Software Abstractions for Trusted Sensors

He Liu[†], Stefan Saroiu, Alec Wolman, Himanshu Raj
Microsoft Research, [†]University of California San Diego

ABSTRACT

With the proliferation of e-commerce, e-wallet, and e-health smartphone applications, the need for trusted mobile applications is greater than ever. Unlike their desktop counterparts, many mobile applications rely heavily on sensor inputs. As a result, trust often requires authenticity and integrity of sensor readings. For example, applications may need trusted readings from sensors such as a GPS, camera, or microphone. Recent research has started to recognize the need for "trusted sensors", yet providing the right programming abstractions and system support for building mobile trusted applications is an open problem.

This paper proposes two software abstractions for offering trusted sensors to mobile applications. We present the design and implementation of these abstractions on both x86 and ARM platforms. We implement a trusted GPS sensor on both platforms, and we provide a privacy control for trusted location using differential privacy. Our evaluation shows that implementing these abstractions comes with moderate overhead on both x86 and ARM platforms. We find these software abstractions to be versatile and practical – using them we implement one novel enterprise mobile application.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Security Kernels

Keywords

Mobile Computing, Sensors, Trusted Platform Module, ARM TrustZone, Differential Privacy

1. INTRODUCTION

Sensor readings gathered from a smartphone have started to have high value. For example, location information is harvested to build maps of Wi-Fi access points at global scale [29]. Photos and videos taken with a smartphone are used by news media to determine abuse and human rights violations by various groups and governments [33, 17]. Blood sugar is being monitored by wireless sensors that control insulin pumps [19]. Location (e.g. geo-fencing) is now being used in mobile payment schemes [30].

As this value is rising, cloud services and mobile applications require "trusted sensors" – the ability to produce sensor readings that instill a high degree of confidence about their integrity and authenticity. Today, it is trivial to fabricate sensor readings by simply making up GPS locations, camera shots, or health readings. As long as fabricating sensor readings can be done by compromising a smartphone's software stack (e.g., deploying a piece of malware), the familiar spectrum of security miscreants will rise to exploit the value of sensor readings. Criminals will try to exploit financial transactions, steal health information for later resale, and prevent the use of sensors data for investigations or prosecutions. Recognizing the need for attesting the authenticity of sensor readings, the research community has started to describe the huge potential of such technology [2, 5, 25, 34] and to propose frameworks able to verify the authenticity of sensor readings captured and modified on a smartphone [6].

Designing the "right" software abstractions for trusted sensors is challenging and non-intuitive. For example, one way to detect tampering is to simply sign the sensor readings. Although this approach meets the definition of trusted sensors (i.e., it protects the integrity of sensor data from malicious applications), it is limited because it does not enable a common scenario where sensor readings are processed on the mobile device before uploading to the cloud [6]. For example, cropping a photo before uploading it to Facebook would invalidate the photo's signature. Signed readings is too rudimentary of an abstraction for trusted sensors because one cannot distinguish malicious tampering from legitimate application needs. Second, simply signing a sensor's reading without semantically understanding the sensor's state is insufficient and can even be insecure. For example, some GPS devices can be put in a "simulated mode" in which they simulate locations different than the actual physical location of the sensor. Finally, mobile applications might want to be able to reveal secrets based on a policy whose input is sensor readings. For example, a geo-fencing security application might want to implement access control based on GPS locations. Unfortunately, simply signing GPS locations does not meet the needs of such applications.

This paper's goal is to present two software abstractions designed to expose trusted sensors to mobile applications and cloud services. The first abstraction is called *sensor attestation* and its role is to protect the sensor reading's integrity and authenticity. This is done by attesting the code producing the reading as well as the sensor configuration (i.e., the sensor's state) when the reading was made. The second abstraction is called *sensor seal*. Sensor seal takes as input a secret, encrypts it, and binds it to a sensor policy. When an application calls sensor unseal, a sensor reading is produced and the secret is revealed only if the reading obeys the policy specified at seal time. For example, a geo-fencing application might seal authentication credentials (i.e., a "secret") to a virtual perimeter. Unseal is successful only if the location readings reveal that the smartphone is within the virtual perimeter. As their names suggest, these abstractions have been inspired from the two primitives used in trusted computing, *software attestation* and

MobiSys'12, June 25–29, 2012, Low Wood Bay, Lake District, UK.

sealed storage. Like their trusted computing counterparts, these two primitives combined are sufficient for exposing trusted sensors to mobile applications and cloud services.

We have implemented these two abstractions on both x86 and ARM SoC platforms because we believe trusted sensors are needed for both laptops (x86) and smartphones and tablets (ARM). The two implementations are similar only at a very high-level – they both implement a runtime environment protected from the OS and the applications running on behalf of the user of the device. Inside these environments, we build a software stack that implements the two trusted sensor primitives – attestation and seal – and offers them to the untrusted OS and applications. The mechanisms used to build these isolated environments are different between the two architectures. On x86, we leverage trusted computing hardware and the Credo research hypervisor [21] that offers strong isolation properties similar to CloudVisor [36], to build guest VMs that can protect data integrity even if the management VM becomes compromised. On ARM, we leverage the ARM TrustZone extensions that can protect sensor readings' integrity from all other software running on the platform. The hardware support on ARM makes our system much simpler than its x86 counterpart.

Currently, both our implementations offer attestation and seal for GPS only. To demonstrate the benefits of trusted GPS, we implemented a new mobile application, called *TrustedDrive*. With TrustedDrive, a user can protect a storage partition by defining a geo-location policy. The storage partition is mounted only if the current location returned by the trusted GPS satisfies the original policy. Such an application is useful in an enterprise environment because it ensures that sensitive partitions are available only if employees are on premise.

Trusted sensors also raise privacy concerns for two reasons: 1) the digital signature that accompanies the sensor reading may identify who signed the reading, and 2) the content of the sensor reading may reveal sensitive information about the device's owner. In this paper, we describe approaches to deal with both problems, and we implement a differential privacy layer to address the latter problem. Our first approach is based on cryptography constructs that change the nature of the signature that accompanies a sensor reading when sent to the cloud. These mechanisms can provide user anonymity and sensor data non-transferability (e.g., while the cloud can verify a sensor reading's authenticity, the cloud cannot convince others of the reading's authenticity).

Our second approach aims to protect the information revealed by the sensor data before it is signed. For this we leverage differential privacy [3, 4], a technique that provides a mathematical framework for injecting "noise" into a query answer and measuring the amount of privacy loss in answering the query. We show how differential privacy can be added as a "privacy layer" to a trusted GPS implementation. This layer handles incoming queries for GPS coordinates and answers them in a differentially private-manner. The answers remain trustworthy because they continue to carry a trusted sensor attestation. Finally, because differential privacy introduces errors in the sensor readings, such errors may affect the correctness of mobile applications that depend on the GPS. We evaluate the magnitude of such errors using a previously collected trace of GPS locations [28], and show that certain classes of mobile applications will not be affected by the noise introduced by differential privacy.

The contributions of our paper are:

- We design two trusted computing abstractions that fit the needs of a growing class of mobile applications – those that require increased assurance about the sensor data produced by a mobile device.
- We demonstrate how to implement these abstractions in a virtualized system. Using a serial port GPS device as an example, we show how to hand over ownership of a sensor device from the management VM to a separate VM whose responsibility is to provide the trusted sensor stack.
- We demonstrate how to implement these abstractions using trusted computing hardware extensions found in ARM-based systems. In particular, we show how to leverage ARM TrustZone to provide a lightweight trusted sensor stack with a small trusted computing base (TCB).
- We take some initial steps toward overcoming the privacy challenges posed by trusted sensors. We demonstrate how differential privacy can be used with a GPS sensor to offer systematic a way of reasoning about how to add noise to sensor readings to improve privacy.

2. THE NEED FOR TRUSTED SENSORS

Previous work [2, 5, 6], including our own workshop papers [25, 34], describes many mobile applications that benefit from the presence of trusted sensors, and the attacks that rise due to the lack of them [31]. Rather than re-enumerating all trusted mobile applications in this paper, instead we classify them in a higher-level taxonomy. This classification helps to build a better intuition of why these two software abstractions meet the needs of these trusted applications.

1. Applications that collect proofs. These applications need to collect sensor readings and present them as "proofs" either immediately after they are collected, or at some later time. Sometimes these proofs are used to demonstrate the authenticity of content, as is the case with validating that photos have not been photoshopped. In other cases, such proofs are used to demonstrate a particular action or behavioral pattern of the user, as is the case with offering store discounts to loyal customers based on location proofs.

2. Participatory sensing/crowdsourcing applications. These applications upload sensor readings to the cloud. The cloud often combines readings from multiple users in a process referred to as *crowdsourcing*, to build an aggregate view of data. This aggregate view is then offered as a cloud service to all users.

This class of applications is subject to database manipulation attacks [31], where an attacker uploads fake sensor readings to the cloud. With sensor attestations, it is much harder for attackers to fabricate or alter sensor readings without being detected.

3. Applications that use authentication. These applications need to perform a security-sensitive operation based on a sensor reading. For example, a user might get access to a secret file or password only when present at a specific location. Sometimes the secret might be revealed by the cloud, whereas in other cases the secret might be revealed by the local smartphone. In both cases, the secret is revealed based on the reading values of a trusted sensor.

The sensor seal abstraction makes it easy to build such applications. The secret data can be first sealed according to a particular sensor-based policy. Any application that uses authentication can just issue an unseal operation to receive the secret data.

3. THREAT MODEL

A mobile user can install malware (whether accidentally or intentionally) on their mobile device, compromising the general purpose operating system. While such malware can tamper with sensor attestations, we enable detection of such tampering through signature verification. Malware may also modify all the software that

runs on the machine (even the persistent copy of the trusted software), but such modifications will be detectable by using software attestation [32]. To provide these guarantees, we require that the trusted computing base (TCB) of our system cannot be compromised. If our system's TCB ever becomes compromised, we cannot provide any guarantees. On x86, the system TCB consists of the trusted sensors software stack plus the Credo hypervisor. On ARM, this consists of all code that runs in the ARM TrustZone secure world. Remember that, on both x86 and ARM, the general purpose OS, system services, and all third party applications are not part of the TCB.

Several classes of attacks are out of scope for our current system. As mentioned in the previous paragraph, all attacks that compromise our system's TCB are out of scope. Another class is side-channel attacks that attempt to infer the secrets (e.g., the signing key) of our trusted environment through covert channels. Another class is tampering with the trusted computing hardware, such as the TPM. TPMs were not designed to protect against physical attacks; the TPM spec does not require tamper-proof manufacturing [32]. Finally, it is possible to manipulate the physical environment to create false sensor readings. For example, keeping a lighter next to a temperature sensor will produce artificially high temperature readings.

As with all trusted computing systems, in addition to relying on the software TCB being exploit free, we must also trust the manufacturer of the trusted computing hardware. There are two aspects to this trust: 1) relying on a correct implementation of the TPM chip or TrustZone feature, and 2) relying on a secure provisioning process. To provision a TPM chip, the manufacturer injects a unique identity, known as the endorsement key-pair, into the TPM chip. The manufacturer then signs a certificate indicating that the public key of the endorsement key-pair is an authentic TPM chip. The security guarantees rely on the manufacturer's signing key remaining secret, and the private key of the endorsement key-pair remaining secret.

4. TRUSTED SENSORS ABSTRACTIONS

Trusted sensors are an important building block to building trustworthy systems in the mobile landscape. Although there is no precise definition, the term "trusted computing" often refers to systems that build upon hardware primitives to provide code integrity protection and confidentiality for their secret data. While the meaning of code integrity is well-understood, it is important to elucidate the meaning of data confidentiality. Typically, trusted computing systems are capable of protecting a piece of data in such a way that only a specific, pre-determined piece of code can access it. The combination of these two properties allows a system to guarantee that trusted code will run unmodified and will protect its secret data from all untrusted code.

Based on these insights, we borrow abstractions from trusted computing and map them to mobile sensing. The remainder of this section describes our abstractions and how they meet the needs of trusted mobile applications.

4.1 Abstraction #1: Sensor Attestation

A sensor attestation protects the sensor reading's integrity and demonstrates its authenticity. To offer these two properties, the sensor reading is signed. The key used for signing is a Trusted Platform Module (TPM)'s Attestation Identity Key (AIK) [32]. The private portion of the AIK is non-migratable and protected by the TPM. The underlying platform also binds the *same* AIK to the software configuration of the platform's TCB, and can use this AIK to sign *remote attestations*. The combination of binding the AIK to a trusted configuration and sharing the *same* AIK to sign both sensor and remote attestations provides a notion of authenticity: *this platform with this specific configuration produced this sensor reading*.

There is one additional practical requirement related to sensor attestation. The sensor attestation must also incorporate a notion of the sensor's state to ensure that software can correctly interpret how the reading was produced. In Section 5, we elaborate on this requirement.

4.2 Abstraction #2: Sensor Seal

Sensor seal protects a secret by encrypting and binding it to a *policy* that uses sensor readings. Unseal reads the sensors and evaluates a policy predicate to determine whether the sensor readings satisfy the policy. If so, the secret is decrypted. The encryption is performed using a secret storage key, similar to the storage root key (SRK) used by the TPMs [32]. As with the AIK, the private portion of the SRK is non-migratable and protected by the trusted computing hardware.

The policies used for sealing can be quite complex and span the readings of multiple sensors. We did not want to restrict application developers in the types of policies they can use. As a result, in our implementation, policies can be specified using a full-featured scripting language (we use Python in our prototype).

4.3 Local Processing of Sensor Readings

In some cases, sensor readings need to be processed on the local device before they can be uploaded to the cloud. For example, photos might be needed to be shrunk, cropped, or re-encoded at a lower resolution to reduce the bandwidth and energy costs of cloud upload [6]. Such legitimate pre-processing needs will affect the integrity of the sensor reading uploaded to the cloud.

To preserve the authenticity of these processed sensor readings, the local application code manipulating them must be able to protect its integrity at runtime. This code will first validate the sensor attestation, then process the sensor reading, and finally upload it to the cloud, all without its code being modified by an attacker. Without code integrity, the sensor readings being uploaded to the cloud could be modified by malware. Many systems for offering code integrity have been previously proposed [27, 14, 13, 26, 36] and they can all be integrated with our sensor attestations to offer end-to-end trust guarantees. In this paper, our x86 implementation supports local sensor processing because of our use of Credo hypervisor [21], as described in Sections 6 and 7.

5. TRUSTED SENSORS API

This section describes the API we use to provide the two software abstractions for trusted sensors. The abstractions are implemented by a runtime environment with strong isolation from the rest of the system, including the OS. Even in the presence of a compromised OS, applications can use our API to obtain sensor readings that protect their integrity and authenticity. The implementation of this API is described in Sections 7 and 8; this section focuses on describing the API design and how mobile applications can use it.

5.1 Sensor Attestation API

At a high-level, a sensor attestation is a sensor reading signed with an AIK [32]. To validate the reading's integrity, a verifier checks the signature using the public part of the AIK. To validate its authenticity, a verifier requests a remote attestation from the platform, using a standard remote attestation verification protocol. This last step attests the software configuration producing the sensor reading.

```
enum flash_mode_t {flash_off, flash_on, flash_auto};
enum autofocus_mode_t {focus_normal, focus_macro};
enum white_balance_t {automatic, incandescent,
                      fluorescent, daylight, cloudy};
enum scale_t {min, low, medium, high, max};
enum iso_t {isoauto, iso50, iso100, iso200,
            iso400, iso800};
enum metering_t {matrix, center_weighted, spot};

typedef struct camera_config {
     int resolution_width;
     int resolution_height;
     flash_mode_t flash_mode;
     autofocus_mode_t autofocus_mode;
     white_balance_t whitebalance;
     scale_t contrast;
     scale_t saturation;
     scale_t sharpness;
     int exposure_value;  // from -2 to +2
     iso_t iso;
     metering_t metering;
     bool wide_dynamic_range;
     bool anti_shake;
} camera_config_t;

CapturePhoto(camera_config_t *camera_config, // out
     int *image_size,                        // out
     unsigned char **image_bytes,            // out
     rsa_sig_t *signature)                   // out
```

Figure 1: Trusted Camera API. ***The settings available for this camera device are based on the Samsung Focus Windows Phone.***

We now describe how to design the API for a trusted sensor. The goal of a trusted sensor is simply to associate a digital signature with a sensor reading, to provide both authenticity and integrity for the sensor reading. Thus, the trusted sensor API must ensure that each call that returns sensor data also provides an additional out parameter for the signature, and this signature should cover all of the output data. We demonstrate the process using three example APIs: 1) for trusted Wi-Fi scanning; 2) for a trusted GPS sensor; and 3) for a trusted camera. We use three simple guidelines in turning a traditional sensor API into a trusted sensor API.

1. The developer must identify the different types of readings that a sensor can produce and the device-specific state needed to correctly interpret each sensor reading. All this device-specific state must be exposed through the trusted sensor API. For example, Figure 1 shows the API for a trusted camera. In this example, all of the camera settings that affect the resulting image, such as the flash mode and the exposure, must be explicitly represented in the API. The camera API allows a caller to read all of the relevant camera configuration state.

2. All API calls should be separated into either read or write operations – there should be no calls that mix both reads and writes into a single call. While sensor devices are typically thought of as read-only devices, most sensors support write operations in practice. A camera exposes operations that control its configuration state. These write operations are used to configure the sensor in ways that affect how the sensor readings should be interpreted.

As another example, many crowdsourced location systems use Wi-Fi scanning to determine the approximate location of a mobile device. In this example, the Wi-Fi radio acts as a sensor device, and many configuration parameters can affect the scan results. The list of channels the radio listens on, the listening duration, and the 802.11 band that the radio supports all affect which beacons will be heard during the scan. Figure 2 shows an example of the trusted sensor API that supports Wi-Fi passive scanning.

3. The signature provided with all read operations should cover

```
enum band_80211_t {80211a, 80211b, 80211g, 80211n};

typedef struct beacon_info {
   char *ap_bssid,
   char *ap_ssid,
   int  mean_rssi,
} beacon_info_t;

typedef struct sender_info {
   char *sender_mac,
   int channel,
   int mean_rssi,
} sender_info_t;

GetWifiPassiveScan(band_80211_t *band,  // out
     int *num_channels,                 // out
     int **channel_list,                // out
     int *scan_delay_ms,                // out
     int *num_beacons;                  // out
     beacon_info_t **beacons,           // out
     int *num_senders;                  // out
     sender_info_t **senders,           // out
     rsa_sig_t *signature)              // out
```

Figure 2: Trusted Wi-Fi Scanning API.

both the sensor's data and its configuration state that affects the sensor reading. One alternative that we considered but decided against was to have a separate API call to fetch the current configuration state along with a digital signature. The advantage of using one signature is that it ensures that the configuration is always sent along with the sensor reading. This ensures that configuration state needed to interpret the semantics of the sensor reading is always available to the software that needs it.

The API does not need to expose control of all device configuration state. The Wi-Fi and camera devices allow calls to modify the device state (elided from our examples), but the GPS example shown in Figure 3 does not. The GPS device does not allow selecting which satellites will be used to produce a location fix, but it does allow for reading which satellites were used in producing a given location fix.

5.2 Sensor Seal API

The API for sensor seal and unseal is shown in Figure 4. Seal takes as inputs a secret and a Python script specifying a sealing policy. For seal, the script's source code is concatenated with the secret and encrypted into a blob returned to the caller. To unseal, the caller passes in the encrypted blob. The blob is first decrypted using a storage key, similar to the TPM's storage root key (SRK), that never leaves the TCB of the system. Decryption produces both the secret and the Python script. The policy script predicate uses the API described above to obtain sensor readings and attestations. The policy script contains a predicate function that obtains the appropriate sensor readings, checks them against the policy, and if those checks pass the predicate returns true. If the predicate returns true, the system then decrypts the sealed blob and returns the secret to the caller.

6. HIGH-LEVEL SYSTEM ARCHITECTURE

The code implementing trusted sensors must be protected from the OS and the applications running on the mobile device. The code must be run in an isolated environment that offers (1) code integrity to ensure that malware cannot modify the trusted sensors' software stack, and (2) data confidentiality to protect the signing and encryption keys. This code is part of the system's TCB and

```
enum fix_t {no_fix, 2d_fix, 3d_fix};
enum fix_quality_t {invalid, gps_fix, dgps_fix,
                    pps_fix, rtk, float_rtk,
                    manual, simulation};

typedef struct sat_info {
     int sat_prn,
     int elevation_degrees,
     int azimuth_degrees,
     int snr,
} sat_info_t;

// combine output of GPGGA and GPGSA sentences to
// produce this data.
GetCurrentLocationFix(time *utc_time,      // out
     float *latitude,                      // out
     float *longitude,                     // out
     float *altitude_above_mean_sea_level, // out
     fix_quality_t *fix_quality,           // out
     fix_t *fix_type;                      // out
     int *num_satellites,                  // out
     float *height_of_geoid,               // out
     float *dilution_of_position,          // out
     float *vertical_dop,                  // out
     float *horizontal_dop,                // out
     rsa_sig_t *signature)                 // out

// use output of GPGSV sentences to produce this data
GetCurrentSatInfo(int *num_sats_in_view,   // out
     sat_info_t **sat_info,                // out
     rsa_sig_t *signature)                 // out
```

Figure 3: Trusted GPS API.

```
SensorSeal(int  secret_length,   // in
           char *secret,         // in
           int  script_length,   // in
           char *script,         // in
           int  seal_length,     // out
           char *sealed_blob)    // out

SensorUnseal(int  seal_length,     // in
             char *sealed_blob,    // in
             int  secret_length,   // out
             char *secret)         // out
```

Figure 4: Sensor Seal/Unseal API.

implements the trusted sensor APIs, the drivers needed to access the sensors, and a policy interpreter for the Python scripts needed for sensor seal.

Figure 5 illustrates the high-level architecture of our trusted sensors system. The TCB is presented at the bottom, while the OS and mobile applications are running at the top of the stack. The middle layer is used to run application code that needs to locally transform the sensor readings before uploading them to the cloud. This local processing needs to be isolated from the OS to preserve the integrity of the readings once processed. This layer is not within our system's TCB because it contains app-specific code and bugs in that code do not affect the security of our trusted sensor abstractions. However, the security of this layer is important to each application that needs to perform app-specific post-processing of sensor readings. As a result, our system needs to provide the ability to produce attestations that can be used to demonstrate the integrity of the application code used to perform this post-processing. The implementation of this layer is not a contribution of this paper, even though our system on the x86 platform does inherit this functionality from the Credo hypervisor. Instead, our system's focus is on the bottom layer of the figure – we present an implementation of the trusted sensor abstractions that remains secure even in the face of an OS compromise.

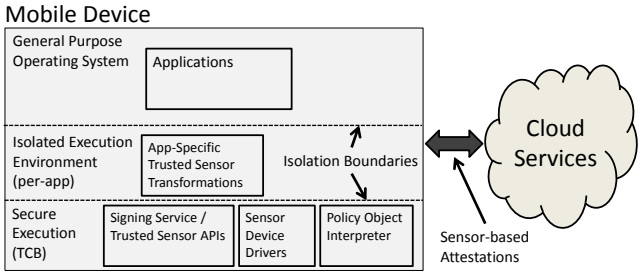

Figure 5: High-level Architecture of Trusted Sensors.

6.1 Design goals

Based on the description of the abstractions and their APIs, our system architecture has five goals.

1. Protect the integrity of the software producing the sensor attestation. Without integrity, our system cannot guarantee the properties of sensor attestation.

2. Protect the confidentiality of the attestation signing key. If the signing key for attestations were to be leaked, attackers can create their own attestations, compromising our system.

3. Protect the integrity of the sensor. It is paramount that the sensor produces correct readings and that it is not subject to erroneous configurations by untrusted software.

4. Provide support for seal/unseal for applications. To unseal successfully, the platform must check the sensor reading against a policy supplied by an application. Such checks can be complex and need to be expressed in a Turing-complete language.

5. The platform must offer adequate forms of privacy protection for mobile users' sensor data.

6.2 Design alternatives

Throughout the design of our systems (both x86 and ARM), we faced several design choices. In the remainder of this section, we describe three such choices we made and the design alternatives. The choices we made were not obvious at the time, and we believe that the alternatives are also interesting points in the trusted sensors design space.

6.2.1 Hardware vs. Software

Trusted sensors can be built both in software and in hardware, although each option presents a different set of challenges and drawbacks. A software-only implementation uses a small piece of trusted code to obtain a sensor reading and to sign that reading with a secret key to detect any tampering with the data. To be secure, this design must meet two requirements. First, the integrity of this code and the confidentiality of its secret key must be protected from all other software running on the machine. Second, the system must protect the integrity of the sensor device during the reading. Otherwise, malicious software could reprogram the sensor to cause its reading to become meaningless. Making such systems secure is challenging because the protection mechanisms must have a small TCB. For example, one option would be to rely on the general purpose OS to produce signed sensor readings. We believe such a design is insecure because OSes have large TCBs and cannot offer code integrity and data confidentiality. However, more secure designs have been proposed; some rely on hypervisors[27, 13, 36] to

offer code integrity and data confidentiality, whereas others rely on secure co-processors[14]. None of this previous work was aimed at building trusted sensors and does not meet the sensor integrity requirement.

The other alternative is building trusted sensors in hardware. For this, sensor manufacturers need to equip their sensors with additional logic to perform digital signatures. Such designs are even more secure than software-only designs because their software TCB is significantly smaller. The main drawback of building trusted sensors in hardware is the added cost – building sensors is a low margin business and adding logic to compute signatures may increase the sensor's cost. Thus, building hardware trusted sensors at large-scale has a high barrier to deployment. To avoid this barrier, we have chosen a software-only design for the systems presented in this paper.

6.2.2 Isolation mechanisms

The software stack for producing trusted sensors must run in an isolated environment to protect its code and data. Such an isolation mechanism must not rely on an operating system, because OSes have too large of a trusted computing base to provide adequate security. In practice, there are two mechanisms to build such isolated environments – using hypervisors, and using hardware support for trusted computing, such as Intel's Trusted eXecution Technology (TXT) or ARM TrustZone.

We regard using trusted computing hardware as a more secure alternative over hypervisors. Building isolation mechanisms with these features can be done with a much smaller TCB than that of a hypervisor. Unfortunately, a pure TXT environment on x86 platforms has serious performance shortcomings in practice. These limitations are well-known [14] and led others to combine the use of TXT with a hypervisor to build isolation on x86 platforms [27, 13]. In particular, the TXT environment is used to perform a measured launch of a hypervisor whose trustworthiness can be attested by the TPM. The hypervisor then provides an isolation environment whose trust is rooted in the hypervisor.

We use a similar combination of a hypervisor and Intel TXT for our isolation mechanism on x86. On ARM however, we used the ARM TrustZones, as this mechanism does not suffer from the performance limitations of TXT. As a result, our ARM-based trusted sensors system has a much smaller TCB than its x86 counterpart.

6.2.3 Privacy mechanisms

Attaching a digital signature to every sensor reading has the potential to impact users' privacy, because one must provide the corresponding public key to any entity that wants to validate the authenticity of the sensor reading. At a high level, we see two general approaches to addressing these privacy concerns: 1) using cryptographic techniques to reintroduce anonymity, and 2) using differential privacy to provide statistical information about a sensor data set. Each approach provides a very different kind of privacy – the former hides the identity of the system that generates the sensor readings, whereas the latter reduces the accuracy of individual sensor readings to protect users' privacy.

The first and simplest option is to allow selective disabling of signing the sensor readings. To accomplish this, the system must allow users to remove the attestation from the corresponding sensor reading. With our implementation this is trivial because our trusted sensing API provides the attestation as an extra field in the sensor readings. Users can simply configure their systems to return *null* in place of an attestation. While this approach is simple, users who choose this approach will lose the benefits of trusted sensors.

Another option is to use cryptographic protocols that provide anonymity and information non-transferability. Our previous work [25] advocated using group signatures [1] for anonymity combined with zero-knowledge protocols [7] for non-transferability. While these approaches continue to be viable, several reasons made us not pursue a complete implementation of these approaches. First, group signatures require setting up a trusted group manager in charge of managing group members. Compromising the group manager leads to anonymity compromises of group members. In practice, setting up a trusted group manager appears expensive and technically difficult. Second, providing non-transferability has a high performance cost because these algorithms require high computational power. In previous work, we implemented a zero-knowledge protocol whose performance overhead was about 900 ms on a Pentium 1 GHz CPU. Adding an overhead of one second to each sensor reading may be too much for certain applications.

The third option (and the one we take) is to use differential privacy – a mathematical approach to measuring the amount of privacy loss given a query and a "noise" constant. Users can design policies that specify *privacy budgets* for each application or for the entire system. Applications run their queries on a set of sensor readings until the privacy budget is exhausted. The query results are noisy – they are inaccurate so that the amount of privacy loss due to the query stays within the budget. Section 10 describes how we use differential privacy for a trusted GPS sensor.

7. IMPLEMENTATION ON X86 PLATFORMS

On x86, we use Credo to provide an isolated environment for producing trusted sensor attestations. Credo is a research hypervisor, based on Microsoft's Hyper-V, which provides guest VMs additional protections from the Root VM. First, we provide a short primer on Credo (more details can be found in [21]). Next, we describe our modifications to Credo to enable trusted sensors.

7.1 Background on Credo

Credo is a hypervisor that leverages the TPM to establish trust in hypervisor that is launched at boot time. The Credo hypervisor also offers a new type of guest VM, called an *emancipated VM*. Emancipated VMs differ from traditional guest VMs in two respects: 1) the hypervisor provides stronger isolation, and 2) Credo enables measurement of the software state of the emancipated VM. The combination of these features ensures that as long as the hypervisor and the TPM remain uncompromised, then: 1) an emancipated VM can attest its configuration to external applications and services, and 2) an emancipated VM can persist secrets in untrusted storage that can only be decrypted by that same emancipated VM. Unlike most commodity hypervisors, the trusted computing base (TCB) of Credo's emancipated VMs is only the hypervisor; the Root VM (similar to Xen's Dom0) cannot compromise the integrity and confidentiality of an emancipated guest VM.

To verify the trustworthiness of the Credo hypervisor and to protect itself from attacks in the OS pre-boot environment, Credo uses the dynamic root of trust measurement (DRTM) features available on both Intel and AMD CPUs.

To support the emancipated guest VM features, Credo performs the following two actions when booting an emancipated guest VM. First, it isolates the memory and the virtual CPU state of the guest VM from all other VMs. Credo restricts the Root VM from accessing the guest VM's memory pages and from making any intercepts that could change the guest's virtual CPU state. Second, when resuming an emancipated guest VM image, Credo records a measurement to attest the software configuration of the emanci-

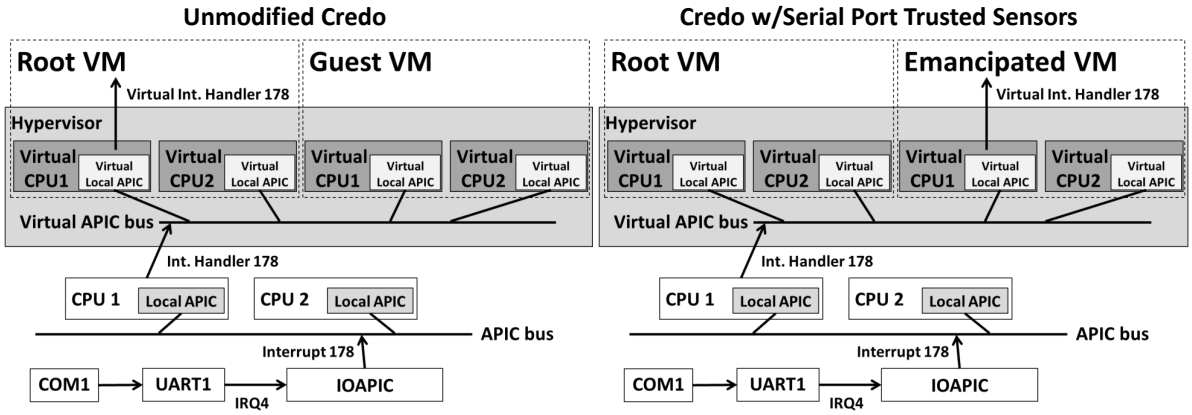

Figure 6: Interrupt Handling Architecture for the Serial Port on x86. ***On the left, the figure shows the standard interrupt handling for Credo (or any other generic hypervisor). On the right, the figure shows our modification to Credo to add support for mapping the serial port to emancipated guest VM.***

pated guest VM. This enables software running inside this emancipated guest VM to seal a secret to the measurement of emancipated guest VM's software configuration. Credo then ensures that only the guest VM whose current measurement matches the specified measurement that came with the seal operation can decrypt this secret. A similar mechanism can be used by external cloud services: they can also create secrets in way that only specific emancipated VMs with a specific software stack can decrypt them.

7.2 Modifying Credo For Trusted Sensors

Credo provides a building block for two of the design requirements for trusted sensors – code integrity and data confidentiality. We enable trusted sensors with Credo by placing the trusted sensor software stack in an emancipated VM, thereby protecting it from the Root VM which runs the general purpose OS and applications. However, Credo leaves the Root VM in control of the physical I/O devices, and uses encryption to protect data before handing it off to the Root VM on its way to an I/O device. This ensures that the Root VM is no longer part of the system's TCB. However, we need direct access to sensor I/O devices from the emancipated VM that hosts the trusted sensor software stack. To support this, we modify Credo to support mapping certain I/O devices directly into an emancipated guest VM. In particular, for our implementation we enable one form of I/O support: our current implementation uses a GPS sensor that attaches through a serial port, as a result we need to enable serial port I/O from an emancipated guest VM.

7.3 Enabling Serial Port Trusted Sensors on Credo

We begin our implementation description by presenting how the serial port is handled by Credo. Note that for the features described is this discussion, Credo is identical to Hyper-V. Also, the architecture described here is very similar to other commodity hypervisors such as VMware or Xen. The goal of our modifications to Hyper-V/Credo is to take away ownership of the serial port from the Root VM and enable the emancipated guest VM to own the serial port. To accomplish this, we first enable mapping the serial port into an emancipated guest VM (rather than to the Root VM), and then we remove the Root VM's ability to communicate with that serial port.

7.3.1 Serial Port Interrupt Handling in Credo

X86 hardware uses an universal asynchronous receiver/transmitter (UART) to control the serial (COM) port. Whenever data is available on the COM port, the UART sends an interrupt request (IRQ4) to the I/O advanced programmable interrupt controller (IO APIC). The IO APIC translates this interrupt into IRQ 178, which it then sends on the APIC bus to which all CPUs are connected. All interrupts sent on the APIC bus are handled by the local APICs, interrupt controllers within each CPU core which handle delivery of the interrupt to the CPU. Each local APIC is programmed to control which IRQ messages should be delivered to the local CPU. The hypervisor configures one CPU, designated at boot time as the master, to handle all interrupts from I/O devices that arrive on the APIC bus. This master CPU also happens to be the CPU that runs the Root VM, and therefore the full virtualization stack. While the hypervisor is handling an interrupt, lower priority interrupts are masked (blocked) until the current handler completes. When the hypervisor interrupt handler finishes, it clears the interrupt signaling that the CPU is ready to receive additional interrupts.

When the local APIC interrupts the CPU to execute the appropriate interrupt handler, the hypervisor transforms this hardware interrupt into a message sent on the *virtual APIC bus* inside the hypervisor. Each virtual core has a virtual local APIC which receives these message and passes them on to the corresponding virtual core. For each VM, the hypervisor maintains a mapping of which virtual processors are hosted on which physical cores. Interrupts destined for a particular VM are always sent to the first virtual processor for that VM. *Synthetic interrupts* sent on the virtual APIC bus are implemented in two ways: if the destination virtual processor is hosted on the same physical core, the corresponding virtual local APIC can be manipulated directly, and if not then the interrupt is sent using an inter-processor interrupt (IPI) sent over the physical APIC bus. Upon receipt of an interrupt destined for the local virtual processor, the virtual local APIC delivers the interrupt to the hypervisor, which then dispatches it to the first virtual processor of the destination guest VM. By default, all I/O devices, including the serial port, are always owned by Root VM. On the left, the figure 6 shows a diagram of the standard serial port interrupt handling in Credo.

7.3.2 Interrupt Handling for Trusted Sensors

We modified the interrupt handling in Credo to enable mapping the serial port into an emancipated guest VM, rather than the Root VM. The reason for this is that we want to take away ownership of the serial port from the Root VM, and enable an emancipated guest VM to own the serial port. To accomplish this, we modified the

virtual local APIC implementation in the Credo hypervisor. When an interrupt arrives, the hypervisor looks up which physical processor hosts the virtual processor 1 for the emancipated guest VM. It then uses the synthetic interrupt mechanism described in the previous section to deliver the interrupt to the correct virtual processor. The hypervisor then forwards the serial port interrupt up into the emancipated VM whose serial port driver handles it. In summary, we simply implement a bypass mechanism whereby interrupts for trusted sensor devices are delivered to the emancipated VM rather than the Root VM. The modifications to the hypervisor to enable this are quite simple, requiring less than 100 lines of code to implement.

7.3.3 Handling Serial Port I/O in Credo

The Root VM uses port-mapped I/O to configure the serial port and to perform actual I/O operations on the serial port. To enable port-mapped I/O from VMs, the hypervisor maintains an IO intercept bitmap that determines which of the port-mapped I/O addresses are trapped for each VM. These intercepts occur when the a VM executes any CPU instruction (such as IN or OUT) that accesses a port. By default, all port-mapped IO addresses are trapped for normal guest VMs, but for the Root VM only the address range corresponding to keyboard controller is trapped. As a result, the Root VM can use IN and OUT instructions to directly access the serial port UART.

7.3.4 Serial Port I/O for Trusted Sensors

To disable access to the serial port from the Root VM, we modify the IO intercept bitmap in the hypervisor so that the COM port IO addresses (0x3F8 to 0x3FF) are added to the Root VM's intercept bitmap. When the hypervisor traps these instructions, we modify the hypervisor so that all reads on these ports return 0xFF, and all writes are simply discarded. We chose this behavior because it matches the behavior of a serial port when no device is attached, and as a result this tells the Root VM that no device is present.

The final step is to enable the emancipated guest VM to access the serial port. To enable this, we modify the hypervisor to emulate all the IN, OUT, and OUTS instructions. We did not need to emulate the INS instruction because the serial port driver does not use that instruction. We chose to emulate, rather than modifying the guest VM IO intercept bitmap because it made our implementation simpler and because the overhead of emulation is not a problem at the low bitrates supported by the serial port.

7.4 Building Sensor Attestation and Seal

We use an emancipated guest VM to host the software that controls the sensor device and produces signed readings. Inside this VM, we run a 64-bit version of the Windows Preinstallation Environment (WinPE) created with the Windows Automated Installation Kit. WinPE is a minimal but fully functional Windows 7 environment with the full set of standard Windows 7 libraries. Because the software that runs in the emancipated guest VM is part of our TCB, it is critical to make the size of these components as small as possible. While WinPE has a much smaller TCB than a full installation of Windows 7, the TCB is still relatively large. In the future we could replace this with a much smaller software stack that is similar to the one we use for our ARM TrustZone implementation described in Section 8. The initial image for the emancipated VM that contains the trusted sensor software stack (and the measurement of that image) is created in a trusted environment to ensure that malware does not enter into the emancipated VM.

When the emancipated VM boots for the very first time, it creates a new RSA-1024 key-pair for signing, and uses the TPM's AIK to produce a *quote* for the public key of that key-pair. The reason for this step, rather than just directly using the TPM's AIK, is that it improves the performance of our sensor attestation operations. This is because RSA crypto operations on the main CPU are much faster than using the TPM for these operations. Further, the TPM quote includes relevant platform state which can be used to remotely attest that the Credo hypervisor has been launched using TXT, and that the VM is executing in emancipated mode. We also create a new symmetric storage key (an AES key) that will be used to implement sensor seal and unseal. We use the TPM's *seal* operation to protect this storage key and the private half of the signing key-pair. Finally, we save these items together on external storage: the sealed blob of keys, the TPM's AIK certificate, and the quote of the key-pair's public key.

Each time the emancipated VM boots, we initialize the network, and start the trusted sensors service. To support sensor seal and unseal, we also initialize a python interpreter inside the emancipated VM. Next, the trusted sensor service uses the TPM to unseal the blob of keys. Because the TPM seal operation includes the platform state (in a manner similar to the TPM quote), when the unseal is successful this ensures that: 1) the Credo hypervisor is running securely, 2) the VM is running emancipated, and 3) *this* emancipated VM performed the original seal operation. Finally, our trusted sensors service starts listening on a TCP port for requests to read a trusted sensor.

To handle incoming requests, our trusted sensor service reads from the specified sensor hardware, signs the readings with the private signing key, and replies with the signed data. We sign a SHA-1 hash of the data using the private RSA key. While the private signing key is kept confidential, the public key, the TPM's AIK certificate, and the TPM quote of the public key, are all exported to callers. This enables external programs to verify the integrity and authenticity of sensor readings produced by this VM.

8. IMPLEMENTATION ON ARM PLATFORMS

ARM TrustZone is ARM's hardware support for trusted computing. It is a set of security extensions found in many recent ARM processors (including Cortex A8 and Cortex A9). ARM TrustZone provides two virtual processors backed by hardware access control. The software stack can switch between the two states, referred to as "worlds". One world is called *secure world* (SW), and the other *normal world* (NW). With TrustZone, the entire OS and all applications run in the normal world, whereas a small trusted kernel runs in the secure world. The secure world provides code and data integrity and confidentiality because untrusted code running in the normal world cannot access the protected resources, such as memory pages and peripheral registers of the secure world.

An ARM platform first boots into the secure world. Here, the system firmware provisions the entire runtime environment of the secure world. Once provisioning is complete, the secure world yields to the normal world where the firmware loads up the bootloader which then loads the OS. The normal world must use a special ARM instruction called *smc* (secure monitor call), to call back into the secure world. When the CPU executes the *smc* instruction, the hardware switches into the secure monitor, which performs a secure context switch into the secure world.

Hardware interrupts can trap directly into the secure monitor code, which enables flexible routing of those interrupts to either world. This allows sensors (or other I/O devices) to map all their interrupts to the secure world and to protect the integrity of those sensors from untrusted code. However, this step is not sufficient

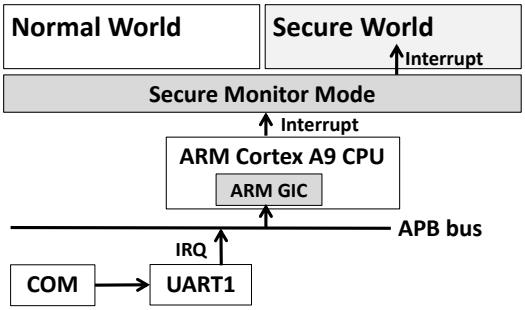

Figure 7: Architecture of handling the serial port on ARM. ***Overview of how the serial port interrupts are handled on the ARM platform.***

to fully protect I/O devices, because untrusted code executing in the normal world could potentially reprogram the device to disable it from generating interrupts, even if those interrupts are routed to the secure world. This is possible only if the normal world can memory-map the device and therefore access its control registers.

In this context, there are two approaches to secure access to a sensor device. One approach uses hardware configuration: the AXI-to-APB bridge reads an input signal that indicates which devices attached to the APB bus are only accessible from secure world, and these devices cannot be memory-mapped by the normal world. For the other approach, the device can be memory-mapped into both worlds. Thus, when the secure world starts executing, it must reset the device to a known good state, and then it must not relinquish control to the normal world until the sensor reading is complete.

8.1 Trusted GPS on ARM TrustZone

Before we describe the Trusted GPS implementation, we provide a brief description of the software stack we developed for the ARM TrustZone secure world, which we call the Secure-world Execution Environment (SEE). SEE is composed of three parts - a kernel, a runtime environment, and secure services. Trusted GPS is implemented as one secure service inside SEE.

Our kernel, which includes the secure monitor mentioned above, implements context switching between normal world and secure world, and dispatching of secure service requests from the NW to the appropriate service. Service requests are made via the *smc* instruction. The kernel also implements interrupt handling – for non-secure interrupts, the context is switched back to NW where they are handled by the NW operating system. This mechanism is also used to make the kernel and secure services preemptible. However, any time a secure service is preempted it returns to the NW with a special *continue* status. The NW is responsible for retrying the smc instruction to let the service continue execution. Since the SEE currently supports only a single context, the NW must serialize access to the SW among multiple secure service calls - it must ensure that a single secure service call finishes execution before allowing a new secure service call to start.

The SEE provides minimal runtime support in form of libraries to build secure services - in particular, it provides: **platform support**, such as a dynamic memory allocator, a clock, and secure storage; and **crypto support**, realized via a port of the OpenSSL library to the SEE.

Each secure service is uniquely identified using a 32-bit service identifier (SID). All secure services implement well defined entry points that are uniquely identified by 32-bit method identifiers (MID). These methods are used as callback routines by the dispatcher. Every secure service call from the NW identifies the SID and the MID as parameters. These parameters are passed in CPU registers r0 and r1, respectively. A secure service call can pass two additional service specific parameters using registers r2 and r3. The return code from the service call is returned in register r0. This calling convention matches the C calling convention for ARM, and allows the NW to wrap the smc assembly instruction with the following signature.

```
int SecureCall (int SID, int MID, int param0, int param1)
```

Besides registers, secure services use shared memory to communicate with the caller in the NW. To facilitate this communication, all of the NW memory is mapped in the SW with identity mapping (virtual address is same as physical address).

The trusted GPS secure service implements the trusted GPS API described earlier. For both functions, param0 provides the physical address of a shared memory buffer, and param1 provides the size of this buffer. The *GetCurrentLocationFix* secure service call returns a signed marshalled LocationFix structure in the buffer; the *SatelliteInfo* secure service call returns a signed marshalled SatInfo structure in the buffer.

Our SEE implementation is based on the Tianocore UEFI firmware [10]. We added the SEE functionality (the kernel and secure services) to the UEFI initialization "Security" phase that executes in the SW, and we implemented a trampoline to execute the remaining the UEFI phases (the Pre-EFI phase, the Driver Execution Environment (DXE), and the UEFI applications) in the NW. An example of a UEFI application is a bootloader that boots the normal world OS. Once the system starts executing in the NW, it can only interact with the SW by using the smc instruction.

Our SEE implementation provides an embedded model of secure service development, where services and the SEE kernel are compiled together to form a single binary executable image for the firmware. This firmware image is copied to the flash storage via a platform specific firmware utility. To ensure SEE integrity, the SEE firmware image is signed. Upon platform reboot, the first stage firmware loader verifies this signature before starting the SEE.

Figure 7 shows an overview of how serial port I/O and interrupts for the trusted GPS sensor are handled on the ARM platform. As with the x86 implementation, our GPS device is connected to the ARM system through a UART. Because our ARM development board does not setup the UART device as a secure-world only peripheral, this means that software running in the NW can access the UART's control registers and disable it generating interrupts. Therefore, we cannot rely on interrupts being delivered to the secure world. As a result, to protect the UART we must use the second approach we described in the previous section: our trusted GPS secure service uses polling to obtain a complete GPS reading, during which time the SW never yields to the NW.

9. EVALUATION

In this section, we present performance results from the prototype trusted sensor systems we have built. This includes our prototype running on an x86 system with a TPM chip, and our prototype running on an ARM SoC using ARM TrustZone. We also present performance results from the Trusted Drive application that we have built. Overall, our goal is simply to demonstrate that the performance overhead of the trusted sensor mechanisms do not introduce overhead that would be a significant barrier to deployment.

9.1 Experiment Platforms

We perform our x86 experiments on a system with an Intel Core2Duo E6850 CPU running at 3.0 GHz, with 4 GB of RAM.

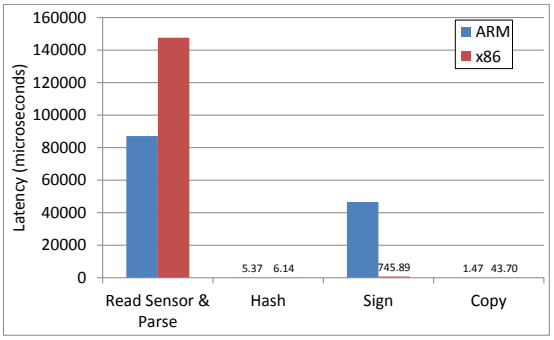

Figure 8: Latency Breakdown on ARM & x86 Platforms. ***Shows the latency of reading and parsing the sensor output, then hashing, signing, and copying the results.***

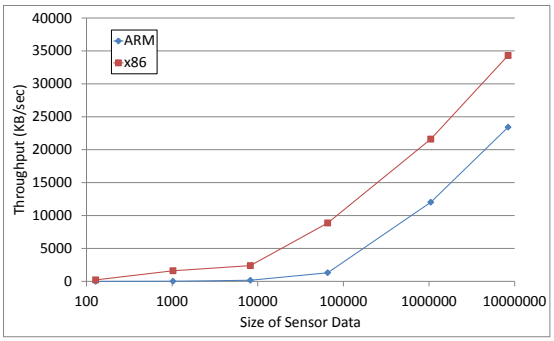

Figure 9: Sensor Attestation Throughput on ARM & x86 Platforms. ***Shows the throughput of sensor attestation for different sizes of sensor readings.***

Our Intel machine uses a Broadcom TPM 1.2 chip. Our ARM SoC platform prototype is built on an ARM SoC development board. This system contains 4 ARM Cortex A9 cores running at 1.2 GHz, with 2 GB of RAM. Our secure execution environment in the ARM TrustZone implementation uses only a single Cortex A9 core running in the TrustZone secure world, with 32 MB of RAM dedicated to the secure world. All of the individual points plotted in Figures 8 through 11 are the mean of 10 individual experiments.

9.2 Trusted Sensor Overhead

We look at three different aspects of trusted sensor overhead: the latency of each step in sensor attestation, the relationship between the size of the sensor reading and the performance overhead of signing, and the throughput of sensor-based seal and unseal.

In Figure 8, we break down the latency overhead of sensor attestation into the overhead of reading and parsing the sensor output and the overhead of hashing, signing, and copying the data back to the application. For these experiments, the reading is a 48 byte reading from the GPS sensor. We perform this experiment on both x86 and ARM, and our results demonstrate that the signing overhead is reasonable compared to the overhead of reading the sensor. Note that the overhead of signing on the ARM platform is surprisingly large (46 ms), but this is due to an unoptimized RSA implementation and the fact that our L2 cache controller is currently disabled in the SEE environment. We tested an optimized implementation of RSA in the general purpose OS on the same hardware, and it improved signing performance by more than a factor of 4.

We also observe that the overhead of reading the GPS device is noticeably larger on the x86 platform. This is because of our use of

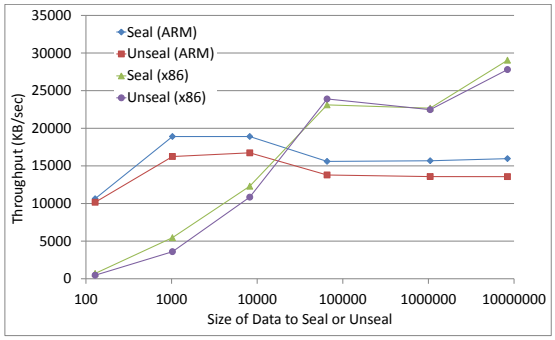

Figure 10: Sensor Seal & Unseal Throughput on ARM & x86 Platforms. ***Shows the throughput of sensor seal and unseal for different sizes of data.***

virtualization on the x86 platform. This additional overhead arises from instruction emulation that occurs for all port-mapped I/O, as well as higher interrupt latency.

In Figure 9, we show the overhead of hashing the sensor reading, signing the hash, and copying the data as a function of the size of the sensor reading. In this graph, we omit the cost of reading the GPS sensor because we want to demonstrate the overhead of trusted sensors for high data rate sensors. Our GPS is a low data rate sensor that uses a UART interface which does not support DMA transfers. Such an interface is inappropriate for high data rate sensors. On both platforms, we see that the performance scales well because only the hashing and copying steps incur additional overhead as the size of the data grows.

Finally, in Figure 10, we show the performance overhead of sensor-based seal and unseal. The results in this graph show different components for our x86 and ARM implementation. On x86, as described in Section 7.4, our benchmark application runs in the Root VM and it communicates with the trusted sensor environment in the emancipated guest VM using a TCP connection. On ARM, as described in Section 8, our benchmark application runs in the normal world, and it communicates with the trusted sensor environment using a shared memory interface.

On both platforms, the overhead includes the performance overhead of copying combined with either AES encryption or decryption. Our results are already reasonably fast, and newer x86 platforms include AES CPU instructions which will improve performance even more. ARM SoC platforms often include custom hardware accelerators for AES, but our currently implementation does not yet take advantage of such improvements. In Figure 10, the ARM performance results show higher throughput when the size of the data is less than the L1 cache size of 32 KB, and slightly lower throughput for sizes greater than 32 KB. The x86 performance results show increasing throughput (due to TCP slow start) from 128 bytes up to 64 KB, and then the throughput flattens out at sizes larger than 64 KB.

9.3 Trusted Drive Application

The trusted drive is a trusted sensor application that manages an encrypted disk drive. The goal of this application is to avoid accidental disclosure of sensitive data, such as customers' personal information. To accomplish this, the trusted drive application ensures that the drive is only accessible (mounted) when the user's mobile device is in a trusted location, such as when the user is on the campus of a large enterprise. This provides a geo-fenced file system: the file-system is automatically mounted when the user is

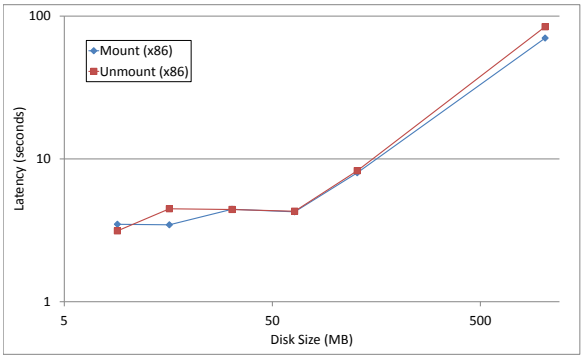

Figure 11: Trusted Drive Latency on x86 Platforms. ***Shows the latency of using the trusted drive application to mount (unseal) and unmount (seal) a trusted drive, as a function of the size of the drive.***

inside the fence, and automatically unmounted when the user ventures outside the fence.

Our scenario for this application makes two assumptions. First, we assume that users are much more likely to acquire malware outside of the trusted location. If malware infects the system while the trusted drive is mounted, our application offers little protection because the general purpose operating system can read data from the decrytped drive. Second, we assume that the mobile device implements an automatic anti-malware check whenever the device re-enters the trusted location. This ensures that the disk is not re-mounted if malware has infected the system.

To implement trusted drive, we use our trusted GPS sensor and the sensor seal abstraction. To ensure that the drive can only be accessed when the sensor reading shows the correct location, it utilizes the sensor seal and unseal abstraction to encrypt and decrypt the trusted drive. A sensor policy script is used to specify the set of locations where the drive should be enabled. This policy script is evaluated within the emancipated VM, and the drive will be unsealed if the script determines that the location falls within the specified boundaries.

The trusted drive application consists of a service that runs in the Root VM and manages mounting and unmounting the trusted drive. When the drive is not mounted, the trusted drive service attempts to mount the drive when the user attempts to access the drive. To mount the drive, the app calls unseal which will decrypt the drive as long as the policy allows it based on the current location. When the drive is mounted, the service periodically checks the sensor readings, and immediately unmounts and seals the drive when the user leaves the trusted location. Figure 11 shows the performance of mounting and unmounting the trusted drive as a function of the drive size. Our results show that all mount and unmount operations complete in less than 100 seconds.

10. DIFFERENTIAL PRIVACY

Attaching signatures to sensor data has the potential to negatively impact users' privacy, because the sensor readings produced by a device are signed by the same entity. Previous research [8] has shown that manipulating sensor readings (e.g., making a GPS location more coarse-grained) can be used to offset some of this privacy loss. With trusted sensors, any manipulation of sensor readings must occur *before* the readings are signed. In this section, we examine the potential of implementing a differential privacy layer inside the trusted sensors stack. Differential privacy offers a principled approach to manipulating sensor readings in a way that allows the system to measure how much information is revealed, and therefore how much privacy is lost, in answering a given query. Our initial work on differential privacy is limited to the GPS location sensor. We do not address how to apply differential privacy to all sensors: similar techniques may work well for an accelerometer, but it is not obvious how to apply differential privacy to the sensor readings produced by a smartphone camera.

10.1 Brief Primer

Differential privacy [3, 4] provides an intuitive formalization of privacy. Given a dataset and a query, differential privacy measures how much information is revealed by answering the query. Information is revealed when an attacker who knows the query answer is more likely to guess the existence of a data item in the dataset. Any query answered on the dataset leaks some information, however certain queries leak more information than others.

The amount of privacy loss is controlled by injecting *noise* in the query answer. Differential privacy frameworks expose a *noise knob* – if set to "high", the query answer has low privacy loss, but it is also more inaccurate, and vice-versa. Noise is generated dynamically for each query answer; if the same query is repeated, the answer changes from one run to another based on the random noise. Answering the same query twice increases the amount of privacy lost. To see why this is true, consider answering the same query repeatedly. Eventually, the attacker could infer the true query answer by looking at the distribution of query answers and factoring out the noise. Caching provides a practical way to answer repeated queries without losing more privacy. With caching, the query is not re-executed on the dataset, and as a result there is no additional privacy loss by returning the cached result. An attacker is no more likely to guess anything about the original dataset through caching.

10.2 Setting Up Differential Privacy in a System

Differential privacy is used as a *privacy layer* on top of the sensitive data. This layer's role is to manage the data produced by a sensor and answer incoming queries using differential privacy. Mobile applications must now issue queries to this privacy layer when making sensor readings. For example, a mobile application might ask for the user's average GPS location over the last hour. Whenever a query is issued to the system, this layer measures the degree of privacy lost by answering the query and decides whether to execute the query on the data. In this context, an implementation of a differential privacy layer requires three parameters.

The first parameter is P, the privacy budget of the entire dataset. The privacy lost by each query answer is deducted from this privacy budget. Once it reaches zero, the system refuses to answer any additional queries on this particular dataset. The data *owner* must set the value of P.

The second parameter is ϵ, the noise knob of a query. If set to "low", the query might disclose a lot of information and thus be rejected by the privacy layer because there is not enough budget. By changing ϵ to high, the query may be answered because there is enough privacy budget left for it. The issuer of the query controls setting this parameter. There is nothing security sensitive about controlling this parameter; it merely reflects how "interested" the query issuer is in a more "correct" query answer in exchange for spending more of the budget.

The final parameter is Δ, the *epoch* duration. After the privacy budget of a dataset has been exhausted, no future queries can be answered (except for the cached queries), putting the system into live-lock. A system in which new data is continually produced, as

is the case for sensors, can overcome this situation by partitioning the data into epochs and assigning a separate privacy budget to each epoch.

10.3 Differential Privacy for Trusted Sensors

Differential privacy can be added to the sensor attestation abstraction. Although differential privacy makes the sensor readings "noisy", the readings' integrity remains protected by our system. This means that applications receive trusted sensor readings whose values are altered *due to differential privacy alone, and not due to malware.* With differential privacy, upon an application call, the trusted API cannot return raw sensor readings anymore because they are blocked by the privacy layer. Instead, applications can submit queries over the sensor readings which are then evaluated and executed by the differential privacy layer. Query results are returned together with attestations that validate their integrity. The privacy layer implementation must be part of the system's TCB. For illustration, our privacy layer is implemented as part of the Policy Object Interpreter module shown in Figure 5.

Our differential privacy layer is based on the Privacy Integrated Queries (PINQ) [15] language for writing differentially-private queries. PINQ supports a number of LINQ-based aggregators *Count, Sum, Average, Median* and transformations *Where, Select, Distinct, GroupBy, Join, Concat, Intersect,* and *Partition*. Any query expressed in terms of these LINQ-based aggregators can be answered by our privacy layer. Although less complete than the full SQL language, the combination of these aggregators and transformations is quite versatile.

The differentially private trusted sensor API is extended to provide an interface for PINQ queries. For example, suppose an application wants to know how often a user was at a particular geofenced location, say latitude between "-95.2" and "-95.1" and longitude between "29.29" and "29.30". This computation can be expressed as a PINQ query as shown below. Parameters in the body of the query, such as GetCurrentLocationFix in this example, correspond to the types of the output parameters defined by the original trusted API. In this example, the amount of the privacy budget to spend on this query is *epsilon*.

```
locFixes = new PINQueryable<GetCurrentLocationFix>();
locFixes.Where(fix => fix.latitude > -95.2 &&
                      fix.latitude < -95.1 &&
                      fix.longitude > 29.29 &&
                      fix.longitude < 29.3)
        .Count(epsilon);
```

10.4 Evaluation

Applying differential privacy to mobile sensor applications creates a trade-off between privacy and accuracy loss. Here, we look at whether mobile applications can continue to offer their functionality when their sensor inputs are subject to differential privacy. We examine two mobile applications that use GPS location. Our first application is centered on people's commute and measures the distance mobile users travel during a day. The second application applies clustering to users' locations, a common building block for context-based systems [35]. We use a real GPS dataset to drive these applications, both with and without differential privacy, and we compare the accuracy losses.

10.4.1 Methodology

The GPS traces used were gathered by a different research group [28]. They were obtained from 34 iPhone 3GS users over the course of one year for 24 of them, and half a year for the remaining 10. The GPS reads were gathered every 15 minutes as long as the phone was not turned off (the logging process was continuously run in the background). Each experiment was run with three different settings for ϵ of 0.1, 1.0, and 10.0, settings commonly used in other systems [16]. A setting lower than 0.1 is considered very strong privacy-wise, whereas a setting higher than 10 is considered weak. We are not advocating specific levels of differential privacy as sufficient, but are instead interested in understanding the trade-off between accuracy and privacy.

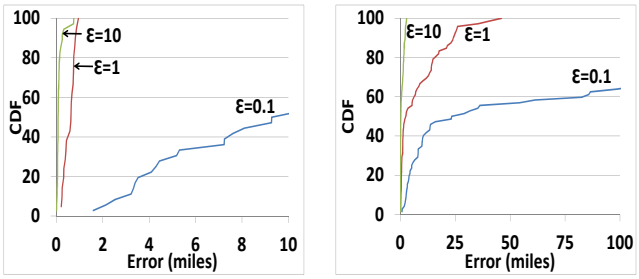

Figure 12: CDFs of error introduced by differential privacy. *On the left, the graph shows the error for the commuting distance application; on the right, the graph shows the error for inferring places of interest.*

10.4.2 Results

For each user, we measure their daily commute by summing up the distances between each successive GPS reading in the trace, and computing the average daily commuting distance. We ignore all days with less than two GPS readings because we cannot compute a distance. On the left, Figure 12 shows the cumulative distribution function of the error in the average commute distance of all users measured in miles as a function of ϵ. With high levels of noise ($\epsilon = 0.1$), half of the users' commutes have an error higher than 10 miles. However, even with a moderate setting of noise levels ($\epsilon = 1$), the error never becomes greater than one mile.

To infer places of interest, we applied 2-means clustering to each user's GPS location history. Each cluster is likely to be centered around a place of interest for that user, such as their home, workplace, or school. We then compute the error in the location of the cluster center introduced by running this algorithm with differential privacy. On the right, Figure 12 shows the cumulative distribution function of these errors measured in miles as a function of ϵ. With high levels of noise ($\epsilon = 0.1$), 40% of cluster centers have an error of more than 80 miles, whereas with $\epsilon = 1$, the error introduced by differential privacy is always less than 3 miles. With a setting of $\epsilon = 1$, the error is within 16 miles 80% of the time, with a worst-case error of 46 miles. Such errors are tolerable if the application is trying to infer the city or town for the place of interest. On the other hand, if a more precise location is needed, then a lower level of noise must be used ($\epsilon = 10$).

Our preliminary results indicate that differential privacy has potential for protecting users' privacy with a trusted location sensor. Moderate noise does not appear to be a hindrance for the distance-based mobile application, and for coarse-grained inference of places-of-interest. Our analysis, however, is only the first step in understanding the potential of differential privacy, and the key question is how to apply it to a broad range of sensors.

11. RELATED WORK

1. Trusted sensors. Much recent work motivates the need of mobile applications to obtain sensor readings with a high degree of integrity and authenticity. Some sensors, such as GPS or Wi-Fi scanning, produce their readings by contacting an infrastructure,

such as GPS satellites or Wi-Fi access points. In these cases, one alternative is for the infrastructure to sign the sensor readings, a technique sometimes referred to as "location proofs" [11, 24]. On the surface, such an approach seems to not require any changes on the client-side; as long as the infrastructure can sign its readings, mobile devices can collect them as proofs. However, such a design makes these proofs easily transferable from one device to another. A transferable proof has little value because it is unclear which device obtained the proof. Making proofs non-transferable requires devices to run client-side software that manages device identities; the infrastructure can then issue the proof to a specific device identity. Also, the presence of identities escalates the privacy concerns [12, 20].

The other approach is to rely on client-side mechanisms to increase the security of sensor readings. Most of this previous work only motivates the need for trusted sensors without describing an implementation [2, 5, 25, 34]. The piece of work closest to ours is YouProve [6]. YouProve generates a fidelity certificate for a sensor reading that has been manipulated by an untrusted application. A fidelity certificate describes how "close" the post-processed sensor reading is to the original one. One of YouProve's main challenges is coming up with a robust and complete set of techniques to characterize the loss of fidelity when applying any arbitrary transformation to a sensor reading. Furthermore, the set of techniques used for measuring fidelity loss is different from one sensor to another. Instead, our abstractions attest the code manipulating the readings and leave it up to the verifier to decide what pieces of code it trusts. Such a verification model is more closely aligned with the techniques used by trusted computing.

2. Trusted computing systems. Another area of related work is building systems that offer code integrity and data confidentiality for applications. Many of these systems can be extended to incorporate sensor I/O in their TCB. With such extensions, these systems could build and offer our two trusted sensor abstractions. Although we could have used a number of these systems in our x86 implementation, we chose to use Credo for several reasons. First, Credo offers an entire virtual machine as a secure runtime environment rather than just a hypervisor (e.g., TrustVisor [13]), a barebones TXT environment (e.g., Flicker [14]), or a new operating system (e.g., Nexus [26]). This makes programming in Credo much simpler. Credo's properties are similar to those of CloudVisor [36], but CloudVisor does not appear to be available for download.

On ARM, the Trusted Language Runtime (TLR) [23] runs parts of a mobile application inside the ARM TrustZone. We did not use TLR because its implementation is only partly done, and it has no support for I/O. Another related project implemented remote attestation on an Android platform [18]; however, this system does not appear to support running code in an isolated environment.

3. Differential privacy. Differential privacy is relatively new [3, 4] and a few recent systems have started to adopt it [22, 16]. Despite its theoretical guarantees, recently it has been shown that current implementations of differential privacy [22, 15] are subject to side-channel attacks [9]. While it is believed to be difficult for these attacks to reveal secrets, it appears possible to consume more than the pre-allocated privacy budget.

12. CONCLUSIONS

This paper presents two software abstractions for offering sensor readings to trusted mobile applications. With these abstractions, mobile applications can verify the integrity and authenticity of data produced by sensors. The paper presents two implementations of these abstractions one for x86, and one for ARM. Each of these implementations leverage the trusted computing mechanisms appropriate for each hardware platform, TPMs and hypervisors for x86, and ARM TrustZone for ARM. Finally, the paper presents a performance evaluation of these two implementations, and starts examining the potential of using differential privacy for trusted mobile applications.

13. ACKNOWLEDGEMENTS

Thanks to our shepherd, Lakshmi Subramanian, and the anonymous MobiSys reviewers for their helpful feedback and suggestions to improve this paper. We also wish to thank Landon Cox and Bryan Parno for their comments on earlier drafts of this paper. Thanks to Lin Zhong for making the Rice location traces available to us, and thanks to Stefan Savage and Geoff Voelker for supporting Lonnie's work on this project after his MSR internship was finished.

14. REFERENCES

[1] D. Chaum and E. van Heyst. Group Signatures. In *Proceedings of the 10th EUROCRYPT*, 1991.

[2] A. Dua, N. Bulusu, and W. Feng. Towards Trustworthy Participatory Sensing. In *Proceedings of the 4th HotSec*, August 2009.

[3] C. Dwork. Differential Privacy. In *Proceedings of the 33rd ICALP*, 2006.

[4] C. Dwork, F. McSherry, K. Nissim, and A. Smith. Calibrating Noise to Sensitivity in Private Data Analysis. In *Proceedings of the 3rd IACR Theory of Cryptography Conference*, March 2006.

[5] P. Gilbert, L. Cox, J. Jung, and D. Wetherall. Toward Trustworthy Mobile Sensing. In *Proceedings of the 11th HotMobile*, 2010.

[6] P. Gilbert, J. Jung, K. Lee, H. Qin, D. Sharekey, A. Sheth, and L. Cox. YouProve: Authenticity and Fidelity in Mobile Sensing. In *Proceedings of the 9th SenSys*, November 2011.

[7] S. Goldwasser, S. Micali, and C. Rackoff. The knowledge complexity of interactive proof-systems. In *Proceedings of the 17th STOC*, May 1985.

[8] M. Gruteser and D. Grunwald. Anonymous Usage of Location-Based Services through Spatial and Temporal Cloaking. In *Proceedings of the 1st MobiSys*, May 2003.

[9] A. Haeberlen, B. C. Pierce, and A. Narayan. Differential privacy under fire. In *Proceedings of the 20th USENIX Security Symposium*, August 2011.

[10] Intel. Intel UEFI implementation codenamed Tianocore. `http://tianocore.sourceforge.net`, last accessed December 2011.

[11] V. Lenders, E. Koukoumidis, P. Zhang, and M. Martonosi. Location-based Trust for Mobile User-generated Content: Applications, Challenges and Implementations. In *Proceedings of the 9th HotMobile*, 2008.

[12] W. Luo and U. Hengartner. Proving Your Location without Giving up Your Privacy. In *Proceedings of the 10th HotMobile*, 2009.

[13] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. TrustVisor: Efficient TCB Reduction and Attestation. In *Proceedings of IEEE Symposium on Security and Privacy*, May 2010.

[14] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An Execution Infrastructure for TCB Minimization. In *Proceedings of EuroSys*, 2008.

[15] F. McSherry. PINQ Library. http://research.microsoft.com/en-us/projects/pinq/.

[16] F. McSherry and R. Mahajan. Differentially-Private Network Trace Analysis. In *Proceedings of SIGCOMM*, 2010.

[17] MediaWatch. Beware the "trusted" source. http://www.abc.net.au/mediawatch/transcripts/s3218415.htm, 2011.

[18] M. Nauman, S. Khan, X. Zhang, and J.-P. Seifert. Beyond Kernel-level Integrity Measurement: Enabling Remote Attestation for the Android Platform. In *Proceedings of the 3rd TRUST conference*, June 2010.

[19] N. Paul, T. Kohno, and D. C. Klonoff. A Review of the Security of Insulin Pump Infusion Systems. *Journal of Diabetes Science and Technology*, 5(6):1557–1562, November 2011.

[20] K. P. N. Puttaswamy and B. Y. Zhao. Preserving Privacy in Location-based Mobile Social Applications. In *Proceedings of the 10th HotMobile*, 2009.

[21] H. Raj, D. Robinson, T. Tariq, P. England, S. Saroiu, and A. Wolman. Credo: Trusted Computing for Guest VMs with a Commodity Hypervisor. Technical Report MSR-TR-2011-130, Microsoft Research, 2011.

[22] I. Roy, S. T. V. Setty, A. Kilzer, V. Shmatikov, and E. Witchel. Airavat: Security and Privacy for MapReduce. In *Proceedings of the 7th NSDI*, 2010.

[23] N. Santos, H. Raj, S. Saroiu, and A. Wolman. Trusted Language Runtime (TLR): Enabling Trusted Applications on Smartphones. In *Proceedings of the 12th HotMobile*, 2011.

[24] S. Saroiu and A. Wolman. Enabling New Mobile Applications with Location Proofs. In *Proceedings of the 10th HotMobile*, 2009.

[25] S. Saroiu and A. Wolman. I am a Sensor, and I Approve This Message. In *Proceedings of the 11th HotMobile*, 2010.

[26] F. B. Schneider, K. Walsh, and E. G. Sirer. Nexus Authorization Logic (NAL): Design Rationale and Applications. *ACM Transactions on Information and System Security*, 14(1), May 2011.

[27] A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. In *Proceedings of the 21st SOSP*, October 2007.

[28] C. Shepard, A. Rahmati, C. Tossell, L. Zhong, and P. Kortum. LiveLab: measuring wireless networks and smartphone users in the field. *ACM SIGMETRICS Performance Evaluation Review*, 38(3), December 2010.

[29] Skyhook Inc. http://www.skyhookwireless.com, 2011.

[30] Slate. The End of the Credit Card? http://www.slate.com/articles/technology/technology/2011/11/card_cae_the_new_payments_app_that_could_make_cash_and_plastic_.single.html, 2011.

[31] N. O. Tippenhauer, K. Rasmussen, C. Pöpper, and S. Capkun. Attacks on Public WLAN-based Positioning Systems. In *Proceedings of the 7th Mobisys*, June 2009.

[32] Trusted Computing Group. Trusted Platform Module Main Specification, Part 1: Design Principles, Part 2: TPM Structures, Part 3: Commands. Revision 116, March 2011. http://www.trustedcomputinggroup.com/resources/tpm_main_specification.

[33] Washington Times. Guard at Hanging Blamed for Covert Video of Hussein. http://www.washingtonpost.com/wp-dyn/content/article/2007/01/03/AR2007010300358.html, 2007.

[34] A. Wolman, S. Saroiu, and V. Bahl. Using Trusted Sensors to Monitor Patients' Habits. In *Proceedings of the 1st HealthSec*, August 2010.

[35] T. Yan, D. Chu, D. Ganesan, A. Kansal, and J. Liu. Fast App Launching for Mobile Devices Using Predictive User Context. In *Proceedings of the 10th MobiSys*, June 2012.

[36] F. Zhang, J. Chen, H. Chen, and B. Zang. CloudVisor: Retrofitting Protection of Virtual Machines in Multi-tenant Cloud with Nested Virtualization. In *Proceedings of the 23rd SOSP*, 2011.