To appear in ACM SIGCOMM Computer Communications Review (CCR), July, 2016

TussleOS: Managing Privacy Versus Functionality Trade-Offs on IoT Devices

Rayman Preet Singh, Benjamin Cassell, S. Keshav, Tim Brecht
Cheriton School of Computer Science, University of Waterloo
{rmmathar, becassel, keshav, brecht}@cs.uwaterloo.ca

Abstract

Networked sensors and actuators are increasingly permeating our computing devices, and provide a variety of functions for Internet of Things (IoT) devices and applications. However, this sensor data can also be used by applications to extract private information about users. Applications and users are thus in a *tussle* over access to private data. Tussles occur in operating systems when stakeholders with competing interests try to access shared resources such as sensor data, CPU time, or network bandwidth. Unfortunately, existing operating systems lack a principled approach for identifying, tracking, and resolving such tussles. Moreover, users typically have little control over how tussles are resolved. Controls for sensor data tussles, for example, often fail to address trade-offs between functionality and privacy. Therefore, we propose a framework to explicitly recognize and manage tussles. Using sensor data as an example resource, we investigate the design of mechanisms for detecting and resolving privacy tussles in a cyber-physical system, enabling privacy and functionality to be negotiated between users and applications. In doing so, we identify shortcomings of existing research and present directions for future work.

1. INTRODUCTION

Networked sensors such as accelerometers, occupancy detectors, and gyroscopes are now common in smartphones, tablets and smart home management solutions [12, 17]. Collectively, these devices make up part of the expanding Internet of Things (IoT). By 2020, analysts predict that 50 billion networked sensors will be deployed worldwide [5]. This explosive growth has transformed commodity compute platforms from simple computational resources into networked *cyber-physical* systems.

Operating systems have traditionally mediated conflicting access requirements (or *tussles* [11]) for computational resources such as CPU time, memory, disk and network bandwidth, and access to I/O devices. Networked cyber-physical systems, however, contain a broader set of contended resources (outlined in Table 1), which a wide variety of stakeholders, including device users, application developers and OS providers compete for.

As an example, consider the Nest thermostat [2], which uses passive infrared (PIR) sensors to infer users' home occupancy. This occupancy information is used by the device to manage home heating and cooling systems, but it could also be misused to reveal information which the user wishes to keep private. In this case, a privacy tussle arises from the application's desire to have unlimited sensor access and a user's desire for privacy. Michalevsky et al. [29] provide another example, demonstrating that a smartphone's MEMS gyroscope can be abused to eavesdrop on the speech of users near the phone. These privacy tussles could not have arisen in a traditional (non-networked and sensorless) compute platform and thus present an exciting area for research.

Resource	Examples
Traditional	CPU, RAM, disk, disk and network I/O
Sensors	Mic, camera, GPS, accelerometer
Actuators	Electrical switch, thermostat, lock, display
Data	Sensor data streams, user-generated content (contacts, images, etc)

Table 1: System resources.

OSes typically implement a set of resource management mechanisms and policies, such as CPU schedulers and memory managers, with little user control and no formal abstractions for tussle management. Moreover, most OSes are ineffective at resolving privacy tussles, and popular applications tend to leak private data [3]. Smartphone OSes such as Android [1] require users to allow or deny full sensor access for applications, and do not adequately inform the user of the implications of their choices on privacy or functionality. Overwhelmed users gravitate towards extreme policies, either by outright avoiding useful, privacy-sensitive applications [10], or by thoughtlessly allowing sensor access requests without understanding the consequences [16, 33]. In contrast, a tussle-based abstraction benefits i) users, by allowing them to gracefully balance application functionality against data privacy, and ii) application developers, by allowing broader distribution of applications.

Prior systems have attempted to manage privacy tussles [8, 7, 30, 6, 25], but they suffer from several shortcomings. For instance, these systems do not do not permit a degradation in application functionality to protect user data privacy, and most of them require frequent device software updates to handle advances in sensing capabilities or inference algorithms. We elaborate on this in Section 9.

In their seminal work, Clark et al. [11] introduced the notion of tussles in cyberspace. We argue that cyber-physical OSes need to formally recognize privacy tussles and must implement mechanisms and policies to resolve them. Providing stakeholders (or *actors*) with a principled way to express and negotiate their privacy and functionality requirements, and ensuring that the OS resolves tussles according to structured policies, enables an easier understanding of system be-

haviour for both users and applications, while also providing a reliable way of expressing and interacting with resources.

In this paper, we ask: how can we best outfit a cyber-physical operating system with a framework for managing privacy versus functionality trade-offs? We take the first steps in answering this question by designing a framework to formalize, detect, and resolve privacy tussles. We make the following contributions:

- We propose a privacy tussle abstraction for cyber-physical operating systems.
- We survey prior work to identify mechanisms and policies that underlie a tussle-based framework.
- We identify various open problems for instantiating a privacy management framework and outline directions for future work.
- We outline an architecture that allows users and applications to detect and manage tussles. For simplicity, we focus on tussles involving sensors, that balance privacy and functionality.

2. DESIGN GOALS

Our primary design goal is to extend an existing cyber-physical OS (such as Android) with a framework that:

- Provides actors with high-level abstractions to express resource and privacy requirements.
- Allows potentially conflicting requirements to be arbitrated to achieve an appropriate balance of resource access and functionality.
- Provides the operating system with a set of mechanisms to resolve tussles between actors and to enforce the resolved behaviour. This encompasses compiling the high-level requirements of actors into manageable low-level system behaviours.

Our initial focus is on networked sensor data tussles because of their rapid emergence, and because of the sensitive nature of the information involved. Our framework therefore incorporates users and applications as the two primary stakeholders. Furthermore, our initial design ignores collusion between applications. We leave the incorporation of information flow control mechanisms to prevent malicious or colluding actors to future work.

We believe that an ideal tussle framework should demonstrate the following properties:

Expressiveness: Numerous networked sensor applications exist today, and others are rapidly emerging [5]. Therefore, the framework should allow applications to freely express their sensing requirements. That is, they should be able to specify the type of sensors the application requires, the frequency at which access is required, and the level of access. The framework should also allow the user to define acceptable sensor access by applications and understand the impact of different levels of privacy on applications' functionality.

Resolution: After the stakeholders have expressed their constraints on sensor access, the framework should i) detect conflicting requirements (i.e. tussles), and ii) resolve tussles by balancing the requirements (with user input, if necessary). The functionality of the application may be limited by the user's privacy requirements. This implies that application functionality should degrade gracefully with increasingly strict privacy requirements. This is a significant departure from most existing application frameworks, which typically require a user to accept all of an application's sensor access requests at installation time. Even a system such as Android Marshmallow, which allows a user to remove an application's previously-granted sensor capabilities, is insufficient as it only works on a binary allow or disallow basis and provides no useful inference breakdown for the user to be able to make functionality-privacy trade-offs. A user returning to adjust an application's permissions in Android Marshmallow or iOS is provided with no relatable criteria from which they can refine their decisions, and even if they were to do so, their adjustments are limited to rendering the application usable with few meaningful privacy restrictions (turning sensors on) or rendering the application unusable (turning sensors off). Rather, we believe that applications and users should be able to negotiate their requirements to arrive at a flexible and informed compromise which provides both adequate functionality for applications and privacy for users.

Robustness: The framework should guarantee that any sensor access made by an application respects the tussle resolution. In addition, since many applications rely on the integrity of sensor readings, the framework should guarantee this integrity. A sample application that requires trusted readings to be effective is an energy billing application [38].

Extensibility: The framework should be able to support new sensor types, new algorithms that can draw more sophisticated inferences using existing sensor data, and new ways in which stakeholders may want to express their requirements.

Understandability: The framework should be comprehensible to technically-illiterate users. It should allow the user to understand the functionality an application can provide, and the inferences it can draw, given a certain level of sensor access. This will make it intuitive for the user to make informed trade-offs between functionality and privacy.

3. ARCHITECTURE OUTLINE

In light of the design goals, we present an outline for a framework to define, resolve, and implement solutions to privacy tussles. In subsequent sections, we survey existing work and determine the extent to which it can be leveraged to achieve these design goals.

We focus on devices such as the Raspberry-Pi [4] that are widely used in cyber-physical systems, and which provide applications with computational resources, and networked sensors and actuators. Applications run on the device, collecting and processing sensor data, and may be supported by a server-side component. This server-side component may run on a cloud-hosted virtual machine, or a user-controlled virtual execution environment [38]. Applications may use cloud-backed storage systems such as Bolt [17] for data storage.

Figure 1 provides an overview of our framework for handling networked sensor data tussles. Application developers and users are provided with interfaces which they use to express their sensing requirements and data privacy require-

ments respectively. Application developers express their sensing requirements through timing parameters (labelled (t_s, t_w, t_p)). Users express their requirements using inferences (labelled I_1, I_2, I_3). Sections 4 and 5 explain these parameters and their interfaces in greater detail, and related work.

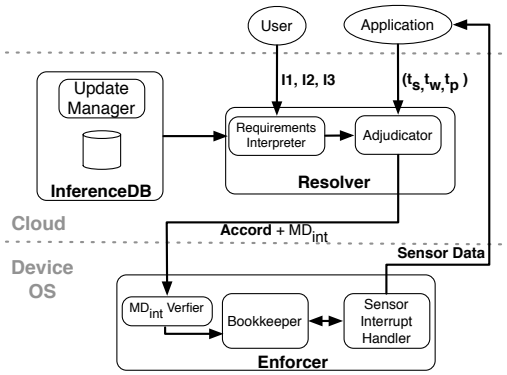

Figure 1: Design of a tussle-resolving system.

The *resolver* receives the requirements from the user and application, and resolves tussles by generating an *accord*. The accord defines the time, level, and scope of sensor access for the application. To formulate an accord, the resolver leverages the *InferenceDB*, a continually updated database of existing inference algorithms. Section 6 describes the resolver, accord, and InferenceDB in detail.

The *enforcer*, a component of the device OS, *implements* the accord by ensuring that all applications sensor accesses respect the accord's restrictions. We discuss the enforcer and its components in Section 7.

4. APPLICATION DATA REQUIREMENTS

Most applications in cyber-physical systems collect networked sensor data, use it to draw inferences about users or the environment, and perform actions based on these inferences. Previous work has examined allowing applications to identify their required *inferences* [39, 31], rather than directly expressing their sensing requirements. This enables applications to succinctly describe their requirements, while the framework performs the required data processing and delivers the requested inferences.

This methodology has several deficiencies. First, application developers are limited to using inferences supported by the framework. Second, using a different inference algorithm requires developers to integrate it into the framework (for example, formulating an inference-module in Beam [39]). This places additional development and maintenance burdens on the developer. Third, many application developers want their techniques to remain proprietary, and are reluctant to leverage such frameworks. Finally, the framework may not incorporate certain application-specific design optimizations which the developer wants to employ. Given these drawbacks, we ask: Is it possible to allow developers to directly express their sensing requirements?

We survey a range of applications that use sensors across various platforms [9, 20, 22, 32, 19, 21]. On analyzing the applications' sensor access, we observe a few similarities, which we explain in the context of a few example applications. We choose these example applications because of their diverse categories they represent (healthcare, safety, and efficiency), and because they process private information about their users. The example applications are:

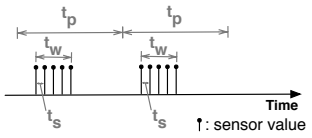

Figure 2: Data sampling parameters.

- **SleepMon** [20, 21]: This application monitors users' sleep quality using their smartphones. It uses the microphone to detect "sleep events", which are used to create a fine-grained sleep profile, compute sleep efficiency, and relate irregular sleep patterns to possible causes.
- **DriverMode** [32, 22]: This application detects when the user is in a moving vehicle, e.g., a car, and activates a driver-mode user experience on the user's smartphone. It suppresses non-critical notifications and informs other users. It uses the accelerometer to infer the user's transportation mode, and records driving periods for visualization.
- **TempControl** [19]: This application infers the occupancy of a room or a home. The inferred occupancy is then used to adjust the thermostat to save energy [2, 19, 37]. Typically, this application runs on a home computer, and interfaces with sensors in the home.

By analyzing these applications we observe that their sensing requirements, and those of many other applications, can be expressed using the following set of parameters (illustrated in Figure 2).

Sampling rate (t_s): This parameter defines the minimum time granularity for any batch of sensing values read by the application. For instance, 0.125 ms (or 8 kHz) for the microphone readings used by SleepMon [20, 21], 500 ms (or 2 Hz) for TempControl's PIR readings [19], and 100 ms (or 10 Hz) in the case of DriverMode [32]. Application developers tune the sampling rate to accommodate their inference algorithm and deployment scenario. For instance, SleepMon's inference algorithm performs a spectral analysis of the sensor readings using power spectral density. A sampling rate which is too infrequent will prevent the algorithm from being able to differentiate between events (snoring, coughing, moving), thereby impacting its detection of sleep-related events. A sampling rate which is too frequent results in resource waste, including CPU and battery, as the application will processes a larger volume of readings.

Batch size (t_w): Typical sensing applications require a batch of continuous sensor readings, which are processed using inference algorithms. This parameter defines the minimum size of any batch of sensor readings delivered to the application, expressed as a time range. For instance, SleepMon [20] requires sensor readings with t_s=0.125 ms, with a batch size (t_w) of 100 ms (or 500 ms [21]), i.e., 800 readings sampled at 8 kHz. Similarly, TempControl [19] has a t_w of 120 seconds. Developers tune the batch size to meet the

needs of their inference algorithm. An overly small batch size reduces inference accuracy, e.g., lower precision for occupancy, driving, and sleep detection. Moreover, many algorithms require a given minimum batch size in order to detect cyclic patterns or other peculiarities in the data. For instance, inferring if the user is walking using accelerometer readings requires a batch size of at least 3 seconds [23]. In contrast, an overly large batch size decreases algorithm accuracy because multiple events may co-exist within a single batch [32]. It may also lower an algorithm's sensitivity because it increases the number of missed events [32].

Periodicity (t_p): Most sensing applications perform their batch inference computations periodically (with a period t_p), rather than continuously. For instance, if TempControl [19] is exclusively interested in inferring occupancy events that last longer than 10 minutes, then the occupancy inference computation (processing a 120 second batch of PIR readings sampled at 2 Hz) needs to be carried out only once every 10 minutes, meaning t_p=600 seconds. DriverMode and SleepMon use a t_p value of 5 minutes [22, 20]. Unlike the previous parameters, t_p affects the minimum length of events that are to be detected, but does not impact the underlying inference algorithms' accuracy.

These sampling rate, batch size, and periodicity parameters allow developers to compactly describe a variety of sensing requirements. We envision that developers would add this metadata to applications, to be used by the tussle framework. Each application would require these parameters to lie in a certain operating range. Similarly, users' requirements also map to permissible ranges. The challenge lies in determining parameter values that satisfy both stakeholders' operating ranges.

We are aware that some applications do not cleanly map into our model of sensing parameters, but still contain privacy-functionality tussles. A social networking application, for instance, may ask the user for a variety of non-sensor personal information. This private data ostensibly allows the application to provide additional functionality (such as friendship recommendations, targeted news articles and advertisements, and a customized user experience), at the cost of the user revealing this information to the application and possibly its other users. A different approach would be required to specify the privacy and 'sensing' requirements of this type of application. Nevertheless, the tussle abstraction itself still holds - there is still a tension between functionality and resource access. Although the mechanism used to define the tussle may be different, it should still, given the right mechanism for specification, enable a negotiation between the application and the user.

Similarly, there are also applications that do not fit into our described model despite their use of sensors. One such example is an application that uses the camera on a device, perhaps to provide image recognition or product lookup. It is immediately apparent that timing parameters will have no effect on such an application (only a single photograph is required in most cases to provide functionality). That said, it may be possible for the application to specify alternate requirements - for instance, the compression quality of photo necessary to make recognition possible. As long as this criteria is meetable, and the application can attach a level of functionality to different levels of resource availability, then a negotiation is possible between the involved actors.

We intend to explore these extensions to our framework in future work.

5. USER PRIVACY REQUIREMENTS

Although the parameters discussed in Section 4 adequately describe the needs of applications, it would be difficult for the average user to understand how they relate to privacy and functionality. We therefore require a more intuitive interface for users to be able to specify their privacy needs.

Many OSes provide users with prompt-based accept-or-deny sensor controls to express their data privacy requirements. Smartphone OSes such as Android, Windows Phone, and iOS allow the user to grant or deny an application's access to sensors either at application installation time (in Windows Phone and older versions of Android), or when the sensors are first accessed at runtime (in Android Marshmallow and iOS). These models, however, do a poor job of informing users about which sensors an application should access and why. When prompted at install time, users are conditioned to mindlessly grant all of an application's sensor requests [16, 33, 34]. Although runtime prompts typically add some form of use context for the user, the user may be unaware of the inferences which their choices will allow the application to draw, especially if a large amount of time has passed since the application made previous sensor requests. Moreover, a malicious or poorly written application could request access to all sensors immediately with no utility context, creating a scenario almost identical to asking for permissions at install time.

Other work has focused on enabling users to express their data privacy requirements as a set of rules enforced by the OS [43, 8]. Rules can be of the form "block sensor S for a given application [43]", or can be tuples, (C, S, A), where C specifies the context under which a given sensor S is accessed, and A specifies an action (such as perturbation) which needs to be performed on the sensor data before providing it to an application [8]. Although this approach provides greater access control, it burdens the user with the task of setting rules for a growing number of sensors and applications. Furthermore, it may not be clear to the user how the varying rules relate to particular levels of functionality or privacy.

Chakraborty et al. [8] propose an approach where users are provided with a high-level inference abstraction to express their requirements. Users specify prioritized lists of acceptable and unacceptable inferences. The system then computes sensor assignments to maximize the accuracy of desired inferences and minimize the accuracy of undesired inferences. An inference-based interface was found to be highly effective. [1]

We recognize the value of an inference-based interface for specifying privacy requirements. We furthermore seek to ensure, using other components of our prototype, that a user's policy decisions are not violated by future developments in inference algorithms, nor by increases or decreases in granularity of sensor access. However, we must still address the complexity associated with specifying privacy protections on a per-inference basis, as this may be too complicated or too time consuming for some users.

[1]Although we note some additional obstacles which such approaches should overcome in Section 9, notably that they should be able to differentiate between different granularities of sensor access.

Instead, we believe that it may be possible to simplify the specification of inferences by providing users with access to an open-source catalogue of trusted privacy profiles, from which a user may select a desired set of inference policies. This is akin to "battery profiles" commonly found on mobile devices and laptop computers. An alternative method would be to poll users with high-level, privacy-related questions, in order to build a user-customized privacy profile. A third possibility would be to provide the user with a per-application slider and a breakdown of inferences and functionality available to the application. At one end of the slider, an application would have limited sensor access, acquiring fewer inferences at the cost of functionality. At the other end of the slider, the application would have broad sensor access, providing improved functionality at the cost of privacy. Based on feedback from the privacy tussle-management framework - which would inform the user of inferences the application could make given its sensor setting - and feedback from the application over the level of functionality given the user-imposed limitations, the user could make informed decisions about his or her privacy policies.

6. TUSSLE RESOLUTION

As described above, users should express their requirements using an inference-based control interface, whereas application developers express application requirements using a formal description, such as (t_s, t_w, t_p) tuples, for sensing requirements. The *resolver* (shown in Figure 1) is a service that processes a given set of applications' sensing requirements and a set of user's data privacy requirements to produce the tussle resolution, referred to as the *accord*.

Since users express their requirements through inferences, the resolver contains a *requirements interpreter*, which interfaces with the *InferenceDB*. The requirements interpreter maps the the user's requirements to resource-specific parameters such as sensor-specific bounds on parameters t_w and t_p. The *InferenceDB* is a service which provides the requirements interpreter with a mapping of the set of inferences that can be derived from the set of sensor types and sensing rates. We envision that the InferenceDB would be hosted by trusted third party providers, perhaps as public-facing web services in exchange for a fee paid by the user (or freely from trusted open-source providers). These third parties would be responsible for updating their respective InferenceDBs with existing and newly discovered inference algorithms. The InferenceDB plays the role of an informed trusted friend who warns users of the potential privacy implications of a particular application's data requirements.

After the user's requirements are converted by the requirements interpreter, they are provided to the *adjudicator*. The *adjudicator* balances the user's and application's requirements, and resolves any conflicting requirements according to the user's privacy specifications. This process incorporates the negotiation aspect of the tussle - through the adjudicator, the application may desire to inform the user of the impact of their privacy requirements on functionality. Based on this feedback, the client can choose to either retain their privacy requirements, or adjust their requirements and begin the process again. Eventually, the two parties will reach a mediated solution that works for both in terms of functionality and privacy.

As sensors and inference algorithms evolve, both the requirements interpreter and adjudicator may need to perform significant amounts of computation, as they are essentially solving an optimization problem. Therefore, we envision hosting the resolver in the cloud, either with a trusted third party provider, or on a user's personal server [38].

Applications present their requirements to the tussle framework at the time of initialization. Similarly, the user presents his or her preferences to the framework via a suitable user interface. The framework then forwards both sets of requirements to the resolver. Figure 3 illustrates a sample request.

```
User: Occupancy-10 min, Activity-30 min,
Sleep-1 hr, ...
App-1: Mic: (8 kHz, 0.125 ms, 1800 s)
App-2: Accelerometer : (10 Hz, 120 s, 300 s)
App-3: PIR: (2 Hz, 60 s, 600 s)
```

Figure 3: Example resolver requests.

We assume a two-way encrypted channel of communication between the resolver and the device OS. The resolver sends the corresponding accord to the device OS, along with integrity metadata MD_{int}. MD_{int} is computed as follows: $MD_{int} = Sig_{K^{Resolver}_{priv}}(H[Request]||H[Accord])$.
As described in Table 2, MD_{int} is a signed hash of the request and the corresponding accord. MD_{int} ensures integrity of the accord, prevents replay attacks, and allows the device OS to determine if an accord matches the last issued resolver request. This design allows the resolver to be stateless, ensuring easy scalability for a large number of users and devices (when hosted as a trusted third party service).

1. $H[x]$: Cryptographic hash of x
2. Sig_K[x]: Digital signature of x with the key K
3. K^{owner}_{pub}, K^{owner}_{priv}: a public-private key pair of *owner*
4. ||: Concatenation

Table 2: Glossary.

7. RESOLUTION ENFORCEMENT

After an accord has been formulated, its adoption and enforcement needs to be ensured. The *enforcer* (shown in Figure 1) is a service which runs as a part of the device OS. It ensures that any resource access by any application conforms to the accord generated by the resolver.

Upon receiving an accord from the resolver, the enforcer first verifies if the accord matches the current resolution request issued by the device OS (Figure 3). It then decodes and stores the accord in memory. To ensure that application resource accesses respect the accord, the enforcer maintains bookkeeping information. For instance, for networked sensor data tussles, the enforcer records the types, levels, and extents of sensor accesses by applications. The enforcer also handles sensor interrupts which may be passed to waiting applications. Upon receipt of such an interrupt, the enforcer consults its bookkeeping information and any accords, and determines the set of applications to which the data can be permissibly delivered, e.g., though an asynchronous callback. For other resource tussles, we envision that an enforcer would also encompass a CPU scheduler, memory manager, a disk manager, and a network manager.

8. FUTURE DIRECTIONS

8.1 Describing Resource Requirements

We provide a way for applications to express their sensing requirements, but a more comprehensive tussle-based framework could enable applications to express other significant resource requirements. This includes access to computational resources, actuators, and even software resources, such as user-generated content (e.g., photos, and videos), network ports, and file descriptors. Applications could also express the durations for which they require access to a resource, and the size of their requirement such as % CPU time, or amount of memory. This implies that the interface between the application and the OS (usually the process abstraction) may need to be altered to give the OS a non-black box view into the application's consumption of resources. Many modern smartphone OSes, including Android and Windows Phone, have adopted this approach, with applications divided into UI-intensive components, e.g., Android activities [1], and computation-intensive components, e.g., Android services [1]. Unfortunately, these abstractions only help to implement certain static policies defined by the OS-provider, such as maximizing battery life. For designing a tussle-based framework, existing unified resource abstractions such as Fence [26] may be leveraged. However, they need to be supplemented with i) a language to allow applications to freely express their resource requirements, and ii) policies that resolve more complex resource tussles. This will allow users to control and reason about resource tussles in the same manner as privacy tussles.

8.2 Trusted Readings

Our current work assumes that the OS is trusted, and does not tamper with sensor readings. Trusted readings are required in many applications such as billing [38] and geofencing [27]. In practice, however, it may be difficult to guarantee that a particular OS is trusted. Moreover, the user may want to provide readings that are perturbed or otherwise modified, for example to spoof readings during development, or replay previous GPS traces to shield their current location [8]. Therefore, perturbations to sensor readings by the user, and applications' desires for unperturbed readings can also be viewed as a tussle. A possible solution is to include a noise coefficient with sensor requests, $0 \leq t_n \leq 1$. A coefficient of zero corresponds to unperturbed readings, whereas a coefficient of one could correspond to entirely fabricated readings. The resolver can then balance an application's requested noise coefficient against the user's privacy requirements, while the enforcer ensures that readings delivered to applications are perturbed to the agreed upon level.

Existing work has shown that trusted readings can be provided despite an untrusted OS by securing appropriate components by using a secure execution mode in modern CPUs [27]. However, the approach in [27] requires sensor drivers to be a part of the Trusted Computing Base (TCB), and thus increases its vulnerabilities [36]. A potential solution is to partition sensor drivers' functionality so that only their security-critical components contribute to the TCB.

8.3 Tussles on a Cloud-Hosted OS

Applications such as Nest [2] use networked sensors deployed in users' homes and backhaul data to servers in the cloud which host the application logic. This is problematic in two ways. First, users' sensor data needs to be transferred to the application developer's server, even though data processing can be hosted on devices at home, for example using a "home hub" [12]-based TempControl [19] application. Second, the cloud server hosting the application logic has no means to allow users to express their data privacy requirements.

What is needed, therefore, is an instantiation of a tussle-based framework on the device such as Nest [2]. This framework may coordinate with instantiations on other user devices and/or the cloud server. The challenge therefore lies in designing a tussle-based framework for such IoT devices.

8.4 Verifiable OS Implementation

A potential direction for building tussle-oriented operating system frameworks is to use a Unikernel [28]. In this approach, the OS kernel is specialized to support only a pre-specified set of applications. Moreover, the entire kernel is written in a strongly-typed language (OCaml) that allows the OS's correctness to be formally proven, in the sense of always obeying certain high-level assertions. We believe that tussle accords can be formally expressed in terms of these assertions, thus a Unikernel that implements a tussle accord can be trusted to enforce the accord.

9. RELATED WORK

Clark et al. [11] recognize tussles between stakeholders in the networking space, and outline solutions to resolve them. Likewise, we propose recognizing tussles among stakeholders on operating systems on commodity devices, and outline mechanisms to detect and resolve them.

Information flow control (IFC) is an area of research that deals with data privacy and integrity, including data that is jointly owned and operated on by multiple actors. Past work has shown that IFC techniques can be applied to traditional operating system abstractions such as file descriptors, with minimal changes to existing applications and acceptable levels of overhead [24]. Zeldovich et al. demonstrate how a group of mutually distrustful components can communicate across physical machines without leaking information [42]. TaintDroid extends IFC into the Android runtime, dynamically tracking the flow of sensitive data between applications [14]. PiOS statically analyzes data flows in iOS applications in order to detect potential leaks [13], while PDroid uses static data flow analysis to detect privacy leakage in Android applications [43]. IFC, while useful for tracking access to sensitive information, does not necessarily inform users about what types of inferences are being generated by applications based on their sensor access. It furthermore does not address a user's desire to balance an application's functionality against privacy. Thus, while information flow control is an important component for us to consider, it is not sufficient on its own.

Existing work has proposed additions to OSes to enable sensor and actuator privacy control, and data resource management. Several systems have focused on detecting overprivileged applications, applications which request more sensor access than is needed to achieve their functionality goals, on smartphones [15, 41]. Spahn et al. [40] design an OS service that discovers application-level data objects, e.g., emails and documents, and provides users with unified object management. Santos et al. [35] propose a lease-based allocation

of resources on smartphone OSes to enable verifiable application behaviour. Haddadi et al. [18] propose a trusted arbitrator for allowing users to control applications' access to historical data, e.g. photos and videos. These approaches are complementary to ours. They focus on providing mechanisms to address various relevant sub-problems, however they do not recognize tussles between stakeholders, which are the main cause of these problems. We propose a framework to recognize, detect, and resolve these tussles which can be further enriched by incorporating these existing mechanisms.

MockDroid [7] provides users with a modified version of Android which allows them to spoof sensor readings provided to applications. Likewise, Apex [30] allows Android users to selectively grant applications sensor permissions (now an integrated feature of Android Marshmallow) and impose runtime constraints on those sensors. PMP offers runtime resource control for iOS applications [6]. PiBox [25] provides a cross-device sandbox environment that wholly contains a user's private data on a per-app basis. While these applications are spiritually similar to TussleOS (PiBox and MockDroid in particular acknowledge the fundamental trade-off between privacy and functionality) they suffer from shortcomings. MockDroid, for instance, only allows limited sensor spoofing and full sensor access, and only applies to explicitly requested resources. PMP does not work for sensors. Although Apex introduces runtime constraints, sensor readouts are still binary. PiBox stops the leakage of private information by malicious applications, but provides much less protection when users opt to share their data. Furthermore, these systems do not provide any feedback to the user about the implications of their decisions, nor do they allow an opportunity for the application and user to negotiate privacy-functionality trade-offs directly.

ipShield [8] allows users to specify allowed and disallowed inferences within Android applications. These lists are used to maximize allowed inference accuracy and restrict disallowed inference accuracy. In practice, this proves to be a flexible, low-overhead interface for managing data privacy, but it does not take sensor sampling granularity into consideration. The amount of information revealed by sampling a sensor at two different granularities can vary significantly. For instance, an application that accesses the accelerometer to infer if the user is in motion requires a much smaller time granularity than another application that infers the user's physical activity. Because the user receives no interpretable metric conveying the amount of private information revealed, it may be difficult for them to specify which inferences to allow and disallow for a particular application. Furthermore, ipShield does not inform the user of differences in levels of functionality when modifying sensor access for an application. Nevertheless, ipShield serves as a good starting point for building other user-understandable approaches which balance privacy and functionality.

Michalevsky et al. [29] use a smartphone's MEMS gyroscope to capture users' speech. They also show that the ability of an application to capture speech is significantly impacted by the frequency at which the application is able to poll the gyroscope. This is an explicit example of a scenario which would be easily managed by a tussle-based framework. By limiting the application's access to the sensor, both the user's privacy and the application's core functionality are preserved.

10. CONCLUSION

Existing OSes lack a principled approach to handling the privacy implications stemming from newfound networked sensing and actuation capabilities. Numerous one-off systems have been conceived to patch existing OSes to handle these capabilities, but they suffer from numerous shortcomings. We advocate that OSes need to recognize privacy conflicts between different stakeholders accessing system resources, and require suitable software mechanisms and policies to detect and resolve such tussles. Using sensing applications as a starting point, we outline the design of a privacy framework which provides applications and users with high-level interfaces to express their requirements, detect conflicting requirements, resolve them, and ensure the implementation of the resolution. Furthermore, we identify various open problems that need to be solved to instantiate such an OS framework.

11. ACKNOWLEDGMENTS

Funding for this project was provided in part by Cisco Systems, the University of Waterloo President's Scholarship, the David R. Cheriton Graduate Scholarship, the Natural Sciences and Engineering Research Council of Canada (NSERC) Alexander Graham Bell Canada Graduate Scholarship - Doctoral, the NSERC Discovery Grant, and the NSERC Discovery Accelerator Supplement.

12. REFERENCES

[1] Android SDK. `http://developer.android.com`.

[2] Nest. `http://www.nest.com/`.

[3] Popular Android Apps Leak Sensitive User Data. `https://blog.kaspersky.com/privacy_holes_in_popular_android_apps/`.

[4] Raspberry Pi. `https://www.raspberrypi.org/`.

[5] The Internet of Things. `http://share.cisco.com/internet-of-things.html/`.

[6] Agarwal, Y., and Hall, M. ProtectMyPrivacy: detecting and mitigating privacy leaks on iOS devices using crowdsourcing. In *Proc. ACM MobiSys* (2013).

[7] Beresford, A. R., Rice, A., Skehin, N., and Sohan, R. Mockdroid: trading privacy for application functionality on smartphones. In *Proc. ACM HotMobile* (2011).

[8] Chakraborty, S., Shen, C., Raghavan, K. R., Shoukry, Y., Millar, M., and Srivastava, M. ipShield: a framework for enforcing context-aware privacy. In *Proc. USENIX NSDI* (2014).

[9] Chen, Z., Lin, M., Chen, F., Lane, N. D., Cardone, G., Wang, R., Li, T., Chen, Y., Choudhury, T., and Campbell, A. T. Unobtrusive sleep monitoring using smartphones. In *IEEE PervasiveHealth* (2013).

[10] Chin, E., Felt, A. P., Sekar, V., and Wagner, D. Measuring user confidence in smartphone security and privacy. In *Proc. ACM SOUPS '12*.

[11] Clark, D. D., Wroclawski, J., Sollins, K. R., and Braden, R. Tussle in cyberspace: defining tomorrow's Internet. In *ACM SIGCOMM CCR* (2002).

[12] Dixon, C., Mahajan, R., Agarwal, S., Brush, A. J., Lee, B., Saroiu, S., and Bahl, P. An

operating system for the home. In *USENIX NSDI* (2012).

[13] Egele, M., Kruegel, C., Kirda, E., and Vigna, G. PiOS: Detecting privacy leaks in iOS applications. In *Proc. NDSS* (2011).

[14] Enck, W., Gilbert, P., Han, S., Tendulkar, V., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. N. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM TOCS* (2014).

[15] Felt, A. P., Chin, E., Hanna, S., Song, D., and Wagner, D. Android permissions demystified. In *Proc. ACM CCS* (2011).

[16] Felt, A. P., Ha, E., Egelman, S., Haney, A., Chin, E., and Wagner, D. Android permissions: User attention, comprehension, and behavior. In *Proc. ACM SOUPS* (2012).

[17] Gupta, T., Singh, R. P., Phanishayee, A., Jung, J., and Mahajan, R. Bolt: Data management for connected homes. In *USENIX NSDI* (2014).

[18] Haddadi, H., Howard, H., Chaudhry, A., Crowcroft, J., Madhavapeddy, A., and Mortier, R. Personal data: Thinking inside the box. *arXiv:1501.04737* (2015).

[19] Hailemariam, E., Goldstein, R., Attar, R., and Khan, A. Real-time occupancy detection using decision trees with multiple sensor types. In *Proc. ACM SimAUD* (2011).

[20] Hao, T., Xing, G., and Zhou, G. iSleep: unobtrusive sleep quality monitoring using smartphones. In *Proc. ACM SenSys* (2013).

[21] Huynh, T., and Schiele, B. Analyzing features for activity recognition. In *Proc. sOc-EUSAI 2005* (2005).

[22] Kansal, A., Saponas, S., Brush, A., McKinley, K. S., Mytkowicz, T., and Ziola, R. The latency, accuracy, and battery (LAB) abstraction: programmer productivity and energy efficiency for continuous mobile context sensing. In *Proc. ACM OOPSLA* (2013).

[23] Karantonis, D., Narayanan, M., Mathie, M., Lovell, N., and Celler, B. Implementation of a real-time human movement classifier using a triaxial accelerometer for ambulatory monitoring. *IEEE Transactions on Information Technology in Biomedicine* (2006).

[24] Krohn, M., Yip, A., Brodsky, M., Cliffer, N., Kaashoek, M. F., Kohler, E., and Morris, R. Information flow control for standard OS abstractions. In *Proc. ACM SIGOPS Operating Systems Review* (2007), vol. 41, pp. 321–334.

[25] Lee, S., Wong, E. L., Goel, D., Dahlin, M., and Shmatikov, V. πbox: a platform for privacy-preserving apps. In *Proc. USENIX NSDI* (2013).

[26] Li, T., Rafetseder, A., Fonseca, R., and Cappos, J. Fence: Protecting device availability with uniform resource control. In *USENIX ATC '15* (July 2015), USENIX, pp. 177–191.

[27] Liu, H., Saroiu, S., Wolman, A., and Raj, H. Software abstractions for trusted sensors. In *Proc. ACM MobiSys* (2012).

[28] Madhavapeddy, A., Mortier, R., Rotsos, C., Scott, D., Singh, B., Gazagnaire, T., Smith, S., Hand, S., and Crowcroft, J. Unikernels: Library operating systems for the cloud. In *ACM SIGPLAN Notices* (2013), vol. 48, ACM, pp. 461–472.

[29] Michalevsky, Y., Boneh, D., and Nakibly, G. Gyrophone: Recognizing speech from gyroscope signals. In *Proc. USENIX SEC* (2014).

[30] Nauman, M., Khan, S., and Zhang, X. Apex: extending android permission model and enforcement with user-defined runtime constraints. In *Proc. ACM ASIA CCS* (2010).

[31] Nirjon, S., Dickerson, R. F., Asare, P., Li, Q., Hong, D., Stankovic, J. A., Hu, P., Shen, G., and Jiang, X. Auditeur: A mobile-cloud service platform for acoustic event detection on smartphones. In *ACM MobiSys* (2013).

[32] Reddy, S., Mun, M., Burke, J., Estrin, D., Hansen, M., and Srivastava, M. Using mobile phones to determine transportation modes. *ACM TOSN* (2010).

[33] Roesner, F., Kohno, T., Moshchuk, A., Parno, B., Wang, H. J., and Cowan, C. User-driven access control: Rethinking permission granting in modern operating systems. In *Proc. IEEE SP* (2012).

[34] Roesner, F., Molnar, D., Moshchuk, A., Kohno, T., and Wang, H. J. World-driven access control for continuous sensing. In *Proc. ACM CCS* (2014).

[35] Santos, N., Duarte, N. O., Costa, M. B., and Ferreira, P. A case for enforcing app-specific constraints to mobile devices by using trust leases. In *Proc. USENIX HotOS* (2015).

[36] Santos, N., Raj, H., Saroiu, S., and Wolman, A. Using arm trustzone to build a trusted language runtime for mobile applications. In *Proc. ACM ASPLOS* (2014).

[37] Scott, J., Brush, A. J. B., Krumm, J., Meyers, B., Hazas, M., Hodges, S., and Villar, N. PreHeat: Controlling home heating using occupancy prediction. In *Ubicomp* (2011).

[38] Singh, R. P., Keshav, S., and Brecht, T. A cloud-based consumer-centric architecture for energy data analytics. In *e-Energy* (2013).

[39] Singh, R. P., Shen, C., Phanishayee, A., Kansal, A., and Mahajan, R. A case for ending monolithic apps for connected devices. In *Proc. USENIX HotOS* (2015).

[40] Spahn, R., Bell, J., Lee, M. Z., Bhamidipati, S., Geambasu, R., and Kaiser, G. Pebbles: Fine-grained data management abstractions for modern operating systems. In *Proc. USENIX OSDI* (2014).

[41] Xu, W., Zhang, F., and Zhu, S. Permlyzer: Analyzing permission usage in android applications. In *Proc. IEEE ISSRE* (2013).

[42] Zeldovich, N., Boyd-Wickizer, S., and Mazieres, D. Securing distributed systems with information flow control. In *Proc. NSDI '12* (2008), vol. 8, pp. 293–308.

[43] Zhang, P. H., Li, J. Z., Shao, S., and Wang, P. Pdroid: Detecting privacy leakage on android. In *Applied Mechanics and Materials* (2014).