data collection (because the gateway would almost always be present), and would reduce cost, by obviating the need for a dedicated gateway device.

Unfortunately, it is risky to use mobile phones to collect and process privacy-sensitive and safety-critical medical data. The software running on today's mobile phones is increasingly complex – comparable to desktops and laptops – and vulnerable to malware and other software-based attacks. As phones are used in more sensitive processes (like banking or location-based services), they present an attractive target for attackers. Without reducing this risk, these attacks present a significant obstacle standing in the way of the widespread adoption and deployment of mobile healthcare systems.

We aim to provide strong security and privacy guarantees for mHealth-sensing applications that are based on wearable sensors and off-the-shelf mobile phones. At a high level, there are two aspects to this problem. First, we must ensure the desired confidentiality and integrity properties, within the sensing devices and the mobile phone. Second, we must ensure these properties in the communications between the phone and the sensors. We address the first aspect in this paper; we addressed the second aspect in an earlier paper [29].

Our approach, which we call Plug-n-Trust, is to create a tiny trusted component, implemented on a smart card, that literally "plugs in" to the patient's mobile phone. The system is designed to allow a health-related app to run on the phone, collecting data from a body-area network of sensor devices. Even if the phone has been compromised by malware, the phone will not leak sensitive sensor data, nor be able to tamper with the results reported to the back-end services used by the patient or his caregivers.

There are other possible approaches, which we discuss in more detail in Section 6. Trusted hypervisors [9, 19] try to provide a secure sandbox for sensitive applications, but require the user to replace (or underlay) their phone OS with a hypervisor—not easy for any user and not possible on some phones. Trusted hardware (such as a TPM [41]) provides a secure root of trust on which one may build a trusted software stack that can isolate sensing applications from other less-trusted applications. Such approaches require trust and agreement among many parties, most of whom have nothing to do with mHealth, including the handset manufacturer, the TPM chip maker, the hypervisor developer, and the OS developer. Furthermore, although TPM hardware is common in many desktop and laptop computers, the technology has yet to be broadly leveraged, and has not yet appeared in common mobile phones.

Ultimately, to be practical, the trusted computing base (TCB) on which mHealth rests should require trust between as few parties as possible and these trust relationships should be relevant to the application. For medical applications, it makes sense for the patient to trust the doctor or hospital, and the manufacturers of the sensing devices, but requiring them to trust the handset-maker or its OS vendor is unnecessarily risky. In Plug-n-Trust, as we show, the mobile portion of the trusted computing base is small and robust, and easily deployed: the physician (or other trusted caregiver) simply provides a smart card to the patient, who then plugs it into her phone.

We make three contributions in this paper. First, we describe the design of Plug-n-Trust (PnT), a practical approach to ensuring both the confidentiality and integrity of both the sensing and processing of medical data on untrusted smart phones. Second, we demonstrate the feasibility and limitations of PnT using a prototype implementation on Java-based smart cards and Android mobile phones. Third, we provide a careful security analysis of PnT, demonstrating that it meets the desired properties.

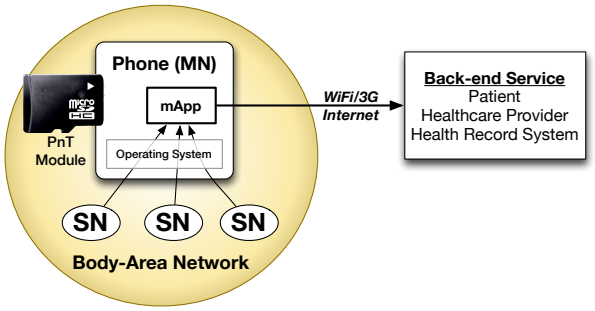

Figure 1: System model

Our discussion of Plug-n-Trust focuses primarily on enabling trusted computing simply, requiring little or no support from smart-phone vendors and OS providers; however, in Section 6.2 we discuss how PnT's secure computing environment can also be adapted to a vendor-deployed solution using built-in secure elements like TrustZone [42].

2. SECURITY MODEL

In this section, we describe the architecture of mHealth monitoring systems that we aim to protect, the adversary model and threat model, the security goals that we aim to achieve, and our trust assumptions. These assumptions form the foundation for the design presented in the next section, and for the security analysis in Section 5.

Our overall goal is to provide tools for vulnerable off-the-shelf mobile devices to host privacy- and safety-critical mHealth applications with minimal risk to patient privacy and security, in the event of an attack.

2.1 System model

Figure 1 provides a high-level view of our mHealth-sensing system. The mobile portion of the system consists of a *mobile node (MN)*, typically a mobile phone, and one or more *sensor nodes (SNs)*. The SNs communicate with the MN via a body-area wireless network, using a protocol like Bluetooth, Zigbee, or our secure protocol [29]. The MN communicates via the Internet to various back-end *services*, such as EHRs, PHRs, or vendor-managed portals that support customers of their mHealth device.

Inside the MN, an mHealth application (*mApp*) is responsible for collecting sensor data from the SNs, processing or aggregating the information in some application-specific manner, and then uploading (some of) the resulting data to the back-end services. Some mHealth sensors (like EKG or EEG sensors) collect data at a high rate, and some mHealth applications dynamically adjust sensing activity based on context. It may not be feasible to upload all of the raw encrypted data for processing on the back-end server. This approach may require too much bandwidth, with accompanying energy and monetary costs; with PnT we enable local operations on the data, reducing bandwidth needs and enabling disconnected operation. Thus, to support application-specific computations on the sensor data when the MN may itself be compromised, we include a plug-in smart card, the "PnT module" in the figure. This card provides a hardware root of

trust, and in our implementation is a commercially-available microSD card that simply snaps into the microSD slot in off-the-shelf mobile phones.

Hardware assumptions

H1. *Crypto.* Each SN has the capability to encrypt and hash its sensor readings, with standard algorithms like AES and SHA [13, 45]. This capability is becoming increasingly feasible in mote-class devices as hardware support for encryption is present in an increasing number of low-power microcontrollers (e.g. TI CC430).

H2. *Clock.* Each SN has an embedded real-time clock that is sufficiently accurate to timestamp its data.

H3. *Platform.* The MN is a general-purpose mobile platform, such as a smart phone, with at least two wireless network interfaces: one for body-area communications with the SNs (comparable to Bluetooth or Zigbee), and one for Internet communications (e.g., Wi-Fi or 3G).

H4. *Internal sensors.* The MN may have internal sensors, such as a clock, camera, or GPS, but we must assume that these are compromised when the MN is compromised and thus PnT is unable to use them.

2.2 Adversary and threat model

A wide range of attacks have been demonstrated against computing devices and networks, ranging from passive network sniffing to hardware-based attacks that require physical possession of the device. We consider secure communication between the mobile phone and the back-end service, as well as securing the back-end service itself, to be relatively routine problems that have been addressed in a variety of ways. In this paper, therefore, we focus on the security of the MN. (We are less concerned with the SNs, which are single-purpose embedded devices with a small attack surface, and are not connected to the Internet. Although some attacks have been demonstrated [23], mobile phones are much easier to attack.)

Mobile phones are at risk from many kinds of compromise. Software-based attacks are particularly dangerous, since software vulnerabilities are universally common, and attacks can be launched remotely on many devices at once, at a small cost to the attacker. Continuous efforts are being made to protect mobile platforms against intrusion (e.g., MTM [12] or virtualization [9]); however, attacks will continue to evolve, and some attacks will inevitably succeed. All system components from the applications down to the kernel, hypervisor, and device drivers may potentially be compromised. These attackers may have the ability to read system memory, access communication channels, and arbitrarily modify system behavior – including the processing of medical sensor data.

The patient should be concerned about several threats.

T1. The adversary may recover medical sensor data (or data derived from it).

T2. The adversary may modify sensor data (or data derived from sensor data), or inject false data, without it being detected.

T3. The adversary may prevent the system from collecting or reporting sensor data (DoS) without it being apparent to the patient and back-end server.[1]

[1]Although DoS attacks are a real threat, we do not address DoS attacks in this paper. See Section 7.1 for discussion.

We assume a strong adversary with some limitations, relevant to the threats above:

A1. *Computationally bounded.* The adversary cannot break cryptographic primitives such as AES and SHA.

A2. *No compromise of SN.* The adversary does not have the capability to compromise either the software or the hardware of the SN, at least, without it being immediately evident to the patient.

A3. *Physical attacks.* The adversary is not capable of physical side-channel attacks such as power, heat, sound, and electromagnetic clues to learn about the information stored within the card.

A4. *Local adversary.* The adversary cannot compromise any of the back-end services, but can compromise the software running locally on the MN.

A5. *Secure body-area network.* The adversary is unable to compromise patient confidentiality or anonymity, or data integrity, by observing wireless body-area network traffic; solutions exist [29].

2.3 Security properties

In the face of a powerful adversary, and to avoid the threats listed above, we focus on the providing the following security and privacy properties in our design of a more secure approach to mHealth sensing.

SP1. *Data Confidentiality.* Sensor data and any data derived from sensor data remains confidential to all entities other than the system's trusted components (sensor node, the smart card, and the back-end server).

SP2. *Sensor Data Integrity.* The sensor data remains intact during its processing and delivery to the back-end server, and any injection of incorrect data or past data into the system is identified.

SP3. *Derived Data Integrity.* Derived data is obtained from sensor data by a series of computations defined by the doctor. Derived data remains intact in transit, and any attempt to deviate from the predefined derivation logic is identified.

2.4 Trust model

Any trustworthy system is built on certain assumptions about who trusts whom, and in what way. In our efforts to achieve the above goals, we make the following trust assumptions about the patient who uses the sensors, the healthcare provider (doctor, hospital, or other entity that consumes the data), and the companies that manufacture the sensors and smart cards.

TR1. The patient and the healthcare provider trust the SN manufacturer to produce calibrated sensors that operate correctly, so that the patient and the health provider can trust the sensor to provide the right reading.

TR2. The patient and the healthcare provider trust the card manufacturer to correctly implement PnT within the card; thus, it protects confidentiality and integrity according to the above goals, and in face of the above adversaries.

TR3. The patient trusts the healthcare provider (or other data consumer) to keep sensor data confidential after it has been delivered, and not to reveal encryption keys.

TR4. The healthcare provider trusts the patient to not be malicious and to not tamper with the hardware or software of the sensor node, or the PnT card.

The manufacturer has no stake in the system, so the manufacturer assumes nothing about other principals.

3. Plug-n-Trust

Plug-n-Trust (or PnT) is a novel approach that enables confidential and trustworthy processing of safety-critical medical sensor data. As its name implies, PnT uses a smart card that *plugs* into a phone's microSD slot and serves as a *tiny trusted third party* [24] to create a safe and trusted computing environment for critical data processing on an untrusted mobile phone. In short, sensitive applications rely on the card to perform sensitive processing, and the card provides verifiable proof that computations were conducted properly.

In addition to the security goals of ensuring confidentiality and integrity, PnT is designed to be usable, deployable, flexible, and minimal, as the following paragraphs explain:

Usable: Requiring users to remember secrets (e.g., passwords), configure permissions, and annotate content often makes systems more difficult to use, and may lead users to bypass or disable security features altogether [11]. Instead, PnT only requires a user to plug the trusted smart card into her phone. After that the system operates automatically in the background, without any additional interaction.

Easily Deployable: In addition to being difficult to use, many technically feasible security systems require too much infrastructure or agreement between too many different parties (e.g., hardware manufacturers, software providers, and network operators) to be deployed. For example, trusted software stacks whose validity is verified by a hardware Trusted Platform Module (TPM) [41] require a complex trust relationship between the TPM manufacturer, the platform manufacturer, the operating-system vendor, and the application developer, and in some cases, the cellular-network provider (who often controls aspects of the platform or its operating system). Furthermore, TPM programming remains highly challenging, so TPMs have only been used in limited settings [37]. New approaches, such as Logical Attestation [39], may ease the use of TPMs if smart-phone vendors were to adapt the Nexus operating system to small-scale platforms.

In contrast, PnT can be easily deployed because it only requires trust relationships between the parties that are directly involved (e.g., patient, provider, and card manufacturer), independent of those who are not (e.g., cell-phone service providers, phone manufacturers, operating system developers, or app stores).

Flexible: Applications change. Providers may want to adjust sensing and processing just like they adjust doses of medication. PnT is flexible by design. Applications can be updated, modified, and tuned. Application designers need only describe to the back-end server how data is processed, so that its integrity can be verified. (Specifically, they provide a compact representation of the sequence of mathematical operations on the sensor data, but need not provide the source code for the whole sensing application.)

Minimal: Minimizing a system's Trusted Computing Base (TCB) – the set of code that must remain intact to ensure security – is essential to building secure systems [38]. In existing systems the TCB is either the OS kernel, a hypervisor [19], or a secure coprocessor [24]. In PnT the TCB (on the phone) runs only on the plug-in smart card. This hardware separation from all other code running on the system, with a simple API, provides an extremely small attack surface.

Functional: The PnT card's storage space can also be used to store other media.

3.1 Plug-in smart card

For decades, smart cards have been used to protect encryption keys and other sensitive information in applications ranging from financial payment systems to cell phones (SIMs). Early cards were used primarily for identification and had limited storage; however, recent increases in both storage and computation resources (including hardware support for cryptography and secure random-number generation) make these tiny tamper-resistant devices capable candidates for more sophisticated tasks.

In Plug-n-Trust, we use a smart card in a microSD form factor [17] as a pluggable trusted third party or secure coprocessor that represents the patient's interests in a potentially unsafe mobile phone. Phones that do not have a microSD card slot could run PnT on a SIM card instead [33]. While not as powerful as traditional secure coprocessors (like the IBM 4758, used in the FaeriePlay prototype [24]), smart cards are much more portable and more affordable.

Smart cards also provide PnT users with portability. They are supported by many major mobile-phone operating systems (Android, Windows Mobile, BlackBerry, PocketLinux, and Symbian), and even more importantly they allow a patient's security to be independent of a single mobile device or service provider. When a patient buys a new phone, the user simply transfers the card to the new phone, leaving no sensitive information behind and requiring no configuration, other than installing an app—which may be loaded from the smart card—on the new phone. If the card is lost or stolen, its tamper-resistant design makes it more difficult for the attacker to access private information (see Section 7.1 for more detail). Note that we do not assume tamper-proof cards, for which tampering is impossible; tamper resistance, like all security mechanisms raises the bar for the attacker.

The scant resources (a few kB for data and code) prevent us from loading entire applications onto the card. Instead, our goal is to do as little on the smart card as possible. The application's processing of sensitive data must happen in the card, but program logic for sensor discovery, data collection, and data delivery are performed on the mobile phone. While this design choice is born from necessity, the benefits of a small code base and an extremely simple interface between the application and smart card (smaller attack surface) make PnT easier to secure.

3.2 Bootstrapping Plug-n-Trust

PnT uses encryption to protect sensitive data when it is transmitted between a sensor and the smart card and again during delivery to the back-end server. This encryption requires secret keys that are used to ensure both confidentiality of the data and authenticity of the system components. Specifically, PnT requires each sensor s to have a key, K_s,

which is known to the smart card. The smart card has a key, K_c, which is also known to the back-end system.

A critical challenge with any cryptographic system is, of course, key distribution. We need a secure way to share secret keys between the PnT card and the sensor nodes, without trusting the MN and its operating system. One major advantage of PnT is that the card can be removed from the phone, and paired directly with the SNs. We imagine a range of possible approaches, depending on how the sensors and smart card are deployed. Both sensors and smart card may be provided by the care provider (who also administers the back-end server), allowing keys to be pre-installed before the patient receives the devices. More likely, the patient may receive the sensors from a trusted proxy of the provider, such as a pharmacy. In that case, the pharmacist would insert the patient's PnT card into a slot on the pharmacist's own trusted device, which then may use any number of existing techniques to develop a shared secret between the card and the SN. Alternately, the card might come with a pairing device, much like that in the pharmacy example, which the patient can keep at home and use for introducing new sensors to the card. (Later, if the SN is lost or discarded, the card needs to revoke the relevant keys; the card can be informed via one of these trusted devices, or in a secure message sent from the provider via the MN.) Finally, Go-Trust plans to ship smart cards that support Near-Field Communication (NFC), allowing the card to communicate directly with NFC-capable SNs [21].

Notably, PnT is not tied to a single crypto-system, allowing implementers considerable flexibility. An implementation could easily use either symmetric (AES, DES) or asymmetric (RSA, ECC) encryption. This decision has well-known trade-offs. Asymmetric encryption provides greater flexibility for key distribution and bootstrapping, but requires more computation and larger keys. Our implementation uses AES-256 (in CBC mode), a symmetric algorithm, which is efficient for low-power sensor devices.

3.3 Plug-n-Trust basic operation

In PnT, data is encrypted before it leaves the sensor, remains encrypted while stored on the mobile phone, and only is decrypted (and modifiable) in the card, as shown in Figure 2. The card exposes a simple API that facilitates communication between the application and the card. We assume that a sensing application is a series of *tasks*, and the mApp signals the beginning of a new task with the command:

$$\texttt{start}()$$

which resets internal data structures on the card.

Moving data in and out of the card

Raw data messages arrive from sensor s encrypted with the sensor's key, K_s. In addition to one or more raw data readings (*data*), the message also includes additional metadata—the ID of the originating sensor (s), the time of collection (t), a sequence number (n), an error flag (e), and a message authentication code (MAC) computed over all of the above (s,t,n,e, and *data*) after encryption. The HMAC algorithm uses a key, K_s^m, derived from K_s—to protect the integrity of the data and its processing.

$$m = \{s, t, n, e, data\}_{K_s}, HMAC\left(\{s, t, n, e, data\}_{K_s^m}\right)$$

The application transfers the encrypted data message, m,

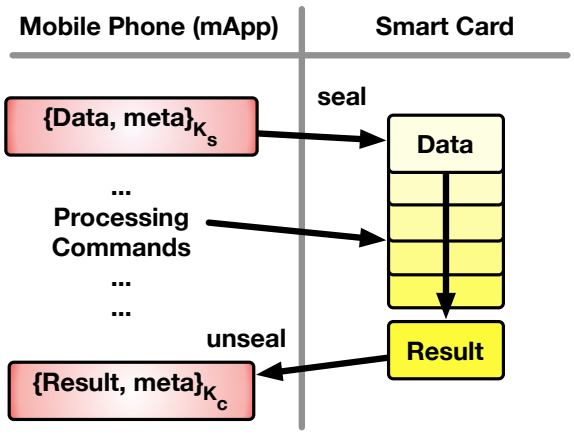

Figure 2: In Plug-n-Trust, encrypted sensor data is passed (sealed) into the smart card, processed according to application-supplied commands, and the encrypted result is then removed (unsealed) from the card.

as well as the ID of the originating sensor, s, into the card using the `seal` command. Passing in the sensor ID, s, is not strictly necessary, since the smart card could try to decrypt the message with all of its known keys; however, as the number of keys increases this quickly becomes inefficient. After the data and metadata are stored in the card, `seal` returns a data-independent reference, r_m, that the application can use in subsequent processing operations on the data.

$$\texttt{seal}(s, m) \Rightarrow r_m$$

Sensor readings can be sealed into the card individually, or as vectors of sequential readings, which is useful for reducing the amount of communication between the application and the card. For simplicity, our design assumes that sensor readings are integer values; floating-point support would be an easy extension to the architecture.

Once processing is complete, the application can use `unseal` to retrieve the computational result and associated metadata used by the server to validate the computation (see Section 3.5). The card encrypts all of this data before releasing it to the phone.

$$\texttt{unseal}(r_m) \Rightarrow \{result, metadata\}_{K_c}$$

Arithmetic operations

$\texttt{add}(r_1, r_2) \Rightarrow r_3$	$\texttt{addc}(r_1, c) \Rightarrow r_2$
$\texttt{sub}(r_1, r_2) \Rightarrow r_3$	$\texttt{subc}(r_1, c) \Rightarrow r_2$
$\texttt{mult}(r_1, r_2) \Rightarrow r_3$	$\texttt{multc}(r_1, c) \Rightarrow r_2$
$\texttt{div}(r_1, r_2) \Rightarrow r_3$	$\texttt{divc}(r_1, c) \Rightarrow r_2$

After data has been sealed into the card, PnT provides a variety of commands for processing the data. Standard arithmetic operations are provided that add, subtract, multiply, and divide the values associated with two references, r_1 and r_2, and return a reference, r_3, to the result.

The references can refer to scalars or vectors, with straightforward semantics. If both references refer to scalar values, then the operation is applied to both and the result is also a scalar. If both are vectors, then the requested operation is applied in pairwise fashion to corresponding elements in each vector, resulting in a vector of the same length. If the

vectors are of different lengths—which is likely an uncommon case—then some entries will be ignored and the length of the result is $min(length(r_1), length(r_2))$. Finally, if one value is a scalar and the other a vector, then the scalar value is applied to each of the vector's entries.

Corresponding commands are also provided for arithmetic with a fixed constant. For example, $\texttt{divc}(r, 2)$ divides an in-card scalar or vector referred to by r by 2.

Summarizing data

$$\texttt{sum}(r_1) \Rightarrow r_2 \qquad \texttt{max}(r_1) \Rightarrow r_2$$
$$\texttt{prod}(r_1) \Rightarrow r_2 \qquad \texttt{min}(r_1) \Rightarrow r_2$$
$$\texttt{len}(r_1) \Rightarrow r_2$$

A set of commands are also provided that reduce a vector of elements to a scalar. These commands allow an application to compute the sum or product of a vector, or determine the maximum or minimum element in the vector. These commands only make sense for vectors, and if applied to a scalar the result will just be a copy of the original argument, except `len`(), which outputs 1.

Logical comparisons

$$\texttt{gt}(r_1, r_2) \Rightarrow r_3 \qquad \texttt{gtc}(r_1, c) \Rightarrow r_2$$
$$\texttt{lt}(r_1, r_2) \Rightarrow r_3 \qquad \texttt{ltc}(r_1, c) \Rightarrow r_2$$
$$\texttt{eq}(r_1, r_2) \Rightarrow r_3 \qquad \texttt{eqc}(r_1, c) \Rightarrow r_2$$

In addition to arithmetic computations, the ability to compare sensor readings to each other, or to a constant value, is critical to many applications that process sensor readings.

$$\texttt{and}(r_1, r_2) \Rightarrow r_3 \qquad \texttt{or}(r_1, r_2) \Rightarrow r_3 \qquad \texttt{not}(r_1) \Rightarrow r_2$$

The Boolean results of those comparisons can also be modified using the standard `and`, `or`, and `not` Boolean operators. Both comparison and Boolean commands can be applied to both scalars and vectors. When vectors are used, the desired operation is applied to each element of the vector, and the result is a vector.

Conditional execution

$$\texttt{if}(r_1, r_2, r_3) \Rightarrow r_4$$

Finally, without the ability to support data-dependent processing, PnT would not be useful to a significant number of applications. The `if` command makes conditional execution possible as demonstrated in the following example, which computes the arithmetic mean of the elements of vector, d, and uses conditional execution to avoid dividing by zero.

$$\texttt{seal}(s, \{d\}_{K_s}) \Rightarrow r_1 \quad (1)$$
$$\texttt{sum}(r_1) \Rightarrow r_2 \quad (2)$$
$$\texttt{len}(r_1) \Rightarrow r_3 \quad (3)$$
$$\texttt{eqc}(r_3, 0) \Rightarrow r_4 \quad (4)$$
$$\texttt{div}(r_2, r_3) \Rightarrow r_5 \quad (5)$$
$$\texttt{if}(r_4, r_3, r_5) \Rightarrow r_6 \qquad (r_6 \Leftarrow r_3 \textit{ or } r_5) \quad (6)$$
$$\texttt{unseal}(r_6) \Rightarrow \{\text{result}, \text{metadata}\}_{K_c} \quad (7)$$

The `if` command on line 6 accepts three arguments: a Boolean value stored at reference, r_4, which resulted from the comparison on line 4; and two possible results. The second possible result is r_3, which equals zero in the error condition. The second, r_5, is the result of the division on line 5. If a divide by zero occurred, then the division fails silently, the value in r_5 is undefined, and its error flag is set. According to the typical C-style ternary operator semantics, the value pointed to by r_6 will be the value of r_3 if r_4 is true, and the value of r_5 otherwise. Note that conditional execution in PnT requires both sides of a branch to be executed, which is wasteful in many cases, but necessary to avoid leaking information about the values of the data.

The other major control-flow construct, loops, can be implemented within the mApp. That is, the mApp can loop as many times as needed, sending commands to the card. Data-dependent loop conditions can be implemented (albeit inefficiently) using the `if` command (see Section 7.1).

Additional commands

When selecting commands for PnT, we focused on supporting common processing tasks used in mHealth applications, like summarizing data (statistics such as mean, median, variance, and covariance) and checking data ranges and thresholds (is the patient's blood sugar within a safe range?).

As shown in our evaluation section, our current set of operations also supports Actigraphy [4], which is a well-known metric for measuring activity level based on accelerometer data, often used for diagnosing sleep disorders. There are three ways to compute Actigraphy. The most popular and sophisticated method is Proportional Integral Mode (PIM), which computes the area below the curve; we integrate using the trapezoidal integration rule. Other methods include Zero Crossing Mode (ZCM), which computes the number of intersections between zero axis and the curve, and Time Above Threshold (TAT), which measures the length of time that the signal is above a certain threshold. Our proposed operation set easily supports both ZCM and TAT as well.

We discuss the potential for other, more complex computations in Section 7.2.

3.4 Handling errors and hiding variation

Errors, exceptions, and conditional execution may all potentially leak information about the data being processed. Returning a divide-by-zero error, for example, reveals that the divisor is equal to zero. Likewise, a naive implementation of `if` might allow an attacker to learn which block was executed, if one took longer than the other. Since we assume that the attacker issues arbitrary commands, errors and conditionals could provide the tools an attacker would need to discover the value of any data element stored in the card. In our design of Plug-n-Trust, we deal with these cases in three ways:

Fail silently: When data-dependent errors occur, like divide-by-zero, PnT fails silently. A result is returned as though the operation succeeded, and internally, the data's associated error bit is set. The error will be apparent to the back-end server when the data is received, but not to the attacker.

Take the long way: PnT avoids information leakage by removing timing variation from its execution. For example, in `if` operations, *both* possible results must be computed. This maintains the confidentiality of the data, at the cost of some wasted computation. In practice, we have found this wasted computation to be minimal, though pathological cases exist that would be impractical.

Stick to one size: When unsealing, PnT always returns a vector padded out to its maximum size, which in our implementation must be a multiple of the AES block size

(16 bytes); currently our vectors are all 32 bytes long. The adversary cannot learn anything about data values from the size of the returned data.

3.5 Detecting invalid processing

The architecture, as described thus far, is sufficient to keep sensor data and processing confidential. Of course, ensuring confidentiality alone is not sufficient. The attacker, who can issue arbitrary commands, may attempt to modify the data-processing results to cause harm without knowing any of the sensor values. To protect the patient against invalid processing, the back-end system (acting on behalf of the care provider) should be able to verify whether the expected data processing has occurred.

A naive approach might keep a history of all operations that are performed on each piece of sensor data, and ship that history with the data when it is unsealed. Whoever uses the data can then verify that the history is valid. This would be simple to implement, but completely impractical. A key reason for processing data locally is to reduce bandwidth requirements—a benefit that would be defeated by storing complete histories.

Instead, for each data element, d, PnT represents its processing history (or path) using a compact incremental hash, H_d. These hashes are similar in flavor to "path identifiers" used by path-profiling algorithms (e.g., Ball-Larus [7]); however, path-profiling algorithms only prevent path ID collisions within a program while PnT must protect against external hash collisions. The back-end server knows enough about the mApp to know the proper sequence of operations, the set of sensor IDs, and the expected sequence of operands. It can thus compute an expected path hash, and compare it with the path hash reported from the card, to verify whether the correct data was correctly processed.

Figure 3 shows an example of path hashes; in this computation of $(y_1x_1 + 5x_2)$ two readings x_1 and x_2 are collected from s_x (in that order), and y_1 is collected from s_y. If the adversary changes the operations, or rearranges the operands, the path hash will be different and allow the back-end to detect the attack. PnT computes a hash value at each step, incorporating information about both the op-code and the operands; the hash function H can be any cryptographically secure hash function (our implementation uses SHA-256). For arithmetic commands, such as `mult` or `add`, the path hash is computed by applying H to the concatenation of the op-code Op_{cmd} and a representation of the arguments: for a constant c, we hash in c; for a reference r, we hash in H_r.

At the beginning, the `start`() operator notifies the card of a new task. When operator `seal`$(s, \{s, t, n, e, data\}_{K_s})$ is called, where n is the sequence number, and t is the sensing time, the card does the following: If it is the first seal command for sensor s (since the `start`), the card stores the sequence number n in an internal variable $seq(s)$. For all `seal` operations, the card computes the hash value for the operation as $H(Op_{seal} \mid s \mid n - seq(s))$ where Op_{seal} is the op-code for sealing. We call $(n - seq(s))$ the *relative sequence number*. For example, the first sealings for sensors s_x and s_y (the second and third lines in Figure 3) sets each relative sequence number to 0, while the subsequent sealing (the fifth line) results in 1. With the relative sequence numbers embedded in the path hash, the back-end server can check the correct order of the data (see Section 5.2): if, for example, the adversary swaps x_1 and x_2, H_{r_1} and H_{r_4} would change and thus so would the overall H_{r_6}.

Figure 4: Our prototype, configured for power measurement.

Upon receiving unsealed data, the back-end server precomputes the hashes it expects to receive (based on the desired data processing), and compares those hashes to those included in the data. Hashes that do not match in the back-end's computation are invalid, indicating an attack.

For the back-end server to verify freshness of the computation, it must be given some knowledge of the time during which the data was collected. Thus, we maintain for each reference a time range (t_{min}, t_{max}) that represents the timestamps of the data used to compute its value, and each operator's output is assigned a time range that is the union of the time ranges of its inputs. Figure 3 shows the final time range reported as part of the `unseal` result.

3.6 Managing card resources

Smart cards have limited resources; for example, the G&D card we use has 160 Kbytes of ROM, 4608 bytes of RAM and 72 Kbytes of EEPROM. The EEPROM is shared between program and data and when EEPROM is exhausted the smart card will start returning errors. A simple resource manager could keep track of data storage to handle errors of this nature more gracefully. Further, there is no automatic garbage collection so the programmer must remember to free references when done with them. When data is unsealed all intermediate values could instead be automatically freed. See Section 7 for a discussion of automatic path extraction and automatic reference freeing.

4. IMPLEMENTATION AND EVALUATION

To evaluate the potential for our approach in real applications, we implemented PnT using current mobile-phone and smart-card technology. Our test hardware, shown in Figure 4, includes a G&D smart card [17] and two smart phones – the G1 (shown) and Nexus One Android phones.

PnT's on-card component is implemented as an applet for the JavaCard v2.2.1 platform. The G1 and Nexus One phones

Operation:	Path Hash:	Time range:	Sequence numbers:
`start`()			
`seal`$(s_x, \{x_1, n_{x_1}, t_1\}_{K_{s_x}}) \Rightarrow r_1$	$H_{r_1} = H(Op_{seal} \mid s_x \mid 0)$	$time(r_1) = (t_1, t_1)$	$seq(s_x) = n_{x_1}$
`seal`$(s_y, \{y_1, n_{y_1}, t_2\}_{K_{s_y}}) \Rightarrow r_2$	$H_{r_2} = H(Op_{seal} \mid s_y \mid 0)$	$time(r_2) = (t_2, t_2)$	$seq(s_y) = n_{y_1}$
`mult`$(r_1, r_2) \Rightarrow r_3$	$H_{r_3} = H(Op_{mult} \mid H_{r_1} \mid H_{r_2})$	$time(r_3) = (t_1, t_2)$	
`seal`$(s_x, \{x_2, n_{x_2}, t_3\}_{K_{s_x}}) \Rightarrow r_4$	$H_{r_4} = H(Op_{seal} \mid s_x \mid 1)$	$time(r_4) = (t_3, t_3)$	$n_{x_2} - seq(s_x) = 1$
`multc`$(r_4, 5) \Rightarrow r_5$	$H_{r_5} = H(Op_{multc} \mid H_{r_4} \mid 5)$	$time(r_5) = (t_3, t_3)$	
`add`$(r_3, r_5) \Rightarrow r_6$	$H_{r_6} = H(Op_{add} \mid H_{r_3} \mid H_{r_5})$	$time(r_6) = (t_1, t_3)$	
`unseal`$(r_6) \Rightarrow \{time(r_6), value(r_6), H_{r_6}\}_{K_c}$			

Figure 3: An example procedure for computing path hash and generating the message to be sent to the back-end server. The task is to compute $y_1x_1 + 5x_2$. Assume that $t_1 < t_2 < t_3$ and $n_{x_1} + 1 = n_{x_2}$. For clarity we do not include the error codes (e) in the inputs and outputs or the computed HMACs for `seal` and `unseal`.

run Android versions 1.6 (Donut) and 2.3.5_r1 (Gingerbread) respectively, both modified with seek-for-android [36], a free implementation of PC/SC ("Personal Computer/Smart Card", a specification for smart-card integration into computing environments) for Android phones.

As examples, we implemented several computations that are commonly used in processing health-related sensor data: *Mean & variance*, which compute basic statistics (e.g., arithmetic mean, variance, data range) that help to remove noise and quantify volatility; *Zone detection*, which help to monitor heart-rate or blood-glucose levels, because providers often want to know if values fall outside a particular zone rather than the specific data values; and *Sleep actigraphy*, in which accelerometer readings are integrated over time to measure a patient's sleep patterns.

In this section, we evaluate the overhead and performance limitations of PnT, using our current implementation. Specifically, we focus on energy consumption and processing speed, using today's smart-card technology, and identify key focus areas for future improvement.

4.1 Energy overhead

To assess the impact of PnT on mobile device battery performance, we measured the quiescent power consumption as well as the per-command and per-application energy cost of our PnT prototype. We measured energy consumption using a Monsoon Power Monitor[2] and a microSD card extender, with custom modifications to facilitate power monitoring, as shown in Figure 4. For the sake of comparison, we use this same setup to measure the idle power and active energy characteristics of a standard Transcend microSD flash card plugged into the same phones.

The measured idle power draw of the G&D smart card is roughly 9mW—an order of magnitude higher than that of a standard flash card; however, this would still require 4 days to drain just 10% of the G1 phone's battery. For low-frequency applications where the smart card is used infrequently, the card can also be powered down between uses.

The per-task energy consumptions are shown in Table 1 for several PnT tasks, our applications, and 1 MB reads and writes to the flash card. While we only show results for a subset of PnT's processing commands, the time and energy requirements of the other data-processing operations are comparable. The impact of PnT on device lifetime will depend on how often it is used as well as other unrelated activities on the phone; however, even if PnT were constantly processing data, it would drain only 20% of the phone's battery over the course of a day.

[2]http://www.msoon.com

4.2 Processing performance

While the energy costs for processing are not a significant limitation, processing speed is still a challenge. Table 1 also shows the computation time of the PnT commands and applications, the maximum data rate (samples/s) that each can sustain, and the percentage of time that is spent by each in communication between the phone and the card. For tasks that must compute an HMAC, the percentage of time spent computing the HMAC is also shown. The max data rate is ten times one over the computation time, since ten samples were processed per seal. The percentage of time is the measured communication ping time divided by the computation time (for the apps the ping time was multiplied by the number of operations). These performance results are shown for the Nexus One phone. The results for the G1 phone running Donut were identical, except it had an additional communication latency of roughly 30 ms for each message sent to the card. These results show that, at least for this generation of smart cards, PnT is suitable primarily for applications with low-to-medium sampling rate. This poor performance stems from three key factors: slow communication, lack of HMAC support, and the card's built-in persistence mechanisms.

To date, smart cards have been used primarily as secure key stores that perform cryptographic functions for e-commerce applications with low data-rate requirements. Consequently, they have been optimized for fast cryptographic operations, but not for fast communication—a fact that results in PnT spending 16–93% of a given task's time in communication. Moving PnT to a newer phone (Nexus One) and a newer version of Android (Gingerbread) improved communication performance by nearly 50%, and we expect that overhead can be dramatically reduced further by optimizing the software that communicates with the card.

A second performance challenge is that our current cards do not provide support for HMAC computations. Our software HMAC implementation currently accounts for more than 50% of the `seal` and `unseal` operations and 12–36% of the total runtime in our test applications. Based on the next-generation JavaCard specifications, we expect newer cards to provide hardware support for HMAC in the near future.

The third reason for slow performance in PnT stems from the fact that today's smart-card hardware and software systems are designed for both powered and contactless appli-

Operation	Time (ms)	Max rate (samples/s)	Energy (mJ)	Overhead (% time) Comm	HMAC
`seal`	226.2	44.2	13.1	16.5%	50.3%
`unseal`	182.6	54.8	10.6	20.5%	58.0%
`free`	39.9	250.9	2.3	93.7%	
`add`	142.9	70.0	8.3	26.1%	
`multc`	111.3	89.8	6.4	33.5%	
`div`	117.9	84.8	6.8	31.7%	
`prod`	101.9	98.1	5.9	36.6%	
`if`	112.6	88.8	6.5	33.2%	
Sleep Actigraphy	600.9	16.6	34.7	31.1%	36.6%
Mean & Variance	1585.9	6.3	91.7	44.7%	12.1%
Zone Detection	2101.3	4.8	121.5	31.7%	30.8%
Flash card (read,1MB)	49.8		6.325		
Flash card (write,1MB)	184.6		22.702		

Table 1: Timing/energy results for individual PnT operations on a Nexus One smart phone using 10-sample vectors, and for comparison (in the bottom two rows) for read and write operations on a generic flash-memory card. The maximum sustainable sensing rate (sample/s) is also shown. For PnT, the key limiting factors are the slow communication speeds of current smart cards and slow HMAC performance, which combined result in 59-69% of the total latency for our test applications. As smart cards find increasing use in higher-rate applications (like sensor data processing), we expect hardware performance to improve dramatically.

cations. To survive transient power failures, the JavaCard environments store objects and variables in EEPROM by default, which has much longer access times than RAM, especially for write operations (note that `seal` takes nearly twice as long as `unseal`). JavaCard v2.2.1 allows simple arrays to be declared as *transient* and stored in RAM; however, many EEPROM accesses cannot be avoided (access to objects and their non-array attributes). The potential for improvement can be seen in the `free` operation—PnT's only EEPROM-free operation—which has more than 93% communication overhead. The next generation of smart cards will allow greater use of transient variables.

Finally, some smart-card companies have recently started selling high-performance microSD products that can encrypt voice and video traffic in real-time. These cards are not currently programmable by third-party developers. While communication, HMAC support, and mandatory EEPROM accesses represent significant limitations to PnT's current processing performance, these limitations are neither fundamental nor likely to remain for long. When resolved, we expect PnT to be able to support many more applications with much higher data rates.

5. SECURITY ANALYSIS

To assess the security of PnT, we describe how PnT guarantees the security properties set forth in Section 2.3, preventing adversaries from succeeding in their attacks, some of which were highlighted in Section 2.2. In particular, we discuss how PnT achieves data confidentiality and integrity.

5.1 Data confidentiality

PnT protects the patient's privacy by enforcing our first security property **SP1**:*Data Confidentiality*. The adversary may attempt to break this property either directly or indirectly. In a direct attack, the adversary tries to directly access the sensor data or any derived information, i.e., the final or intermediate results of the data processing. PnT prevents this attack by encrypting all sensor data whenever it leaves trusted components: sensor nodes, the plug-in card, and the back-end server. These components are assumed to protect data privacy and integrity (assumptions A2 and TR1 for the SN; A3 and TR2 for the card; A4 and TR3 for the server). The adversary cannot succeed in this attack unless she can break the chosen cryptosystem (e.g., AES). The intermediate results are never exposed outside the card; only data-independent references to the results are revealed.

In an indirect attack, the adversary tries to learn about the sensor data and derived information by observing the size or timing of responses by the plug-in card. The card returns only two types of values: references (which are small, opaque fixed-size objects), and unsealed results (which are encrypted, fixed-sized vectors). Thus, the size of intermediate and final results provide no information; what can the adversary learn from the computation time? Obviously, more-complex computations take longer. However, this information is useless because the adversary already knows what computation will be performed. For the basic operators (arithmetic, Boolean, and conditional operators) each operation takes a fixed amount of time; the adversary learns nothing about the data values. The `if` statement may seem promising; if the adversary can learn which branch (then/else) was chosen, it can infer properties of the data used in the conditional expression. PnT prevents this inference by performing both sides of the branches, and only retaining the result of the correct branch.

Since most smart cards are designed to resist physical attacks, we assume that physical side-channel attacks are outside the capability of the adversary (A5).

5.2 Integrity

The provision of security property **S2** (*Sensor Data Integrity*) is straightforward because the encryption, path hash, MAC, and timestamp protect data against modification,

forgery, or replay (by assumptions H1, H2, A1, and A5). The adversary cannot break the data integrity unless she can break the chosen cryptographic primitives (A1) or the clock within the sensor (H2, A2).

Security property **S3** (*Derived Data Integrity*) is more challenging due to its larger attack surface. We next discuss possible attack methods and justify how PnT counteracts.

Wrong sensor type: When processing sensor data (body temperature, for example) the adversarial mApp may attempt to feed the processing routine with the wrong type of sensor data (say, systolic blood pressure). It does so by giving the wrong arguments to the `seal` operator: supplying the wrong sensor ID or sensor data. If the wrong sensor ID is given to the `seal` command, the card will fail to decrypt the data and will return an error. If the correct sensor ID is provided, but the sensor is the wrong type for the operations, the computation will proceed. Because the sensor ID is included in the computation of the path hash, however, the back-end server can immediately detect the use of an incorrect sensor.

Stale sensor data: The adversary may provide a set of sensor readings received in the past, rather than fresh readings. The back-end server can detect this replayed or delayed data immediately because the sensor timestamp is incorporated into the (t_{min}, t_{max}) range reported in the package exported by the card, as described in Section 3.

Wrongly ordered data: The adversary may provide fresh data, from the right sensor, but reorder, repeat, or omit some of the readings. When sensor readings are re-ordered, or repeated, or omitted, the computation may result in different values. PnT prevents this attack by adding the relative sequence number of each datum into the path hash. So, whenever a re-ordering or omission attack occurs, the path hash will not match what is expected.

Wrong constants: The adversary can easily change the processing algorithm by replacing one constant with another, for example, to call `div(r1,4)` instead of `div(r1,2)`. Since the constant value is included in the path-hash computation, this attack causes a mismatching hash value, detected at the back-end server.

Wrong procedure: The adversary may manipulate the procedure, change the order of operations, insert new operations, remove operations, and so on. These changes affect the hash value and the back-end server can detect them.

Hash collision attack: The most sophisticated attack is by an attacker who knows how all the above attacks can fail. Here, the attacker strives to find a computation path that has the identical hash value as the correct path, and that produces a different result, hoping the result will be harmful to the patient. She cannot succeed in this attack without, in effect, breaking the underlying hash function (A1). The point of cryptographic hashes, like SHA-256, is to make it difficult to find such collisions.

6. RELATED WORK

Plug-n-Trust builds on and is inspired by a large body of related work, providing support for trustworthy sensing, secure multiparty computation, and virtualization.

6.1 Trustworthy mobile sensing

Several systems have worked towards providing mobile sensing systems that can be trusted to provide users with reliable data. A common approach is to have wireless sensors, with or without hardware support for cryptography, sign their own sensor readings [13, 34, 45]. Encryption can be used with this sign-and-send model to protect privacy, and data can be trusted, since it cannot be modified until it is delivered. Unfortunately, this can result in a large volume of data that can strain bandwidth limits and increase patient cost.

Another proposed approach [19] allows local processing of sensor data within a trusted OS and hypervisor backed by a TPM [41], without trusting the application. While this allows greater flexibility than the simpler sign-and-send systems, it also presents a much larger attack surface, and other challenges faced by TPM-based and hypervisor-based systems, which we describe in the following sections.

6.2 Trusted platforms

The Trusted Computing Group [41] has proposed using an on-board hardware root of trust to establish a trusted software stack (or Trusted Platform) using binary attestation. Others [2, 12, 15, 19] have extended this idea to mobile devices (e.g., Mobile Trusted Module (MTM) [31]).

To date, in spite of the widespread availability of TPMs in desktop and laptop computers, trusted platforms have failed to find widespread use. MTMs have not been embraced by industry, who have chosen instead to use legacy security solutions. Debates about Digital Rights Management (DRM), a primary driver for both TPMs and MTMs, have highlighted another challenge, determining what entities should have control over the TPM and its trusted software. Shubina, et al. [37] conclude that error handling for TPM primitives is a significant source of confusion for programmers, which has further limited the use of TPMs. Furthermore, flaws have been noted in the MTM specification [26] and attempts to address them are ongoing.

ARM's TrustZone [42] architecture also hopes to improve the situation, by integrating MTM-like secure processing and general purpose processing into a single chip. In addition to tighter integration, TrustZone supports additional features, including secure I/O to compatible peripherals (e.g., a secure display or keyboard). The future of both TrustZone and MTMs is unclear, but so far, neither has been widely adopted.

If MTMs do, at some point, become part of the mobile computing landscape, approaches like Flicker [30] provide an alternative to PnT; however, trusted computations in Flicker must occur while the OS is suspended, leading to competition that could render the mobile device unresponsive and unusable when under high load.

Additionally, PnT could be implemented using TrustZone or an MTM, rather than a smart card, for trusted processing. These alternatives, especially TrustZone, may provide better performance than current smart cards, while complicating the bootstrapping of the system. The MTM's root of trust would likely reside with the smart-phone manufacturer and/or service provider, whom the patient would have to trust with her private medical data.

The new Google Wallet technology [22] incorporates a secure coprocessor (the NXP PN65K) into the smart phone to support secure payment transactions over Near Field Communication (NFC). The PN65K provides storage, computation, and communication that is separate from the rest of the phone, and Android allows access only from the trusted Wallet app. It is not clear whether this chip could support other secure operations, such as health-data processing, but

it seems unlikely that the Wallet business partners would be willing to open this payment-focused technology to other applications. In any case, as above, any phone-embedded solution brings the handset manufacturer and network operator into the trust relationships that should be narrowly focused on the patient and the health provider.

6.3 Virtual machines

Another popular approach to securing computation is to run applications in a virtual machine (VM), where sandboxing can limit the duration and severity of some attacks. If the hypervisor is able to detect attacks, VMs can also simplify recovery by reloading a fresh VM image [9]. This approach is being advanced by a number of mobile-phone industry initiatives, including a partnership between VMWare and LG Electronics [44]; however, virtualization is not yet widely available for mobile phones.

If (or when) mobile phone-based virtualization becomes commonplace, sensitive or safety-critical sensor data could be processed by a trusted hypervisor, instead of a smart card, at the cost of diminished security. Virtualization is complicated and even small hypervisors [19,38] must deal with much more than security and privacy. Hypervisor complexity presents an unnecessarily large attack surface for attackers to search for vulnerabilities, and the consequences of a successful attack on a hypervisor are no less severe than a successful attack on a traditional operating system [43]. Protecting vulnerable software with additional layers of vulnerable software may provide enough security for some applications, but not highly-sensitive or safety-critical computation.

Green Hills Software claims to have developed a formally-verified "hacker-proof" hypervisor [25], that they market to military and government customers. We were not able to verify these claims; however, if these security guarantees prove to be true *and* cell-phone manufacturers were willing to virtualize their current OS on top of the Green Hills VM, this approach may provide an alternative to PnT.

Traditional VM-based approaches also have performance and usability disadvantages. Software-only approaches require a trusted software stack and inherit all of the challenges of the trusted platforms described in Section 6.2. A trusted VM also competes with other applications for resources, while PnT provides its own computing resources by replacing an existing microSD card with a combination microSD card and secure processor.

Additional variants have been proposed that improve performance. The "Divide" system from Enterproid [14] provides "dual-persona" capability to Android phones – a promising approach, which might be extended to support multiple persona, e.g., work, play, finance, and health. However, Divide is nonetheless an Android app and is thus vulnerable to rootkits and other mechanisms that compromise the underlying operating system.

The "Cells" approach goes further, allowing a smart phone to support multiple *virtual phones (VPs)*, each of which can run the full range of Android applications [5]. With much lower overhead than a VM or hypervisor approach, Cells would allow one VP to run mHealth applications and keep them isolated from other activity on the phone. On the other hand, the TCB is still pretty large and the effectiveness of separation and containment still depends on the security of the kernel and the Cells implementation. In that regard, relative to PnT, it has the same drawbacks as other virtualization methods.

6.4 Secure multiparty computation

Another closely-related approach to keeping sensitive computations private is to use a secure multiparty computing (SMC) system like FaeriePlay [24]. In addition to keeping data confidential, SMC systems typically employ garbled Boolean circuits (which also hide the computation being executed). While attractive for mHealth-related applications, evaluating programs as Boolean circuits comes at a high performance cost, limits the kinds of computations that an application can perform, and is difficult to verify.

Existing SMC systems overcome the verifiability challenges by either evaluating the entire program circuit, one gate at a time, inside a secure coprocessor like the IBM 4758 [24] – which is impractical for today's smart cards – or by using a homomorphic encryption scheme [18]; some partially homomorphic encryption schemes are more practical [32], perhaps even for use on mobile phones; however, it is not clear whether these techniques are expressive enough to support general-purpose computation. If approaches to SMC become more practical in the future, they will provide another attractive solution to this problem.

6.5 Control flow integrity

Finally, the path-hashing approach used in PnT is inspired by the field of Control Flow Integrity (CFI). CFI systems [1, 16] use techniques like code rewriting and shadow call stacks to ensure that executing programs execute within a defined control flow graph. While PnT shares many of the same goals with existing CFI approaches, all rely on the assumption that the attacker cannot modify application code. Since we assume a more powerful attacker who can modify application code, we have chosen to make control flow verifiable rather than preventing abuses.

7. DISCUSSION AND FUTURE WORK

This section discusses the limitations of Plug-n-Trust as well as the extensions that are left as future work.

7.1 Limitations

Hiding computation type

While PnT protects the integrity and confidentiality of sensed data and its processing, it does not hide the nature of the data processing, which might also reveal the patient's health condition. Some applications may be coded in a general way that limits the amount of leaked information, and garbled circuits (as mentioned in Section 6.4) could be used if and when SMC techniques become more efficient. Meanwhile, code obfuscation techniques [28] may provide a lower-cost alternative to make it significantly more difficult for an adversary to identify the nature of PnT's computations.

Data accessibility

One current limitation of PnT is the inability to *display data* to the user. With an attacker that has full control over the mobile phone, releasing data in a form that can be displayed to the user also makes it available to the attacker, violating our confidentiality goals. This limitation applies to all other solutions unless the OS can be trusted to be free from malware. In theory, smart cards with a secure input or output

channel (e.g., a button or light) can secure limited communications with the user [20]. (TrustZone [42] is capable of securing I/O and hence can display data without an attacker intercepting it, but has other limitations discussed above.) New smart cards from Go-Trust support Near-Field Communication (NFC), which may allow the card to communicate directly with nearby trusted peripherals [21].

The lack of data accessibility may also make debugging more difficult. Path hashing makes it possible to detect that a problem has occurred, but the specific attack or error may not be clear. This is a common challenge in secure systems, and techniques that embed more information about a specific error are needed to ease this tension.

Data-dependent operation

Another challenge is supporting *data-dependent communication*. An application that, in the interest of efficiency, only communicates when certain events occur (like an irregular heart beat), will unavoidably leak information: the presence of a message reveals information about the data. A potential compromise between efficiency and confidentiality would support data-dependent events masked by interspersed false events. Even with false event messages, this approach would likely be vulnerable to a variety of statistical attacks. This challenge is fundamental, because it represents a traffic-analysis attack on the data stream between the card and the back-end and there is no easy solution whenever the OS or phone cannot be trusted.

The analog of data-dependent communication is *data-dependent loop bounds*. PnT currently does not provide any in-card support for loops, though applications are free to execute commands in loops. When those loops depend on the value of confidential data, the number of iterations reveals information about the data, and could be used in PnT to discover the value of any arbitrary data element. Data-dependent loops with a bounded number of iterations can be implemented in PnT by iterating the maximum number of times, and using a conditional to turn the unneeded iterations into no-ops. This approach is potentially inefficient.

Another analog of data-dependent communication is *data-dependent sensor management*. In some applications, the results of data processing or aggregation within the MN may lead to a need to change the configuration of one or more SNs, e.g., to turn it on or off, to change the sampling rate, or to adjust the gain on a sensor. As with the above examples, feeding such commands back through the mApp would leak information into the untrusted MN. It would be possible, though expensive, for the card to return a vector of encrypted commands (one to be sent to each SN) each time it completed processing a batch of sensor data. Such a plan needs further investigation.

Cost

A search for current (late 2011) prices found microSD smart cards selling for around $60-$70 per card. These cards are new to the market and not in widespread use yet, while more widely deployed (non-microSD) smart cards cost only $1-$10. Mass production will have a similar effect on microSD smart cards; their price already dropped 80% over the past year.

Denial-of-Service (DoS)

An attacker that has compromised the phone can deny the application service in a wide variety of ways—to refuse delivery of messages, deny power to the card, and so forth—and there is nothing the card can do to prevent it. Instead of preventing DoS attacks, we hope to make them apparent, so that the attack can be remedied.

A variety of approaches could be used to detect the presence of a DoS attack. In most areas, mobile phones enjoy better than 90% network coverage and no data delivered for a long time might be a sign of DoS. If applications intentionally wait to deliver data, PnT could also send heart-beat messages. In any event, it is impossible for PnT to discern the difference between a DoS attack and a device failure. When a potential DoS attack is detected, notifying the patient will likely be necessary to determine the nature of the problem.

Theft or loss

An attacker may steal the smart card, the phone, or sensor nodes to learn about the patient's medical condition or to inject invalid data. Although PnT still protects the patient's privacy in face of theft (see Section 5.1), the attacker may carry the sensors, the phone, and the smart card and monitor her own medical condition, and submit the data to the back-end server, impersonating the owner of the smart card. In this case, reporting lost cards to the care provider will be the only effective measure.

To better counteract theft, the smart card should be able to authenticate the user. Since the phone can be malicious, the smart card and the patient cannot rely on the phone for mediating this authentication process. A direct method for the authentication involves some biometric, but at this time it is unfeasible to equip the card with a biometric sensor (e.g., fingerprint scanner) and necessary software. One could employ a physiology-based authentication scheme [40], which uses sensor data as a biometric to ensure authenticity.

Tolerance to data loss

Using sequence numbers and timestamps, PnT ensures detection of any re-ordering, repeating, or omission of sensor readings (within a given time window). While this feature is essential for critical monitoring applications, there are cases when some data loss should be tolerated. Since packet loss is not unusual in wireless communication, discarding all the sensor data because of one missing datum can significantly degrade data availability. Therefore, it is beneficial if PnT can support a computation task that is tolerant of a few missing data elements. We plan to extend our sequence-ensuring mechanism so that the computation is accepted if the number of missing data elements is below a certain threshold.

7.2 Extensions

There are several other extensions that we hope to address.

Complex computation support

PnT can easily be extended to support real numbers as well as integers. It can also be extended to support built-in operators, such as FFT or other signal-processing functions. We plan to add more mathematical operations such as exponentiation, logarithms, trigonometric functions, and set operations, and the ability to handle multi-datatype vectors.

However, in spite of these extensions, PnT is unlikely to fit all applications. Some computationally heavy and data intensive computations (like some machine-learning algorithms) may be too complex to perform in a small smart-

card, and too complex to verify via path hashing. These analyses may be done by the back-end server.

Medical actuator support

Another important extension will support medical actuators, like insulin pumps, without requiring the back-end service to be in the loop. For this to work, the actuator devices must be paired with the card (to exchange keys), and the device should be able to verify the result derived by the PnT card as the back-end server does, that is, by checking the path hash. Path hashes may be pre-computed and pre-installed on the actuator device, reducing the computational cost that verification would place on a resource-constrained platform.

Language and compiler support

Currently, applications interact with PnT by specifying individual PnT commands; the result looks something like assembly language. We plan to improve this situation by providing higher-level programming interfaces that allow system designers to describe how data should be processed without worrying about sealing and unsealing data, unrolling `if` statements, or forgetting to free allocated references.

Internal sensors

Finally, we plan to make the wealth of sensor data provided by a phone's internal sensors accessible to PnT. Without additional hardware, these sensors cannot be trusted if the phone's OS has been compromised; however, many attacks compromise only the application. We plan to support multiple levels of trust in PnT, allowing data from less-trusted sensors to be processed by more risk-tolerant applications.

Multiple-task support

In practice, the patient's condition may require multiple monitoring tasks to run simultaneously. For example, the mobile node may need to analyze ECG data continuously while blood-pressure level is monitored intermittently. We plan to extend PnT to support multiple monitoring tasks by allowing the SD card to maintain multiple contexts of computation, by introducing a task-reference by which the applications can specify the context of each operation.

8. SUMMARY

The connectivity and processing provided by patient-carried smart phones is critical to the success of many mHealth applications; however, as private and safety-critical data processing moves onto commodity smart-phones that are vulnerable to software-based attacks, security and privacy concerns must be addressed.

In this paper, we address this challenge in three ways: First, we describe the design of Plug-n-Trust (PnT), a novel and practical approach to protecting both the confidentiality and integrity of safety-critical medical sensing and data processing on vulnerable mobile phones. Second, our implementation and experiments show that PnT is feasible—using current Java-based smart cards and Android phones—for applications with low-to-medium data-rate requirements, and we identify opportunities for dramatic performance improvements that will allow PnT to support more data-intensive sensing applications. None of PnT's performance challenges are fundamental, and all will be eliminated as smart-card technology evolves. Third, we provide a security analysis demonstrating that PnT meets the desired security goals.

PnT is applicable to a wide range of applications, not limited to healthcare, and is amenable to a range of deployment scenarios that include secure elements built into phones as well as the plug-in card model we have proposed.

Acknowledgements

This research results from a research program at the Institute for Security, Technology, and Society at Dartmouth College, supported by the National Science Foundation under Grant Award Number 0910842 and by the Department of Health and Human Services (SHARP program) under award number 90TR0003-01. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the sponsors.

We also thank the anonymous reviewers, our shepherd Urs Hengartner, and our colleagues at Dartmouth for their valuable feedback.

9. REFERENCES

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information Systems Security*, 13(1), Nov. 2009. DOI 10.1145/1609956.1609960.

[2] O. Aciicmez, A. Latifi, J.-P. Seifert, and X. Zhang. A trusted mobile phone prototype. In *IEEE Consumer Communications and Networking Conference (CCNC)*, pages 1208–1209, Jan. 2008. DOI 10.1109/ccnc08.2007.270.

[3] S. Agarwal and C. T. Lau. Remote health monitoring using mobile phones and web services. *Telemedicine and e-Health*, 16(5):603–607, June 2010. DOI 10.1089/tmj.2009.0165.

[4] S. Ancoli-Israel, R. Cole, C. Alessi, M. Chambers, W. Moorcroft, and C. P. Pollak. The role of actigraphy in the study of sleep and circadian rhythms. *American Academy of Sleep Medicine Review Paper*, 26(3):342–392, 2003.

[5] J. Andrus, C. Dall, A. V. Hof, O. Laadan, and J. Nieh. Cells: a virtual mobile smartphone architecture. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 173–187. ACM, Nov. 2011. DOI 10.1145/2043556.2043574.

[6] A. Arcelus, R. Goubran, H. Sveistrup, M. Bilodeau, and F. Knoefel. Context-aware smart home monitoring through pressure measurement sequences. In *Proceedings of IEEE International Workshop on Medical Measurement and Applications (MEMEA)*, pages 32–37, Apr. 2010. DOI 10.1109/MEMEA.2010.5480223.

[7] T. Ball and J. R. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages Systems*, 16:1319–1360, July 1994. DOI http://doi.acm.org/10.1145/183432.183527.

[8] F. Buttussi and L. Chittaro. Smarter phones for healthier lifestyles: An adaptive fitness game. *IEEE Pervasive Computing*, 9(4):51–57, Oct. 2010. DOI 10.1109/MPRV.2010.52.

[9] L. P. Cox and P. M. Chen. Pocket hypervisors: Opportunities and challenges. In *Proceedings of the IEEE Workshop on Mobile Computing Systems and Applications (HotMobile)*, pages 46–50. IEEE, 2007. DOI 10.1109/HotMobile.2007.20.

[10] S. Coyle, F. Benito-Lopez, R. Byrne, and D. Diamond. On-body chemical sensors for monitoring sweat. In *Wearable and Autonomous Biomedical Devices and Systems for Smart Environment*, volume 75 of *Lecture Notes in Electrical Engineering*, pages 177–193. Springer, 2010. DOI 10.1007/978-3-642-15687-8_9.

[11] L. Cranor and S. Garfinkel. *Security and Usability*. O'Reilly Media, Inc., 2005.

[12] K. Dietrich and J. Winter. Towards customizable, application specific mobile trusted modules. In *Proceedings*

of the ACM Workshop on Scalable Trusted Computing (STC), pages 31–40. ACM, 2010. DOI 10.1145/1867635.1867642.

[13] A. Dua, N. Bulusu, W.-c. Feng, and W. Hu. Towards trustworthy participatory sensing. In *USENIX Conference on Hot Topics in Security*. USENIX, Aug. 2009. Online at http://www.usenix.org/event/hotsec09/tech/full_papers/dua.pdf.

[14] Enterproid.com. Enterproid: The Divide platform. Online at http://www.enterproid.com/features.php, visited Dec. 2011.

[15] J.-E. Ekberg, Nokia. Mobile Trusted Module (MTM) – an introduction. Online paper, 2007. Online at http://research.nokia.com/files/tr/NRC-TR-2007-015.pdf.

[16] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: software guards for system address spaces. In *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, pages 75–88. USENIX, Nov. 2006. Online at http://www.usenix.org/event/osdi06/tech/full_papers/erlingsson/erlingsson.pdf.

[17] Giesecke and Devrient GmbH. Online at http://www.gi-de.com/, visited Mar. 2011.

[18] R. Gennaro, C. Gentry, and B. Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In T. Rabin, editor, *Advances in Cryptology (CRYPTO)*, volume 6223, pages 465–482. Springer, 2010. DOI 10.1007/978-3-642-14623-7_25.

[19] P. Gilbert, L. P. Cox, J. Jung, and D. Wetherall. Toward trustworthy mobile sensing. In *Proceedings of the Workshop on Mobile Computing Systems & Applications (HotMobile)*, pages 31–36. ACM, Feb. 2010. DOI 10.1145/1734583.1734592.

[20] H. Gobioff, S. Smith, J. D. Tygar, and B. Yee. Smart cards in hostile environments. In *Proceedings of the USENIX Workshop on Electronic Commerce*, pages 23–28, 1996. Online at http://www.cs.dartmouth.edu/~sws/pubs/gsty96.pdf.

[21] Go-Trust NFC smart cards. Online at http://www.go-trust.com/products/swp-secure-microsd/, visited Dec. 2011.

[22] Google. Google wallet. Online at http://www.google.com/wallet/, visited Dec. 2011.

[23] D. Halperin, T. S. Heydt-Benjamin, B. Ransford, S. S. Clark, B. Defend, W. Morgan, K. Fu, T. Kohno, and W. H. Maisel. Pacemakers and implantable cardiac defibrillators: Software radio attacks and zero-power defenses. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, pages 129–142. IEEE Press, May 2008. DOI 10.1109/SP.2008.31.

[24] A. Iliev and S. W. Smith. Small, Stupid, and Scalable: Secure Computing with Faerieplay. In *Proceedings of the ACM Workshop on Scalable Trusted Computing (STC)*, 2010.

[25] D. Kleidermacher. Next generation secure mobile devices. *Information Quarterly*, 7(4), 2008.

[26] R. Korthaus, A. R. Sadeghi, C. Stüble, and J. Zhan. A practical property-based bootstrap architecture. In *Proceedings of the ACM Workshop on Scalable Trusted Computing (STC)*, pages 29–38. ACM, 2009. DOI 10.1145/1655108.1655114.

[27] D. Kotz. A threat taxonomy for mHealth privacy. In *Proceedings of the Workshop on Networked Healthcare Technology (NetHealth)*. IEEE Press, Jan. 2011. DOI 10.1109/COMSNETS.2011.5716518.

[28] D. Low. Protecting Java code via code obfuscation. *Crossroads*, 4(3):21–23, Apr. 1998. DOI 10.1145/332084.332092.

[29] S. Mare, J. Sorber, M. Shin, C. Cornelius, and D. Kotz. Adapt-lite: Privacy-aware, secure, and efficient mhealth sensing. In *Proceedings of the Workshop on Privacy in the Electronic Society (WPES)*, Oct. 2011. DOI 10.1145/2046556.2046574.

[30] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: an execution infrastructure for TCB minimization. *SIGOPS Operating Systems Review*, 42:315–328, Apr. 2008. DOI 10.1145/1352592.1352625.

[31] Mobile Phone Work Group. Mobile Trusted Module FAQ. Online at http://www.trustedcomputinggroup.org/resources/mobile_trusted_module_faq, visited Dec. 2010.

[32] M. Naehrig, K. Lauter, and V. Vaikuntanathan. Can homomorphic encryption be practical? In *Proceedings of the ACM Workshop on Cloud Computing Security*, pages 113–124. ACM, Oct. 2011. DOI 10.1145/2046660.2046682.

[33] SIMalliance. SIMaLliance Web Site. Online at http://www.simalliance.org, visited Dec. 2011.

[34] S. Saroiu and A. Wolman. I am a sensor, and I approve this message. In *Proceedings of the Eleventh Workshop on Mobile Computing Systems & Applications (HotMobile)*, HotMobile '10, pages 37–42. ACM, 2010. DOI 10.1145/1734583.1734593.

[35] L. A. Saxon, D. L. Hayes, F. R. Gilliam, P. A. Heidenreich, J. Day, M. Seth, T. E. Meyer, P. W. Jones, and J. P. Boehmer. Long-term outcome after ICD and CRT implantation and influence of remote device follow-up: The ALTITUDE survival study. *Circulation*, 122(23):2359–2367, Dec. 2010. DOI 10.1161/CIRCULATIONAHA.110.960633.

[36] Open source. Seek-for-android. Online at https://code.google.com/p/seek-for-android/wiki/PCSCLite, visited Apr. 2011.

[37] A. Shubina, S. Bratus, W. Ingersol, and S. W. Smith. The diversity of TPMs and its effects on development: a case study of integrating the TPM into OpenSolaris. In *Proceedings of the ACM Workshop on Scalable Trusted Computing (STC)*, pages 85–90. ACM, 2010. DOI 10.1145/1867635.1867649.

[38] L. Singaravelu, C. Pu, H. HÃd'rtig, and C. Helmuth. Reducing TCB complexity for security-sensitive applications: Three case studies. In *Proceedings of European Conference on Computer Systems*, pages 161–174, 2006.

[39] E. G. Sirer, W. de Bruijn, P. Reynolds, A. Shieh, K. Walsh, D. Williams, and F. B. Schneider. Logical attestation: an authorization architecture for trustworthy computing. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, pages 249–264. ACM, Nov. 2011. DOI 10.1145/2043556.2043580.

[40] J. Sriram, M. Shin, T. Choudhury, and D. Kotz. Activity-aware ECG-based patient authentication for remote health monitoring. In *Proceedings of the International Conference on Multimodal Interfaces and Workshop on Machine Learning for Multi-modal Interaction (ICMI-MLMI)*, pages 297–304. ACM Press, Nov. 2009. DOI 10.1145/1647314.1647378.

[41] Trusted Computing Group. TPM Main Specification Level 2 Version 1.2, Rev 103. Online at http://www.trustedcomputinggroup.org/resources/tpm_main_specification, visited Dec. 2010.

[42] TrustZone. ARM TrustZone web site. Online at http://www.arm.com/products/processors/technologies/trustzone.php, visited Dec. 2010.

[43] S. Vaughan-Nichols. Virtualization sparks security concerns. *Computer*, 41(8):13 –15, Aug. 2008. DOI 10.1109/MC.2008.276.

[44] VMWare, Inc. and LG Electronics MobileComm U.S.A, Inc. VMWare News Release, Dec. 7, 2010. Online at http://www.vmware.com/company/news/releases/vmware-lge-partnership.html, visited Dec. 2010.

[45] A. Wolman, S. Saroiu, and V. Bahl. Using trusted sensors to monitor patients' habits. In *USENIX Workshop on Health Security and Privacy*. USENIX Association, Aug. 2010. Online at http://research.microsoft.com/en-us/um/people/alecw/healthsec-2010.pdf.

[46] F. Zhou, H.-I. Yang, J. Álamo, J. Wong, and C. Chang. Mobile personal health care system for patients with diabetes. In Y. Lee, Z. Bien, M. Mokhtari, J. Kim, M. Park, J. Kim, H. Lee, and I. Khalil, editors, *Aging Friendly Technology for Health and Independence*, volume 6159 of *Lecture Notes in Computer Science*, chapter 12, pages 94–101. Springer, 2010. DOI 10.1007/978-3-642-13778-5_12.