instead, these overheads are shifted to the SOUL engine's resources running the datastore on an edge cloud.

3) *Programming Model – SOUL Activities*

The SOUL Activity class is a Java abstract class for apps to use SOUL aggregates, for example, to define its SOUL aggregates–via its *compose* method, and to finalize SOUL aggregate processing–via its *trigger* method. The *remapping* method can define the SOUL aggregates that need to dynamically remap their sensors and actuators to newly available physical ones without additional app intervention. The SOUL Activity class complies with the current sensor APIs of Android because it is implemented as a wrapper of Android's sensor framework, *SensorEventListener*. Its detail appears in the SOUL source code available at https://github.com/gtpcloud/SOUL.git.

V. SELECT IMPLEMENTATION DETAIL

To realize the design principles of SOUL, our implementation must be (i) backwards compatible–allowing existing apps to continue to interact with their sensors as well as virtual sensors in SOUL; (ii) transparent–permitting the use of sensors regardless of their physical location; (iii) portable–allowing SOUL aggregates to run on any of the variety of edge cloud infrastructures; and (iv) controlled–enforcing well-defined access controls for the invokers of SOUL aggregates.

We obtain these properties as follows. First, SOUL aggregates provide legacy apps with the aforementioned sensor *illusions*. To do so, such apps can still benefit from being able to use both on-device and ambient sensors. Second, logically decoupled sensors enable apps to construct entirely new types of sensors from physical ones. Third, we demonstrate portability by realizing SOUL on diverse devices (Galaxy Prevail, S3, S4, Tab 3, and Nexus 4, 7), and by providing the edge cloud translation layer with which SOUL services can run on the PCLOUD or elsewhere (e.g., Cloudlet [21]). Finally, controlling sensor use is ensured by an access control service in SOUL.

A. *SOUL Core on Edge Clouds*

The SOUL Core implements its sensor datastore, access control mechanism backed by the reference monitor along with the policy generator, and resource manager to take sensor management followed by sensor-data processing from mobile devices on top of an edge cloud. The SOUL Core is built on PCLOUD to access distributed resources including computing capabilities, sensors, and actuators via a clean and high-level abstraction.

1) *Edge Cloud Translation Layer*

The edge cloud translation layer allows SOUL to use a different edge cloud infrastructure other than the current

PCLOUD . Since the translation layer hides complexity in the underlying system, the type of underlying edge infrastructure (i.e., Docker vs. QEMU in cloudlet) is not relevant when integrating with SOUL.

For instance, the SOUL Core would run on top of Cloudlets with cognitive services [37] via the translation layer, and the datastore would easily replace its backend DB, currently OpenTSDB with BOLT [38].

The translation layer also provides the single point of contact needed for initiating and finalizing SOUL execution, on the device or elsewhere. The PCLOUD infrastructure with the layer provides the sandboxed environments–on ambient devices and/or the remote cloud–for running individual services of SOUL.

2) Access Control to Mitigate Privacy Risks

To mitigate privacy and security concerns when users share sensors, the reference monitor in SOUL enforces access control policies, which are inherent to how SOUL aggregates are used. Access controls enforced by *the reference monitor* begin with mutual authentication activities between the mobile device user and any other users (e.g., the sensor owner) with whom she might want to interact. In our implementation, those activities involve a Facebook-based app installed by the user on her Facebook account and a server acting as a Certificate Authority (CA) with an X.509-based public key infrastructure.

The reference monitor determines what access policies (if any) exist for the user, i.e., the invoker of a specific resource. This determination is carried out by *the policy generator*, which (i) looks up the social ties shown in Facebook between the user (i.e., the invoker) and the other person involved (e.g., the sensor owner),and (ii) checks for additional context information available for that user. An example of such context is an event, noted in the user's event calendar, where the user is a scheduled participant in a shared meeting with the owner.

The generated policy, i.e., the access policy to be applied, then, is based on social tie (e.g., how well do I know the owner?) and on context (e.g., are we both attending the same scheduled event?). The policy determined in this fashion is enforced with every access by the invoker to the resources of an edge cloud. The outcome is fine-grained access control in which different access policies are enforced for every invoker. Policy enforcement is efficient, as the reference monitor issues an access token to the invoker based on a given policy, and then, every resource request uses that access token when interacting with access control (which checks the token). An additional optimization implemented in the current system skips such explicit checks for new requests made by a user for the same resource within 1 minute of previous requests.

The current implementation uses social tie prediction variables proposed by [32], but we can easily append others. For the policy generator to create policy templates from those variables, we cluster the friends of the user, who owns sensors, on Facebook into different groups using the Jenks algorithm [39]. Each group is mapped to a different policy template, and these groupings (and policy template) are presented to the owner as *assistance* in access control policy. Context is managed similarly: the policy generator again defines a suitable policy template and makes it available for inspection and possible modification by the owner. To capture where a request is made, the current SOUL engine on mobile devices should report its location from a GPS sensor on devices when it requests to connect an edge cloud belonging to others. Beyond using location data, current apps with SOUL use context determined by event pages on Facebook and the Google Calendar. An access token resulting from this context information, which is called a guest token, is required to renew every 2 hours. Further, while such context can be checked rapidly, estimation of social ties from prediction variables is slow, in part because it must walk through and collect all social traces on the owner's Facebook account (to understand the owner's ties to other users). As a result, the policy generator only periodically updates its social tie estimates, according to settings controlled by the owner, but captures context information immediately and on demand. User-defined policies are stored locally.

Access Control via glanceSensor: As discussed in Section IV-B1, when a mobile app invokes the *openSensor* call to access shared sensors, the reference monitor returns a capability token to the app, which indicates 'no access', 'glance', or 'read'. 'No access' simply rejects such an access. The read capability allows the app to use the SOUL engine API, *getSensor* operation to fully customize requests (e.g., the time windows, filters, and resolution for data). *glanceSensor* with the glace capability is a very limited version of the *getSensor* call, which can see only the sensor data that meets the conditions imposed by the owner-defined policy.

3) Sensor Management and Processing

To permit post-processing methods for sensor data to run anywhere, on the mobile device and/or on remote resources, the resource manager in the SOUL core can draw on PCLOUD resources to offload such processing *on demand*. Toward this end, an app defines a *Processing-on-Demand* (PoD) instance, consisting of algorithm written in JavaScript and metadata defining PoD inputs and outputs. To run PoD code, the resource manager uses sandboxes on PCLOUD resources to better isolate and control their activities. The sandbox's runtime executes PoD code and communicates with the datastore to get and put appropriate sensor data. Processing results are again delivered to the app in form of a SOUL aggregate. SOUL relies on a virtual machine created by the Xen hypervisor for its PoD. PCLOUD controls the lifecycle of each sandbox from its creation to termination.

4) Sensor Datastore

The SOUL Datastore uses OpenTSDB [40] as its underlying backend database. More important, however, are its actions manipulating sensor data including batching and reconfiguration as follows.

Reconfiguration. The reconfiguration service implemented by the SOUL engine (i) detects changes in the user's context, whereupon (ii) it triggers remapping between sensor streams and corresponding virtual sensors. Specifically, the current implementation detects a location change, whereupon the reconfiguration service sends a reconfiguration request to the

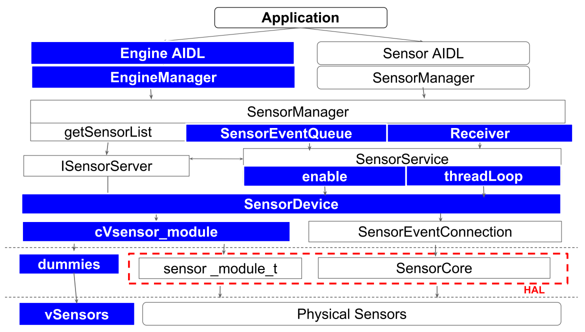


Fig. 2: SOUL's implementation in Android. The colored boxes indicate SOUL's modules added to Android.

SOUL Core. This gives rise to a new sensor stream and ultimately, to different virtual sensor data exposed to the app. The outcome is a complete elimination of app involvement when user context is changed.

Batching. To efficiently transport sensor data across the network, rather than sending individual sensor records, data is batched based on two criteria: (i) the app-defined sampling rate constitutes a constraint applied to SOUL's batching method, and (ii) MTU determines *batch_size* under that constraint, where *batch_size* simply represents the number of records packed into a single batch. In this fashion, SOUL adheres to common Android practice, yet obtains improved performance compared to per record network transfers.

B. SOUL Engine

The sensor framework in Android can be divided into three layers: (i) Java layer, (ii) system layer, and (iii) HAL. The Java layer interacts with apps, the system layer controls built-in sensors via HAL, and finally, HAL talks to the kernel-space device drivers. To transparently support existing apps, SOUL's implementation operates at all three layers, and for remote ambient sensors, SOUL does not make any assumption regarding how they connects to a device. SOUL's per-layer functionality is outlined next.

1) Hardware Abstraction Layer Approach

Android's HAL mandates that when Android introduces a new sensor type that can replace an OEM-define sensor type, the OEM *must* use the official sensor type and stringType on versions of the HAL [41]. Therefore, HAL is a seemingly attractive layer for implementing SOUL-like solutions. Unfortunately, there are several drawbacks. First, HAL itself has become *blackbox* in most Android devices, except for a handful of Google reference devices. This is because device manufacturers do not publish their HAL source codes. Second, HAL no longer provides an abstraction of all sensors on a devices because of fragmentation in the Android sensor ecosystem. As a result, apps *must* manually interact with all individual sensors, resulting in lack of portability and unnecessary overhead.

2) Java-layer approach

Recent work [42], [43] use the Java application layer in Android to export physically attached (e.g., via USB connections) or Bluetooth-connected external sensors. This may help apps access on- and off-device sensors, but they must use such new SDKs, without support for the existing apps.

3) Multi-layer Approach in SOUL

Lessons from the above approaches lead us to a multi-layer approach that spans Android's system service and application framework, shown in Figure 2. This multi-layer approach supports legacy apps and permits new apps written in our API to fully leverage SOUL. We use (i) *sensor list*, (ii) *sensor events*, and (iii) data about *receivers* in each layer of the sensor framework.

getSensorList in the system layer (JNI): Each Android device creates a 'list' of its native sensors at boot time. Utilizing this list, we load both native and *dummy sensors* into *SensorDevice* at boot time, later and on demand replacing those dummy sensors with SOUL-aggregate-based illusions of sensors.

SensorEvent in the system layer (Java): In Android, apps using on-device sensors need to implement *SensorEventListener* to create *SensorEventConnection* and enable hardware sensors.Since SOUL bypasses the opaque event-related part in HAL, we require a mechanism to generate events to trigger *SensorEventConnection* on behalf of HAL. Our implementation uses any existing sensor (currently a light sensor) as an event 'generator' for the rate at which the fastest sensor is updated. This allows us to avoid modifying the HAL, yet still substitute the data in each event with the virtualized sensor data without additional overhead.

Receiver in the HAL layer: The SOUL Core on the receiver maintains a queue of *Device Handler Number* (DHN)s indicating the next sensors to be batched. Upon receiving sensor data from the SOUL Engine, in the *handleEvent* method it substitutes the data in events generated by the light sensor with incoming data. In this method, we compare DHNs of batched events and virtual sensoŕs DHN, and then request an update via *updateSensor(DHN)* or *updateSensorGroup*, which provides a *SensorGroup* granularity update. Upon update, the *SensorManager* notifies registered listeners to handle a new *SensorEvent* via the *onSensorChanged* method in the listener. The result is a low-latency *SensorGroup* granularity update (shown in Section VI-B).

Name	Hardware/Role
Camera Nodes	Exynos 5420 and AMD E450
Speaker, Monitor A/B	at room A/B respectively
EC2	m3.large (Cloud resource)
PCloud Resources	Intel i5, i7 & Core Duo
User's Device	Galaxy S4 with Kitkat(CM11)

TABLE II: SOUL Testbed Setting

VI. Experimental Evaluation

A. Evaluation Setup

SOUL is evaluated with micro benchmarks and with the prototype apps discussed in Section III-A. Table II describes our testbed. For evaluations, we set up two different physical spaces, named Room A and Room B, which are equipped with sensors, actuators, and a monitor and speaker.

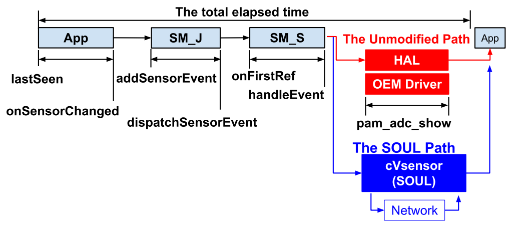


Fig. 3: Measuring elapsed time per layer. SM_J and SM_S denote the SensorManager in the Java (JNI) and the system layer, respectively.

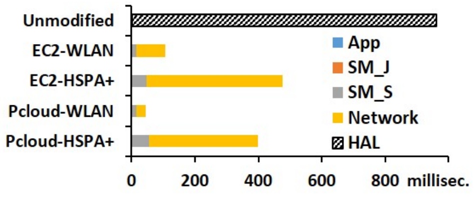


Fig. 4: Elapsed time:Each layer.

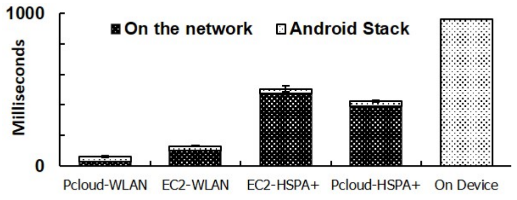


Fig. 5: Elapsed time: On the network vs. inside Android.

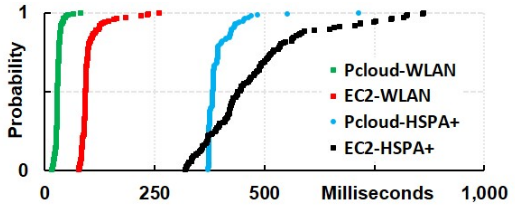


Fig. 6: The cumulative distribution functions of latencies based on resource locations and connections

B. Micro Benchmarks

1) Overhead

The overheads of reading SOUL aggregates (vs. physical sensors) are evaluated with the temperature sensor in a Samsung Galaxy S4 (GS4). We use a Google Play Store app, Sensor Readout without any modification of the app to show SOUL's backward compatibility. In unmodified Android, such an access begins with the SensorManager, followed by HAL and device drivers called *SSP* in Linux Kernel. SOUL bypasses the HAL/SSP layer, so that the SensorManager directly communicates with SOUL. Figure 3 depicts the entry and exit points of each layer for measurement.

In Figure 4 and Figure 5, an interesting result is that even when first storing sensor data in the datastore and then retrieving it via the network, the app-experienced delay in SOUL is much less than that of the unmodified HAL. Even the case that SOUL core is on the remote EC2 and the device is on the relatively slow 3G connection (labeled as EC2-HSPA+), SOUL shows better response time than the HAL. Figure 4 clearly shows that the time taken in the HAL is dominant in unmodified Android (99.8% of total time). This result suggests that the current Android sensor HAL potentially raises huge overheads when apps access sensors. However, the limited access to the sensor HAL source code makes it very hard for us to investigate this overhead further. The HAL delivers sensor reading data to apps at only four fixed intervals, which suggests a lack of guarantee in data freshness. The proximity sensor, for instance, is known to emit a new value every 0.1 seconds or less, but HAL only updates this value to apps every 0.5 seconds. When an edge device reports a sensor data to the SOUL Core, the datastore follows a conventional time-series database model to avoid information getting outdated. This also improves overall latency when moving data from SOUL to a device. In contrast, the HAL provides the same coarse-grained delay intervals to all apps.

With SOUL, most time is spent in the network shown in Figure 5, consequently, with nearby resources (PCloud) accessed via the 802.11g wireless LAN (WLAN) showing better performance than when using a remote cloud (EC2) accessed via T-Mobile's 3G connection(HSPA+). In Figure 6,the latency can change greatly if SOUL runs on the remote cloud (EC2) with the 3G network (HSPA+). Hence, it is desired that SOUL-like services run on edge-cloud resources with local network connections.

2) Scalability for Sensor Use

Scalability is evaluated in terms of power consumption when a test app is running. We measured it on a GS4 with an increasing number of sensors that the test app uses (up to 100). Power consumption is measured with a Smart Power meter [44]. To see the performance overhead when apps accesses sensors, we build another test app just reading physical sensors embedded in the device without doing any post processing. As in Figure 7, just reading five physical sensors is consuming a significant amount of CPU performance(from 20%(baseline) to 64%(Android, 5 Sensors)) resulting in dramatic increasing of CPU frequency from 600MHz to 1.6GHz. These changes generated by the Android sensor framework can hardly be justified when an app interacts with multiple sensors in term of battery life. In addition to Figure 7, Figure 8(a) indicates almost constant power consumption in the SOUL case, even with increasing numbers of sensors up to 100, with the SOUL aggregate consuming less power than when a single sensor is accessed in unmodified Android (*Ref* in the figure). Latency improvements are due in part because of the 'batch' optimization (see Section IV-B3). Figure 8(b) shows 'batch'ing gains of up to 88% in terms of latency, which results from the optimized `batch_size` when the Datastore constructs a sensor stream. Note that `batch_size` are constrained by both end user app requirements, a delay value, and the network MTU.

C. Access Control – Dynamic Authorization

To evaluate performance of our access control method, a sample Facebook account is used to measure social ties from 2675 postings with 3458 comments and 2270 likes. For the

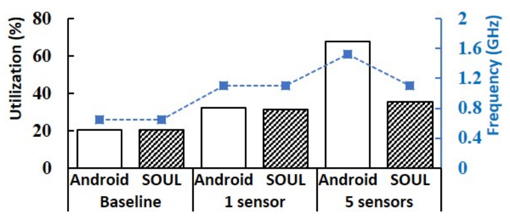


Fig. 7: The CPU overhead in Android vs. SOUL

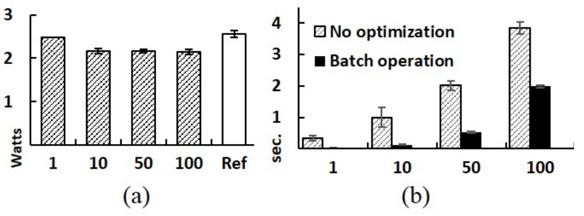


Fig. 8: (a) Power consumption as the number of sensors increases, and (b) Average latency seen by the app with batching. The Ref shows the result from one sensor in unmodified Android.

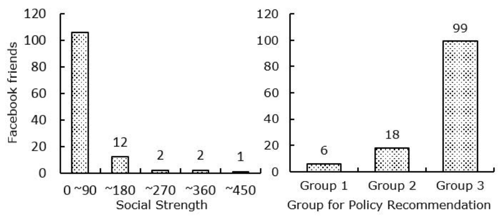


Fig. 9: Evaluation of social tie model

authentication purpose, our trust key server is deployed at an Amazon EC2 m3.medium instance.

Figure 9 shows the strengths of those ties and the clustering of the groups for access policy templates based on those ties. In this case, this account has 121 friends on Facebook, and the policy generator sorts them into three groups based on the strength of the tie. If someone falls into a proper group, the policy generator suggests a policy generated from template policies to the account user, and then the user may accept, customize or reject it.

Table III shows how quickly a policy is offered by the policy generator. The social-tie based policy is created very quickly because the policy generator caches the estimation results from the model at a local edge cloud, which means the policy generator in SOUL does not evaluate the model every time a request comes because its execution time is too long to use it in realtime. With the test settings, it takes 453 minutes since evaluating every single social interactions on Facebook to accurately generate these policies is a demanding task, its weekly recomputed results are cached and reused, with the assumption that social ties are unlikely to change over that time period.

To measure the overhead of a policy enforcement by the reference monitor, we build an Android app using SOUL, which simply tries to create and access an aggregate with a different ownership. Once an access to the aggregate is granted, the app

TABLE III: The elapsed time to create a policy

Source	Time in milliseconds
Social Tie-based	6
Context-based	
– Facebook Event	123
– Google Calendar	90

TABLE IV: The elapsed time for access control

Tasks	Time in ms
Mutual authentication	54.8
Access invocation	
– (a) Request an access	8.5
– (b) Read a value	20.3
– (c) Finalize the acess	9.7

tries to read values from the sensor. In Table IV, other than 20.3 ms (Read a value) is additional processing time for access control conducted by SOUL. Mutual authentication happens just once, and if the app reads more data, this overhead can easily be amortized.

D. Supporting Existing Apps

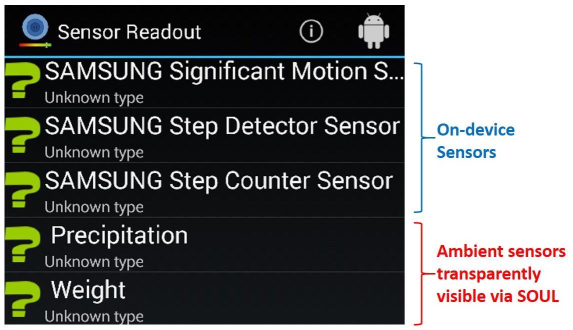
Backward compatibility is evaluated by comparing sensor accesses by unmodified apps with those using SOUL. Recall that even unmodified apps can benefit from SOUL's ability to provide access to both on-device and remote sensors via SOUL aggregates. We verify the backward compatibility of the SOUL abstractions by checking whether the existing apps in the Google Play Store can transparently access the physical sensors they already use, via illusions given by SOUL aggregates, an implicit bonus of such compatibility being that such sensors can be local or remote. Figure 10 is a screenshot of the Sensor Readout app able to transparently interact with on-device–the first three–and remote sensors.

E. Augmenting Existing Apps

The 'Everything Follows Me' service described in Section III-A1 provides a seamless media experience when running the Spotify app. In this evaluation, Android notifications also work with the this service to deliver notifications to the nearest user-visible LED. The 'blackout' time is 1918.3 milliseconds for the Spotify app and 10.9 milliseconds for Android notifications. This time is the elapsed time between the moment that the SOUL's reconfiguration notifies the user's context change to the service after detecting the change of the user's location, and the moment that the services automatically remaps to available resources in the new location. This remapping happens without user or app's intervention and for an unmodified Spotify app and the Android notification service.

F. Processing on Demand

We use the Kalman filter to evaluate SOUL's PoD feature. Our test app creates a PoD instance with this Kalman filter source code written in JavaScript and then SOUL runs the PoD instance on a sandbox on the SOUL Core. We also run the same code on the device itself to make a comparison. Figure 12 (a) shows that it takes 1.69 seconds when such injected filter


Sensor Readout

SAMSUNG Significant Motion S...
Unknown type

SAMSUNG Step Detector Sensor
Unknown type

SAMSUNG Step Counter Sensor
Unknown type

Precipitation
Unknown type

Weight
Unknown type

Fig. 10: A Screenshot of the Sensor Readout App

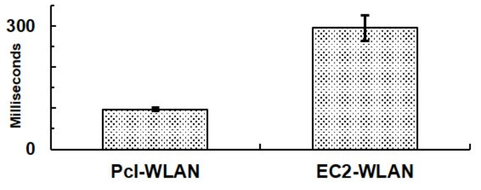
code runs in a SOUL sandbox, including all network times, while it takes 17.64 seconds for the same filter processing to run on the device. There are associated gains in energy efficiency on the GS4, shown in Figure 12 (b). We counts machine cycles for each case. the filter runs 26,158,422,002 cycles on the device whereas 39,541,615 cycles (0.15% compared with the cycles on the device) on the SOUL sandbox. Reductions in elapsed time and improved energy efficiency are explained by the number of CPU-core frequency changes in Figure 12 (a), which indicates that on-device processing operates all CPU cores up to their maximum frequency (1.6GHz) for almost half of the processing time, which is very high compared to the SOUL case. Table V breaks the execution time into each task when its PoD instance runs on a sandbox on PCLOUD .

G. Composing SOUL aggregates

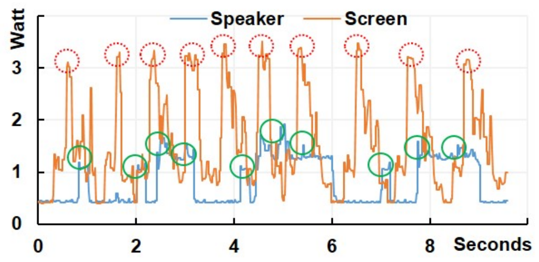
The 'Don't turn on the screen' app defines its gesture aggregate consisting of an ambient and smartphone's camera, and its proximity sensor as its aggregate. The aggregate also includes phone's speaker as an actuator, and the gesture recognition service from PCLOUD for its post processing. The recognition service runs on nearby resources, or on the remote EC2, based on a decision made by the underlying PCLOUD . The end user's request–a report on the current time–is satisfied via the phone's speaker. Results are obtained by checking the current time every second for ten seconds, by either turning on the screen or via this app. We measure the elapsed time from the moment that the finger detection service is run to the moment that the current time is reported, when the recognition service is run on local edge cloud devices or the remote cloud. Using local resources on PCloud results in a latency of about 97.4ms while latency with the EC2 remote cloud is 294.5ms as shown in Figure 11 (a). This result suggests that latency-sensitive apps may need to run on the edge clouds as long as the edge provides enough resources to process app's workload. We also compare the power and energy consumption of the device running this app vs. simply activating the screen and permitting the user to see the time. Figure 11 (b) clearly shows that this app's avoidance of the screen dramatically reduces the device's energy consumption, by up to 46%.

H. Discussion

In addition to the apps demonstrating SOUL's utility and the versatile nature of SOUL aggregates and their use, the experimental evaluations show that offloading sensor data processing (e.g., PoD) from the mobile device has advantages not only in performance and/or energy consumption, but also in the delays seen by end users, particularly when PoD can use nearby computing resources vs. the remote cloud. It also presents opportunities for creating advanced functionality, like the gesture aggregate in Section VI-G by SOUL. Furthermore, with SOUL, even unmodified apps can transparently use ambient sensors. Lastly, SOUL helps create new opportunities for innovative sensing, including via sensor composition and the ability to run potentially complex processing methods on resources beyond a single device.

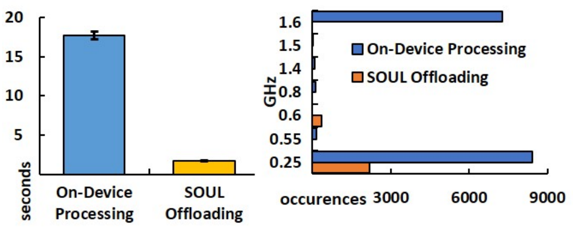


(a) Response time to state the current time

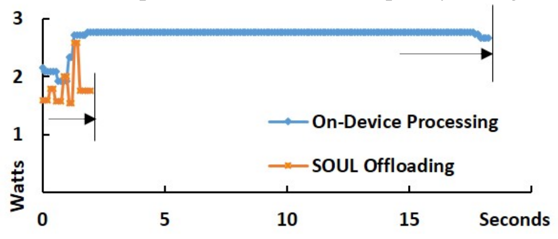


(b) The speaking and screen consume 22.14 and 40.94 mWH, respectively. Each circle shows the moment that a user acknowledges the current time.

Fig. 11: Results of the Don't turn on the screen app



(a) The elapsed time and CPU frequency changes



(b) Comparison of power consumption

Fig. 12: Results of the app with a Kalman filter

TABLE V: PoD–Elapsed Time per task with 95% interval

Task	microsec.	95%
PoD on PCLOUD	946385	
– (a) Access to Datastore	18780	6708
– (b) Access Control	1530	539
– (c) Execution in Sandbox	925065	6320
Android Stack	44013	1462
Sensor stream over Network	28824	791

VII. Related Work

Edge Cloud Infrastructures. SOUL can be built on any edge cloud infrastructures that include [4], [20], [21], [45]. In fact, the SOUL approach is somewhat similar to recent work like BOLT [38] and Gabriel [37], both of which extend underlying cloud infrastructures with new functionalities.

Sensors in Android. The Android sensor framework has recently included a Sensor Hub [46] component. This is an evolution of the original software abstraction, to a dedicated low-power companion microprocessor which records and pre-processes sensor data instead of the power-hungry application processor. This approach reduces power consumption, but does not address all the solutions provided by SOUL because the microprocessor is not capable of the full range of real-time data processing algorithms required by applications. Recent work [23], [42], [43], [47] has suggested extensions for access to off-device sensors. ODK [42] proposes a mechanism using Kernel-level device drivers to access external sensors physically connected via USB or Bluetooth. BraceForce [43] and RTdroid [23] also do so, but because they introduce custom SDKs for interacting with external sensors, they do not support legacy apps written with the standard Android SDK. In addition, access control in RTDroid [23] is similar to what is provided by SOUL, but without SOUL's policy level support. Similar to SOUL, Metis [48] opportunistically offloads sensing tasks to fixed sensors, yet does not support virtual sensor fusions, or actuator controls as SOUL Core and GroupBy operation do. Seemon [49] introduces energy-efficient context monitoring query like SOUL batching operation but lacks offloading mechanisms to leverage nearby processing power.

Programming Models. MiLAN [50] allows applications to define QoS properties for their sensing requirements, based on which it decides on suitable network and sensors configurations. Such techniques may be useful to further extend SOUL's policies that allocate appropriate edge, remote, and device-level resources to SOUL aggregates.

TeenyLIME [51] proposes a high level abstraction for data sharing among one-hop neighboring devices, but unlike SOUL aggregates, it does not fully leverage all available ambient and cloud resources. SOUL and GSN [15] share the motivation of virtualized sensors, but while GSN focuses on an infrastructure for sensor network deployment and distributed query processing, SOUL provides to mobile apps new functionality that permits them to transparently and uniformly access sensors, actuators, and services.

Similar to SOUL, MobileHub [52] proposes automatic rewriting mechanisms for mobile apps to leverage sensor hub on mobile devices. However, it relies on physically present sensors whereas SOUL interpolates virtual sensors as well.

Recent RFC 7252, 7390, and 7641 propose the sensor-oriented protocol and group-granularity operands like SOUL GroupBy. However, the proposed group operation, that requires continuous searching for all nodes or using infinite hierarchical naming, imposes a large overhead, whereas SOUL avoids these burdens by providing generic interfaces and even potential extension by developers.

OpenIoT [53] is a middleware for virtual sensor. It, however, does not incoperate the most sensor-rich device, mobile device, and lacks access control models.

Access Control and Privacy in Sensing. SenSocial [54] combines user activities on such services with sensing the physical context, using the user's mobile devices in a privacy-conserving manner. Hence, applications can easily capture both user context and sensed data. SOUL adopts elements of this approach. The anonymity mechanisms in [55], [26] could be used to implement enriched SOUL's 'glance' calls, or one could use the access control-based privacy mechanism in [35]. Liu [25] suggests a new abstraction for trusted sensors with virtualization and hardware support. Obscuring data as done in statistical databases could be used to improve the Datastore.

VIII. Conclusions and Future Work

SOUL addresses issues with sensors and sensor processing ecosystem using the capabilities of edge clouds. First, it shields applications from today's diverse sensors and vendor-specific interfaces, thus making it easier for apps to scale their sensor use. Second, SOUL's ability to perform sensor processing on external resources, make possible complex sensor processing and integration activities not limited by an individual smartphone's resource constraints. Last, apps must dynamically acquire the rights to access and interact with sensors. Thus, SOUL provides apps with required runtime permissions.

A possible future work will demonstrate the portability to other edge cloud systems and explore a seamless hand-off when mobile devices move around.

Acknowledgement

We would like to thank to our shepherd Roy Want, and anonymous reviewers for their thoughtful comments. The research work for the paper was done while Minsung Jang and HyunJong Lee were the Ph.D. students at the Georgia Institute of Technology. This paper is dedicated to the memory of Professor Karsten Schwan, who passed away last year.

References

[1] "Sensor-based apps offer lots of potential but are hindered by fragmentation," http://goo.gl/aUhi8N.

[2] "The wireless communications alliance 2012," http://goo.gl/fs7hUq.

[3] "Press release," http://goo.gl/CziKJa.

[4] M. Jang, K. Schwan, K. Bhardwaj, A. Gavrilovska, and A. Avasthi, "Personal clouds: Sharing and integrating networked resources to enhance end user experiences," in *IEEE INFOCOM*, April 2014, pp. 2220–2228.

[5] "Google play apps crawler," http://goo.gl/qt4Q5K.
[6] "Playdrone," http://goo.gl/15RaCS.
[7] N. Viennot, E. Garcia, and J. Nieh, "A measurement study of google play," in *SIGMETRICS*, ser. SIGMETRICS '14. New York, NY, USA: ACM, 2014, pp. 221–233.
[8] "Architectural requirements for always-on subsystems," http://goo.gl/MozVwY.
[9] "Open sensor platform," http://www.sensorplatforms.com/osp/.
[10] "Spotify," https://www.spotify.com/us/.
[11] R. E. Kalman, "A new approach to linear filtering and prediction problems," *Transactions of the ASME–Journal of Basic Engineering*, vol. 82, no. Series D, pp. 35–45, 1960.
[12] "The potential of sensor-based monitoring as a tool for health care, health promotion, and research," *Ann Fam Med*, vol. 9, no. 4, pp. 296–298, 2011.
[13] S. Uen, R. Fimmersa, M. Brieger, G. Nickenig, and T. Mengden, "Reproducibility of wrist home blood pressure measurement with position sensor and automatic data storage," *BMC Cardiovascular Disorders*, vol. 9, no. 20, 2009.
[14] M. Buevich, A. Wright, R. Sargent, and A. Rowe, "Respawn: A distributed multi-resolution time-series datastore," in *RTSS*, Dec 2013, pp. 288–297.
[15] K. Aberer, M. Hauswirth, and A. Salehi, "Infrastructure for data processing in large-scale interconnected sensor networks," in *Mobile Data Management, 2007 International Conference on*, May 2007, pp. 198–205.
[16] J. Crowley, "Principles and techniques for sensor data fusion," in *Multisensor Fusion for Computer Vision*, ser. NATO ASI Series, J. Aggarwal, Ed. Springer Berlin Heidelberg, 1993, vol. 99, pp. 15–36.
[17] D. Rosu, K. Schwan, S. Yalamanchili, and R. Jha, "On adaptive resource allocation for complex real-time applications," in *RTSS*, Dec 1997, pp. 320–329.
[18] D. C. Steere, A. Goel, J. Gruenberg, D. McNamee, C. Pu, and J. Walpole, "A feedback-driven proportion allocator for real-rate scheduling," in *OSDI*, ser. OSDI '99. Berkeley, CA, USA: USENIX, 1999, pp. 145–158.
[19] B. Liu, Y. Jiang, F. Sha, and R. Govindan, "Cloud-enabled privacy-preserving collaborative learning for mobile sensing," in *Sensys*, ser. SenSys '12. New York, NY, USA: ACM, 2012, pp. 57–70.
[20] C. Dixon, R. Mahajan, S. Agarwal, A. J. Brush, B. Lee, S. Saroiu, and P. Bahl, "An operating system for the home," in *NSDI*, ser. NSDI'12. Berkeley, CA, USA: USENIX, 2012, pp. 25–25.
[21] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The case for vm-based cloudlets in mobile computing," *Pervasive Computing, IEEE*, vol. 8, no. 4, pp. 14–23, Oct 2009.
[22] M. Jang and K. Schwan, "Stratus: Assembling virtual platforms from device clouds," *IEEE CLOUD*, vol. 0, pp. 476–483, 2011.
[23] Y. Yan, S. Cosgrove, V. Anand, A. Kulkarni, S. H. Konduri, S. Y. Ko, and L. Ziarek, "Real-time android with rtdroid," in *Mobisys*, ser. MobiSys '14. New York, NY, USA: ACM, 2014, pp. 273–286.
[24] "Privacy & security in a connected world," http://goo.gl/pUF2cB.
[25] H. Liu, S. Saroiu, A. Wolman, and H. Raj, "Software abstractions for trusted sensors," in *Mobisys*, ser. MobiSys '12. New York, NY, USA: ACM, 2012, pp. 365–378.
[26] L. Sweeney, "K-anonymity: A model for protecting privacy," *Int. J. Uncertain. Fuzziness Knowl.-Based Syst.*, vol. 10, no. 5, pp. 557–570, Oct. 2002.
[27] F. Roesner, T. Kohno, A. Moshchuk, B. Parno, H. Wang, and C. Cowan, "User-driven access control: Rethinking permission granting in modern operating systems," in *IEEE Security and Privacy*, May 2012, pp. 224–238.
[28] T. Das, P. Mohan, V. N. Padmanabhan, R. Ramjee, and A. Sharma, "Prism: Platform for remote sensing using smartphones," in *MobiSys*, ser. MobiSys '10. New York, NY, USA: ACM, 2010, pp. 63–76.
[29] L. Qiu, Y. Zhang, F. Wang, M. Kyung, and H. R. Mahajan, "Trusted computer system evaluation criteria," in *National Computer Security Center*, 1985.
[30] P. Aditya, V. Erdélyi, M. Lentz, E. Shi, B. Bhattacharjee, and P. Druschel, "Encore: Private, context-based communication for mobile social apps," in *Mobisys*, ser. MobiSys '14. New York, NY, USA: ACM, 2014, pp. 135–148.
[31] M. S. Granovetter, "The strength of weak ties," *American Journal of Sociology*, vol. 78, no. 6, pp. 1360–1380, 1973.
[32] E. Gilbert, "Computing tie strength," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2010.
[33] E. Gilbert and K. Karahalios, "Predicting tie strength with social media," in *CHI*, ser. CHI '09. New York, NY, USA: ACM, 2009, pp. 211–220.
[34] K. Bhardwaj, S. Sreepathy, A. Gavrilovska, and K. Schwan, "Ecc: Edge cloud composites," in *MoibleCloud*, April 2014, pp. 38–47.
[35] D. Kulkarni and A. Tripathi, "Context-aware role-based access control in pervasive computing systems," in *SACMAT*, ser. SACMAT '08. New York, NY, USA: ACM, 2008, pp. 113–122.
[36] A. Beach, M. Gartrell, X. Xing, R. Han, Q. Lv, S. Mishra, and K. Seada, "Fusing mobile, sensor, and social data to fully enable context-aware computing," in *HotMobile*, ser. HotMobile '10. New York, NY, USA: ACM, 2010, pp. 60–65.
[37] K. Ha, Z. Chen, W. Hu, W. Richter, P. Pillai, and M. Satyanarayanan, "Towards wearable cognitive assistance," in *Mobisys*, ser. MobiSys '14. New York, NY, USA: ACM, 2014, pp. 68–81.
[38] T. Gupta, R. P. Singh, A. Phanishayee, J. Jung, and R. Mahajan, "Bolt: Data management for connected homes," in *NSDI*. USENIX, April 2014.
[39] G. F. Jenks, "The data model concept in statistical mapping," in *International Yearbook of Cartography*, 2004, pp. 186–190.
[40] "Opentsdb," http://opentsdb.net.
[41] "sensor.h," http://goo.gl/U2tVRX.
[42] W. Brunette, R. Sodt, R. Chaudhri, M. Goel, M. Falcone, J. Van Orden, and G. Borriello, "Open data kit sensors: A sensor integration framework for android at the application-level," in *Mobisys*, ser. MobiSys '12. New York, NY, USA: ACM, 2012, pp. 351–364.
[43] X. Zheng, D. E. Perry, and C. Julien, "Braceforce: A middleware to enable sensing integration in mobile applications for novice programmers," in *MOBILESoft*, ser. MOBILESoft 2014. New York, NY, USA: ACM, 2014, pp. 8–17.
[44] "Odroid smart power," http://goo.gl/aGLyaJ.
[45] "Smartthings," https://www.smartthings.com.
[46] "Google android sensor hub knows how your nexus is moving," http://goo.gl/FKryUE.
[47] M. Jang, H. Lee, K. Schwan, and K. Bhardwaj, "vsensor: Toward sensor-rich mobile applications," in *Sensors to Cloud Architectures Workshop*, Febrary 2015.
[48] K. K. Rachuri, C. Efstratiou, I. Leontiadis, C. Mascolo, and P. J. Rentfrow, "Metis: Exploring mobile phone sensing offloading for efficiently supporting social sensing applications."
[49] S. Kang, J. Lee, H. Jang, H. Lee, Y. Lee, S. Park, T. Park, and J. Song, "Seemon: scalable and energy-efficient context monitoring framework for sensor-rich mobile environments," in *Mobisys*, ser. MobiSys '08. ACM, 2008, pp. 267–280.
[50] A. L. Murphy and W. B. Heinzelman, "Milan: Middleware linking applications and networks," Rochester, NY, USA, Tech. Rep., 2002.
[51] P. Costa, L. Mottola, A. L. Murphy, and G. P. Picco, "Programming wireless sensor networks with the teenylime middleware," in *Middleware*, ser. Middleware '07. New York, NY, USA: Springer-Verlag New York, Inc., 2007, pp. 429–449.
[52] H. Shen, A. Balasubramanian, A. LaMarca, and D. Wetherall, "Enhancing mobile apps to use sensor hubs without programmer effort," in *UbiComp*, ser. UbiComp '15. New York, NY, USA: ACM, 2015, pp. 227–238.
[53] "Openiot," https://goo.gl/t1hNCb.
[54] A. Mehrotra, V. Pejovic, and M. Musolesi, "Sensocial: A middleware for integrating online social networks and mobile sensing data streams," in *Middleware*, ser. Middleware '14. New York, NY, USA: ACM, 2014, pp. 205–216.
[55] M. Gruteser, G. Schelle, A. Jain, R. Han, and D. Grunwald, "Privacy-aware location sensor networks," in *HOTOS*, ser. HOTOS'03. Berkeley, CA, USA: USENIX, 2003, pp. 28–28.