IoT NOTARY: Sensor Data Attestation in Smart Environment

Nisha Panwar, Shantanu Sharma, Guoxi Wang, Sharad Mehrotra, Nalini Venkatasubramanian, Mamadou H. Diallo, and Ardalan Amiri Sani

Abstract—Contemporary IoT environments, such as smart buildings, require end-users to trust data-capturing rules published by the systems. There are several reasons why such a trust is misplaced — IoT systems may violate the rules deliberately or IoT devices may transfer user data to a malicious third-party due to cyberattacks, leading to the loss of individuals' privacy or service integrity. To address such concerns, we propose IoT NOTARY, a framework to ensure trust in IoT systems and applications. IoT NOTARY provides secure log sealing on live sensor data to produce a verifiable 'proof-of-integrity,' based on which a verifier can attest that captured sensor data adheres to the published data-capturing rules. IoT NOTARY is an integral part of TIPPERS, a smart space system that has been deployed at UCI to provide various real-time location-based services in the campus. IoT NOTARY imposes nominal overheads for verification, thereby users can verify their data of one day in less than two seconds.

Index Terms—IoT, sensor data, smart buildings, WiFi.

1 INTRODUCTION

IoT devices (*e.g.*, camera, WiFi access-points, cell phones, bodyworn sensors, occupancy sensors, temperature sensors, and light sensors) capture user-related data for empowering existing systems with new capabilities. While fine-grained continuous monitoring by IoT devices (*e.g.*, camera and WiFi access-points) offers numerous benefits and empowers existing systems with new capabilities, it also raises several privacy and security concerns (*e.g.*, smoking habits, gender, and religious belief). To highlight the privacy concern, we first share our experience in building location-based services at UC Irvine using WiFi connectivity data.

Use-case: University WiFi data collection. In our on-going project, entitled TIPPERS [2], we have developed a variety of location-based services based on WiFi connectivity dataset. At UC Irvine, more than 2000 WiFi access-points and four WLAN controllers (managed by the university IT department) provide campus-wide wireless network coverage. Whenever a device connects to the campus WiFi network (through an access-point), the access-point generates Simple Network Management Protocol (SNMP) trap for this association event. Each association event contains access-point-id, s_i, user device MAC address, d_j, and the time of the association, t_k. All SNMP traps $\langle s_i, d_j, t_k \rangle$ are sent to access-point's controllers in realtime. The access-point controller anonymizes device MAC addresses (to preserve the privacy of users in the campus).

TIPPERS collects WiFi connectivity data from one of the controllers that manage 490 access-points and receives $\langle s_i, d_j, t_k \rangle$ tuples for each connectivity event. However, before receiving any WiFi data, TIPPERS notifies all WiFi users about the data-capture rules by sending emails over a university mailing list. Subsequently, based on WiFi connectivity data $\langle s_i, d_j, t_k \rangle$, TIPPERS provides various realtime applications. Some of these services, *e.g.*, computing occupancy levels of (regions in) buildings in the form of a live heatmap, require only anonymous data. Other services, *e.g.*, providing location information (within buildings) or contextualized messaging (to provide messages to a user when he/she is in the vicinity of the desired location), require user's original disambiguated data. To date, over one hundred users have registered into TIPPERS to utilize realtime services. A key requirement imposed by the university in sharing data with TIPPERS is that the system supports provable mechanisms to verify that individuals have been notified prior to their data (anonymized or not) being used for service provisioning. Also, an option for users to opt-out of sharing their WiFi connectivity data with TIPPERS must be supported. If users opt-out, the system must prove to the users that indeed their data was not shared with TIPPERS. TIPPERS use immutable log-sealing to help all users to verify that the captured data is consistent with pre-notified data-capture rules.

Our experience in working with various groups in the campus is that (persistent) location data can be deemed quite sensitive by certain individuals with concerns about the spied upon by the administration or by others. Thus, mechanisms for notification of data-capture rules, secure log-sealing, and verification components made a sub-framework, entitled IoT NOTARY, which has become an integral part of TIPPERS.

Data-capture concerns in IoT environments are similar to that in mobile computing, where mobile applications may have continuous access to resident sensors on mobile devices. In the mobile setting, data-capture rules and permissions [3] are used to control data access, *i.e.*, which applications have access to which data generated at the mobile device (*e.g.*, location and contact list) for which purpose and in which context. At the abstract level, a data-capture rule informs the user about the nature of personally identifying information that an application captures, *i.e.*, for what *purpose* and in what *context*. However, in IoT settings, the data-capture framework differs from that in the mobile settings, in two important ways:

1) Unlike the mobile setting, where applications can seek user's permission at the time of installation, in IoT settings, there are no obvious mechanisms/interfaces to seek users'

A preliminary version of this work was accepted in IEEE NCA 2019 [1]. Please see the cover letter to find the details about the new content added in this version.

The authors are with University of California, Irvine, USA.

preferences about the data being captured by sensors of the smart environment. Recent work [4] has begun to explore mechanisms using which environments can broadcast their data-capture rules to users and seek their explicit permissions.

2) Unlike the mobile setting, users cannot control sensors in IoT settings. While in mobile settings, a user can trust the device operating system not to violate the data-capture rules, in IoT settings, trust (in the environment controlling the sensors) may be misplaced. IoT systems may not be honest or may inadvertently capture sensor data, even if data-capture rules are not satisfied [5], [6].

We focus on the above-mentioned second scenario and determine ways to provide trustworthy sensing in an untrusted IoT environment. Thus, the users can verify their data captured by IoT environment based on pre-notified data-capture rules. Particularly, we deal with three sub-problems, namely *secure notification* to the user about data-capture rules, *secure (sensor data) log-sealing* to retain *immutable* sensor data, as well as, data-capture rules, and *remote attestation* to verify the sensor data against pre-notified data-capture rules by a user, without being heavily involved in the attestation process.

Our contribution and outline of the paper. We provide:

- A user-centric framework (§4.2) to ensure trustworthy data collection in untrusted IoT spaces, entitled IoT NOTARY that contains three entities (§3.1): (*i*) *infrastructure deployer* that installs sensors, (*ii*) a *service provider* (SP, *e.g.*, TIPPERS) that establishes a list of data-capture rules that dictates the condition under which a sensor's data can/cannot be utilized to provide realtime services to the user, and (*iii*) *users* that use services provided by sensors, as well as, by SP, (if interested).
- Two models to inform the user about the data-capture rules (§5.1): notice-only model and notice-and-ACK model.
- A secure log-sealing mechanism (§5.2) implemented by secure hardware that cryptographically retains logs, data-capture rules, sensors' state, and contextual information to generate a *proof-of-integrity* in an immutable fashion.
- Optimized secure log-sealing mechanisms that regard the sensor state and user-device-id (§5.2.3 and §5.2.4), implemented by secure hardware to reduce the size of secure log.
- A secure attestation mechanism (§5.3), mixed with SIGMA protocol [7], allowing a verifier (a user or a *non-mandatory* auditor) to securely attest the sealed logs as per the data-capture rules.
- Implementation results of IoT NOTARY on the university live WiFi data in §6.

2 Comparison with Existing Work

We classify the related work in the scope of IoT attestation into the following three categories:

Attestation in the context of IoT. The existing remote attestation protocols verify the internal memory state of untrusted devices through a trusted remote verifier. For example, AID [8] attests the internal state of neighboring devices through key exchange and proofs-of-non-absence. In AID, the adversary can compromise communication channels, the internal state of the device, and the cryptographic keys. SEDA [9] attests low-end embedded devices in a swarm and provides the number of devices that pass attestation. However, SEDA attests neighboring peer devices only. Similarly, DARPA [10] and SANA [11] allow detection of physical attacks by using heartbeat messages and provide aggregate network attestation, with high computation and communication costs, which are quadratic in the network size. SMARM [12] protects against malware by scanning memory in a secret randomized order. However, it may require many iterations to eventually detect malware. RADIS [13] assumes that a compromised IoT service can influence genuine operations of other services without changing the software. Thus, RADIS provides control-flow attestation for distributed services among the interconnected IoT devices. In short, all such work only deals with attestation of sensor devices, and their methods cannot be used to verify sensor data against the data-capture rules.

In contrast, IoT NOTARY does not deal with the verification of internal state of sensor devices, since in our case, (WiFi access-point) sensors are assumed to be deployed by a trusted entity (*e.g.*, the university IT department). Of course, cyberattacks are possible on sensors to maliciously record data and that can also be detected by IoT NOTARY, while not verifying the sensors.

Attestation using secure hardware. [14] provided SGX-based attestation method for physical attacks on the sensor, *e.g.*, modifying memory and changing I/O signals. Fiware [15] provides secure key management through key vault running in SGX, thereby provides an alternative to PKI-based solutions. However, [14], [15] are unable to verify any sensor data. In [14], [15], if data-capture rules are mis-notified to the user, SGX cannot detect any inconsistency.

In contrast, IoT NOTARY does not deal with attacks on sensors, as well as, a specific key management protocol. However, IoT NOTARY can detect and discard the sensor data that does not comply with the notifications released earlier.

Integrity verification. [16], [17] proposed a privacy-preserving scheme based on zero-knowledge proofs to detect log-exclusion attacks. [16] provided solutions for accountability and auditing through hierarchical multi-party computation (MPC) and succinct zero-knowledge proof statements. [18] considered verification process for MPC using a trusted-third-verifier. [17] provided a privacy-preserving certificate transparency service, which signs a message four times, where an auditor can trace the certified logs. Other techniques have proposed the concept of *excerpts* and *snapshots* for log integrity verification. For example, in [19] used hash-chains for integrity protection and identity-based searchable encryption. [20] proposed a Merkle tree-based history tree to prove the sequence of logs over time. [21] proposed a Bloom tree that stores proof of logs at an untrusted cloud. Further, access-pattern-hiding cryptographic techniques [22], [23] may be used to verify any stored log, since an adversary cannot skip the computation on some parts of the data, due to executing an identical computation on each sensor reading. Techniques, *e.g.*, function secret-sharing [22] or vSQL [23], may be used to verify query results on cleartext. However, these techniques cannot detect log deletion. Also, all such techniques incur signification time. For example, vSQL took more than 4000 seconds to verify a SQL query. In addition, any end-to-end encryption model is not sufficient for verification.

In contrast, IoT NOTARY provides a complete security to sensor data and realtime data attestation approach. Unlike [17], IoT NOTARY requires only two signatures per file, where one is used to validate log completeness, and another is used to validate the log ordering with respect to adjacent logs.

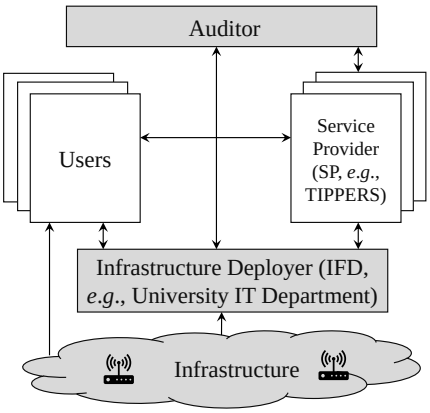

Figure 1: Entities in IoT NOTARY.

3 MODELING IOT DATA ATTESTATION

3.1 Entities

Our model has the following entities, see Figure 1:

Infrastructure Deployer (IFD). IFD (which is the university IT department in our use-case; see §1) deploys and owns a network of p sensors devices (denoted by $s_1, s_2, \ldots, s_p$), which capture information related to users in a space. The sensor devices could be: (*i*) dedicated sensing devices, *e.g.*, energy meters and occupancy detectors, or (*ii*) facility providing sensing devices, *e.g.*, WiFi access-points and RFID readers. Our focus is on facility providing sensing devices, especially WiFi access-points that also capture some user-related information in response to services. E.g., WiFi access-points capture the associated user-device-ids (MAC addresses), time of association, some other parameters (such as signal strength, signal-to-noise ratio); denoted by: $\langle d_i, s_j, t_k, param \rangle$, where d_i is the i^{th} user-device-id, s_j is the j^{th} sensor device, t_k is k^{th} time, and $param$ is other parameters (we do not deal with $param$ field and focus on only the first three fields). All sensor data is collected at a controller (server) owned by IFD. The controller may keep sensor data in cleartext or in encrypted form; however, it only sends encrypted sensor data to the service provider.

Service Providers (SP). SP (which is TIPPERS in our use-case; see §1) utilizes the sensor data of a given space to provide different *services*, *e.g.*, monitoring a location and tracking a person. SP receives encrypted sensor data from the controller.

Data-capture rules. SP establishes data-capture rules (denoted by a list DC having different rules $dc_1, dc_2, \ldots, dc_q$). Data-capture rules are conditions on device-ids, time, and space. Each data-capture rule has an associated *validity* that indicates the time during which a rule is valid. Data-capture rules could be to capture user data by default (unless the user has explicitly opted out). Alternatively, default rules may be to opt-out, unless, users opt-in explicitly. Consider a default rule that individuals on the 6^{th} floor of the building will be monitored from 9pm to 9am. Such a rule has an associated condition on the time and the id of the sensor used to generate the data. Now, consider a rule corresponding to a user with a device d_i opting-out of data capture based on the previously mentioned rule. Such an opt-out rule would have conditions on the user-id, as well as, on time and the sensor-id. For sensor data for which a default data-capture rule is opt-in, the captured data is forwarded to SP, if there does not exist any associated opt-out rules, whose associated conditions are satisfied by the sensor data. Likewise, for sensor data where the default is opt-out, the data is forwarded to SP only, if there exists an explicit opt-in condition. We refer to the sensor data to have a *sensor state* ($s_i.state$ denotes the state of the sensor s_i) of 1 (or active), if the data can be forwarded to SP; otherwise, 0 (or passive). In the remaining paper, unless explicitly noted, opt-out is considered as the default rule, for simplicity of discussion.

Whenever SP creates a new data-capture rule, SP must send a *notice message* to user devices about the current usage of sensor data (this phase is entitled *notification phase*). SP uses Intel Software Guard eXtension (SGX) [24], which works as a trusted agent of IFD, for securely storing sensor data corresponding to data-capture rules. SGX keeps all valid data-capture rules in the secure memory and only allows to keep such data that qualifies pre-notified valid data-capture rules; otherwise, it discards other sensor data. Further, SGX creates immutable and verifiable logs of the sensor data (this phase is entitled *log-sealing phase*). The assumption of secure hardware at a machine is rational with the emerging system architectures, *e.g.*, Intel machines are equipped with SGX [25]. However, existing SGX architectures suffer from side-channel attacks, *e.g.*, cache-line, branch shadow, page-fault attacks [26], which are outside the scope of this paper.

Users. Let $d_1, d_2, \ldots, d_m$ be m (user) devices carried by $u_1, u_2, \ldots, u_{m'}$ users, where $m' \leq m$. Using these devices, users enjoy services provided by SP. We define a term, entitled *user-associated data*. Let $\langle d_i, s_j, t_k \rangle$ be a sensor reading. Let d_i be the i^{th} device-id owned by a user u_i. We refer to $\langle d_i, s_j, t_k \rangle$ as user-associated data with the user u_i. Users worry about their privacy, since SP may capture user data without informing them, or in violation of their preference (*e.g.*, when the opt-out was a default rule or when a user opted-out from an opt-in default). Users may also require SP to prove service integrity by storing all sensor data associated with the user (when users have opted-in into services), while minimally being involved in the attestation process and storing records at their sides (this phase is entitled *attestation phase*).

Auditor. An auditor is a *non-mandatory* trusted-third-party that can (periodically) verify entire sensor data against data-capture rules. Note that a user can only verify his/her data, not the entire sensor data or sensor data related to other users, since it may reveal the privacy of other users.

3.2 Threat Model

We assume that SP and users may behave like adversaries. The adversarial SP may *store* sensor data without informing data-capture rules to the user. The adversarial SP may *tamper* with the sensor data by inserting, deleting, modifying, and truncating sensor readings and secured-logs in the database. By tampering with the sensor data, SP may *simulate* the sealing function over the sensor data to produce secured-logs that are identical to real secured-logs. Thus, the adversary may hinder the attestation process and make it impossible to detect any tampering with the sensor data by the verifier (that may be an auditor or a user). Further, as mentioned before that SP utilizes sensor data to provide services to the user. However, an adversarial SP may provide *false answers* in response to user queries. We assume that the adversarial SP cannot obtain the secret key of the enclave (by any means of side-channel attacks on SGX). Since we assumed that sensors are trusted and cannot be spoofed, we do not need to consider a case when sensors would collude with SP to fabricate the logs.

An adversarial user may *repudiate* the reception of notice messages about data-capture rules. Further, an adversarial user may *impersonate* a real user, and then, may retrieve the sensor data and secured-log during the verification phase. This way an adversarial user may reveal the privacy of the users by observing sensor data. Further, a user may infer the identity of other users associated with sensor data by potentially launching *frequency-count attacks* (*e.g.*, by determining which device-ids are prominent).

3.3 Security Properties

In the above-mentioned adversarial model, an adversary wishes to learn the (entire/partial) data about the user, without notifying or by mis-notifying about data-capture rules, such that the user/auditor cannot detect any inconsistency between data-capture rules and stored sensor data at SP. Hence, a secure attestation algorithm must make it detectable, if the adversary stores sensor data in violation of the data-capture rules notified to the user. To achieve a secure attestation algorithm, we need to satisfy the following properties:

Authentication. Authentication is required: (*i*) between SP and users, during notification phase; thus, the user can detect a rogue SP, as well as, SP can detect rogue users, and (*ii*) between SP and the verifier (auditor/user), before sending sensor data to the verifier to prevent any rogue verifier to obtain sensor data. Thus, authentication prevents threats such as impersonation and repudiation. Further, a periodic mutual authentication is required between IFD and SP, thereby discarding rogue sensor data by SP, as well as, preventing any rogue SP to obtain real sensor data.

Immutability and non-identical outputs. We need to maintain immutability of notice messages, sensor data, and the sealing function. Note that if the adversary can alter notice messages after transmission, it can do anything with the sensor data, in which case, sensor data may be completely stored or deleted without respecting notice messages. Further, if the adversary can alter the sealing function, the adversary can generate a proof-of-integrity, as desired, which makes the flawless attestation impossible. The output of the sealing function should not be identical for each sensor reading to prevent an adversary to forge the sealing function (and to prevent the execution of frequency-count attack by the user). Thus, immutability and non-identical outputs properties prevent threats, *e.g.*, inserting, deleting, modifying, and truncating the sensor data, as well as, simulating the sealing function.

Minimality, non-refutability and privacy-preserving verification. The verification method must find any misbehavior of SP, during storing sensor data inconsistent with pre-notified data-capture rules. However, if the verifiers wish to verify a subset of the sensor data, then they should not verify the entire sensor data. Thus, SP should send a minimal amount of sensor data to the verifier, enabling them to attest what they wish to attest. Further, the verification method: (*i*) cannot be refuted by SP, and (*ii*) should not reveal any additional information to the user about all the other users during the verification process. These properties prevent SP to store only sensor data that is consistent with the data-capture rules notified to the user. Further, these properties preserve the privacy of other users during attestation and impose minimal work on the verifier.

3.4 Assumptions

This section presents assumptions, we made, as follows:

1) The sensor devices are assumed to be computationally-inefficient to locally generate a verifiable log for the continuous data stream as per the data-capture rules.
2) Sensor devices are tamper-proof, and they cannot be replicated/spoofed (*i.e.*, two devices cannot have an identical id). In short, we assume a correct identification of sensors, before accepting any sensor-generated data at the controller at IFD, and it ensures that no rogue sensor device can generate the data on behalf of an authentic sensor. Further, we assume that an adversary cannot deduce any information from the dataflow between a sensor and the controller. Recall that in our setting the university IT department collects the entire sensor data from their owned and deployed sensors, before sending it to TIPPERS.
3) We assume the existence of an authentication protocol between the controller and SP, so that SP receives sensor data only from authenticated and desired controller.
4) The communication channels between SP and users, as well as, between SP and auditor are insecure. Thus, our solution incorporates an authenticated key exchange based on SIGMA protocol (which protects sender identity). When the verifier's identity is proved, the cryptographically sealed logs are sent to the verifier.
5) By any side-channel attacks on SGX, one cannot tamper with SGX and retrieve the secret-key of SGX. (Otherwise, the adversary can simulate the sealing process.)

Definition: Attestation Protocol. Now, we define an attestation protocol that consists of the following components:

• $Setup()$: Given a security parameter 1^k, $Setup()$ produces a public key of the enclave (PK_E) and a corresponding private key (PR_E), used by the enclave to securely write sensor logs.

• $Sealing(PR_E, \langle d_i, s_j, s_j.state, t_k\rangle, dc_l)$: Given the key PR_E, $Sealing()$ (which executes inside the enclave) produces a verifiable *proof-of-(log)-integrity* ($\mathcal{PI}$) and *proof-of-integrity for user/service (query) verification* ($\mathcal{PU}$), based on the received sensor readings and the current data-capture-rule, dc_l.

• $Verify(PK_E, \langle *, s_j, s_j.state, t_k, \mathcal{PI}, dc_l\rangle, Sealing(PR_E, \langle d_i, s_j, s_j.state, t_k\rangle, dc_l))$: Given the public key PK_E, sensor data, proof, and data-capture rule, $Verify()$ is executed at the verifier, where $*$ denotes the presence/absence of a user-device-id, based on dc_l. $Verify()$ produces 1, iff $\mathcal{PI} = Verify(PK_E, \langle *, s_j, s_j.state, t_k, dc_l\rangle, Sealing(PR_E, \langle d_i, s_j, s_j.state, t_k\rangle, dc_l))$; otherwise, 0. Similarly, $Verify()$ can attest $\mathcal{PU}$.

Note that the functions $Sealing()$ and $Verify()$ are known to the user, auditor, and SP. However, the private key PR_E is only known to the enclave.

4 IoT Notary: Challenges and Approach

This section provides challenges we faced during IoT Notary development and an overview of IoT Notary.

4.1 Challenges and Solutions

We classify the problem of data attestation in IoT into three categories: (*i*) secure notification, (*ii*) log-sealing, and (*iii*) log verification. Before going into details, we provide the challenges that we faced during the development and the way we overcome those challenges (Figure 2), as follows:

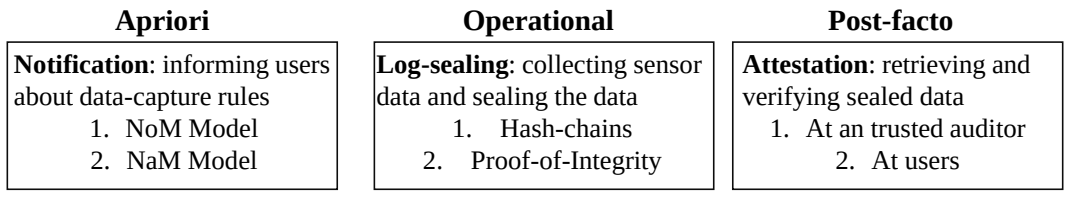

Figure 2: Phases in IoT NOTARY.

C1. Secured notification for data-capture rules. The declaration of data-capture rules requires a reliable and secure notification scheme, thereby users can verify the sender of notice messages. Trivially, this can be done through a unique key establishment between each user and SP. However, this incurs a major overhead at SP, as well as, SP can send different messages to different users.
Solution. To address the above challenge, we develop three solutions: One is based on secure notifier that delivers a cryptographically encrypted notice message, which is signed by the enclave, to all the users (see §5.1). The second solution uses time-based one-time passwords [27], [28] that remain valid for only a specific duration. The users can verify these temporal passwords to authenticate the sender, and hence, the validity of notifications sent (see Appendix A). The third solution uses an acknowledgment from the user and does not need any trusted notifier (see §5.1).

C2. Tamper-proof cryptographically secured-log sealing. The verification process depends on immutable sensor data that is stored at SP. A trivial way is to store the entire data using a strong encryption technique, mixed with access-pattern hiding mechanisms. While it will prevent tampering with the data (except deletion), SP cannot develop realtime services on this data. Thus, the first challenge is how to store sensor data at SP according to data-capture rules; hence, the verifier can attest the data. The second challenge arises due to the computational cost at the verifier and communication cost between the verifier and SP to send the verifiable sensor data; *e.g.*, if the verifier wishes to attest only one-day old sensor data over the sensor data of many years, then sending many years of sensor data to the verifier is impractical. Finally, the last challenge is how to securely store data when data-capture rules are set to be false (*i.e.*, not to store data). In this case, not storing any data would not provide a way for verification, since SP may store data and can inform the verifier that there is no data as per the existing data-capture rules.
Solution. To address the first challenge, we develop a cryptographic sealing method based on hash-chains and XOR-linked-lists to ensure immutable sensor logs, after the sealed logs leave the enclave. Thus, any log addition/deletion/update is detectable (see §5.2.1). To address the second challenge, we execute sealing on small-sized chunks, which each maintains its hash-chain and XOR-links. The XOR-links ensure the log completeness, *i.e.*, a chunk before and after the desired chunk has not been altered (see §5.2.1). To address the third challenge, we store the *device state* of the first sensor-reading for which the data-capture rule is set to false. Further, we discard all subsequent sensor-readings, unless finding a sensor-reading for which data-capture rule is to *store data* (§5.2.3).

C3. Privacy-preserving log verification. In case of log-integrity verification, SP can provide the entire sensor data with cryptographically sealed log to the trusted auditor. But, the challenge arises, if a user asks to verify her user-associated data/query results. Here, SP cannot provide the entire sensor data to the user, since it will reveal other users' privacy.
Solution. To address this challenge, we develop a technique to non-deterministically encrypt the user-id before cryptographically-sealing the sensor data. However, only non-deterministic encryption is also not enough to verify the log completeness, we compute XOR operation on each sensor reading, and then, create XOR-linked-list (see §5.2.2).

4.2 IoT NOTARY: An Overview

This section presents an overview of the three phases and dataflow among different entities and devices, see Figure 3.

Notification phase: SP to Users messages. This is the first phase that notifies users about data-capture rules for the IoT space using notice messages (in a verifiable manner for later stages). Such messages can be of two types: (*i*) notice messages, and (*ii*) notice-and-acknowledgment messages. SP establishes (the default) data-capture rules and informs trusted hardware (❶). Trusted hardware securely stores data-capture rules (❷, ❺) and informs the *trusted notifier* (❸) that transmits the message to all users (❹). Only notice messages need a trusted notifier to transmit the message (see §5.1).

Log-sealing phase: Sensor devices to SP messages. Each sensor sends data to the controller (❶). The controller receives the correct data, generated by the actual sensor, as per our assumptions (and settings of the university IT department). The controller sends encrypted data to SP (❻) that authenticates the controller using any existing authentication protocol, before accepting data. Trusted hardware (Intel SGX) at SP reads the encrypted data in the enclave (❼).
Working of the enclave. The enclave decrypts the data and checks against the pre-notified data-capture rules. Recall that the decrypted data is of the format: $\langle d_i, s_j, t_k \rangle$, where d_i is i^{th} user-device-id, s_j is the j^{th} sensor device, and t_k is k^{th} time. After checking each sensor reading, the enclave adds a new field, entitled *sensor (device) states*. The sensor state of a senor s_j is denoted by $s_j.state$, which can be `active` or `passive`, based on capturing user data. For example, $s_j.state$ = `active` or (`1`), if data captured by the sensor s_j satisfies the data-capture rules; otherwise, $s_j.state$ = `passive` or (`0`). For all the sensors whose $state = 0$, the enclave deletes the data. Then, the enclave cryptographically seals sensor data, regardless of the sensor state, and provides cleartext sensor data of the format: $\langle d_i, s_j, s_j.state = 1, t_k \rangle$ to SP (❽) that provides services using this data (❾). Note that the cryptographically sealed logs and cleartext sensor data are kept at untrusted storage of SP (❽, ❿).

Verification phase: SP to verifier messages. In our model, an auditor and a user can verify the sensor data. The auditor can verify the entire/partial sensor data against data-capture rules by asking

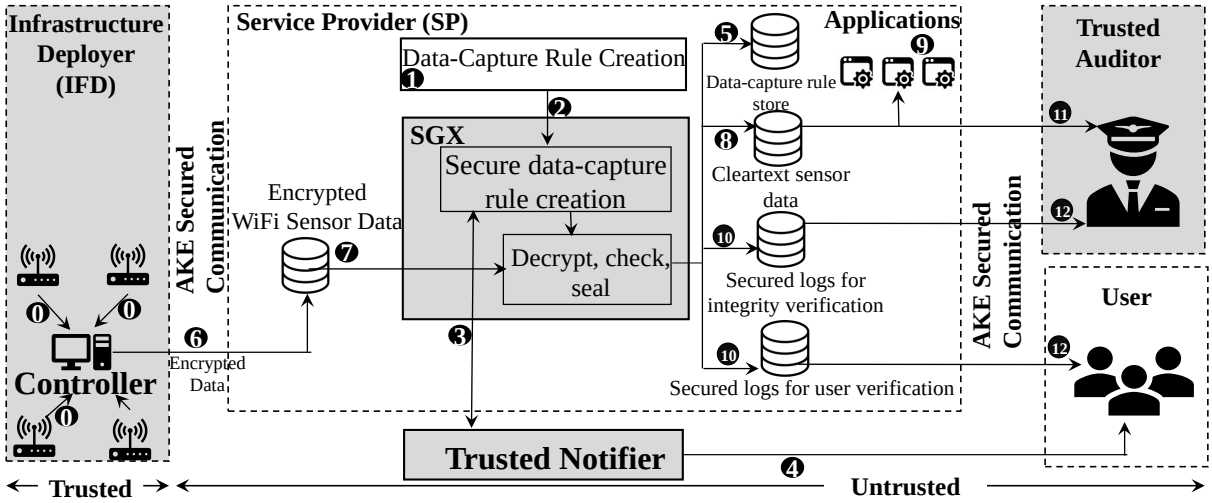

Figure 3: Dataflow and computation in the protocol. Trusted parts are shown in shaded box.

SP to provide cleartext sensor data and cryptographically sealed logs (❽, ❿). The users can also verify their own data against pre-notified messages or can verify the results of the services provided by SP using only cryptographically sealed logs (⓬). Note that using an underlying authentication technique (as per our assumptions), auditor/users and SP authenticate each other before transmitting data from SP to auditor/users.

5 Attestation Protocol

This section presents three phases of attestation protocol.

Preliminary Setup Phase. We assume a preliminary setup phase that distributes public keys (PK) and private keys (PR), as well as, registers user devices into the system. The trusted authority (which is the university IT department in our setup of TIPPERS) generates/renews/revokes keys used by the secure hardware enclave (denoted by $\langle PK_E, PR_E \rangle$) and the notifier (denoted by $\langle PK_N, PR_N \rangle$). The keys are provided to the enclave during the secure hardware registration process. Also, $\langle PK_{di}, PR_{di} \rangle$ denotes keys of the i^{th} user device. *Usages of keys*: The controller uses PK_E to encrypt sensor readings before sending to SP. PR_E is also used by the enclave to write encrypted sensor logs and decrypt sensor readings. PK_N is used during the notification phase by SGX to send an encrypted message to the notifier. User device's keys are used during device registration, as given below.

We assume a registration process during which a user identifies herself to the underlying system. For instance, in a WiFi network, users are identified by their mobile devices, and the registration process consists of users providing the MAC addresses of their devices (and other personally identifiable information, *e.g.*, email and a public key). During registration, users also specify their preferred modality through which the system can communicate with the user (*e.g.*, email and/or push messages to the user device). Such communication is used during the notification phase.[1]

5.1 Notification Phase

The notification phase informs data-capture rules established by SP to the (registered) users by explicitly sending *notice messages*.

[1]Figures 4 and 5 show additional methods that we used in UCI to inform about the data-capturing rules.

We consider two models for notification, differing based on acknowledgment from users.

In the *notice-only model (NoM)*, SP informs users of data-capture rules, but users may not acknowledge receipt of the message. Such a model is used to implement policies, when data capture is mandatory, and the user cannot exercise control, over data capture. Since there is no acknowledgment, SP is only required to ensure that it sends a notice, but is not required to guarantee that the user received the notice. In contrast, a *notice-and-ACK model (NaM)* is intended for discretionary data-capture rules that require explicit permission from users prior to data capture. Such rules may be associated, for instance, with fine-grained location services that require users' location. A user can choose not to let SP track his location, but will likely not be able to avail some services.

Implementation of notification differs based on the model used. Interestingly, since NaM requires acknowledgment, the notification phase is easier as compared to NoM that uses a trusted notifier to deliver the message to users. Below we discuss the implementation of both models:

Notification implementation in NoM. NoM assumes that, by default, data-capture rules are set not to retain any user data, unless SP, first, informs SGX about a data-capture rule, (*i.e.*, SP cannot use the encrypted sensor data for building any application, see ❾ in Figure 3). When SP creates a new data-capture rule, SP must inform SGX. Then, the enclave encrypts the data-capture rule using the public key (*i.e.*, PK_N) of the notifier and informs the trusted notifier (via SP) about the encrypted data-capture rule by writing it outside of the enclave (in our user-case §1, the university IT department works as a trusted notifier). Data-capture rules are maintained by SP on stable storage, which is read by SGX into the enclave to check, if the sensor data should be forwarded to SP. SGX can retain a cache of rules in the enclave, if such rules are still valid (and hence used for enforcement).[2] Finally, the trusted notifier acknowledges SP about receiving the encrypted data-capture rule, and then, informs users of the encrypted data-capture rule via signed notice messages. On

[2]Since the enclave has limited memory, the enclave cannot retain all the valid and non-valid data-capture rules after a certain size. Thus, the enclave writes all the non-valid data-capture rules on the disk after computing a secured hash digest over all the rules. Taking a hash over the rules is required to maintain integrity of all the rules, since any rule written outside of the enclave can be tampered by SP. Recall that altering a rule will make it impossible to verify partial/entire sensor data.

Figure 4: Notice messages placed on different buildings in UCI.

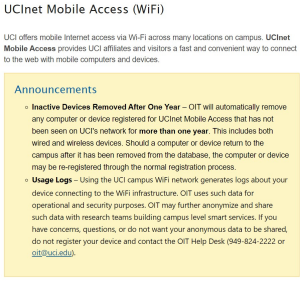

Figure 5: Notice messages at the university IT department website.

receiving the notice message, the users may decrypt it and obtain the data-capture rule.

To see the role of *trusted hardware* above, suppose that SP was responsible for informing users about data-capture rules directly. Data-capture rules are also required by SGX during log-sealing (PHASE 2). An adversarial SP may inform SGX, not to users, or may inform non-identical rules to users and to SGX. Hence, SP cannot inform the rule to users directly.

To see the role of the *trusted notifier* above, suppose that SP can directly inform users about encrypted data-capture rules obtained from SGX. An adversarial SP may not deliver the data-capture rule to all or some of the users; thus, an encrypted data-capture rule is not helpful. Hence, a trusted notifier ensures that the notice message is sent to all the registered users. Note that the trusted notifier might be a trusted web site that lists all the data capture rules which users can access.

Implementation of notification in NaM. Unlike NoM, the notification phase of NaM does not require the trusted notifier. In NaM, by default, SP cannot utilize all those sensor readings having device-ids for which the users have not acknowledged. Likewise NoM, in NaM, SP informs data-capture rules to SGX that encrypts the rule and writes outside of the enclave. The encrypted rules are delivered by SP to users, unlike NoM. On receiving the message, a user may securely acknowledge the enclave about her consent. The enclave retains all those device-ids that acknowledge the notice message for log-sealing phase and considers those device-ids during the log-sealing phase to retain their data while discarding data of others.

5.2 Log Sealing Phase

The second phase consists of cryptographically sealing the sensor data for future verification against pre-notified data-capture rules. The sensor data is sealed into secured logs using authenticated data structures, *e.g.*, hash-chains and XOR-linked lists (as shown in Figures 6, 7), by the sealing function, $Sealing(PR_E, \langle d_i, s_j, s_j.state, t_k \rangle)$, executed in the enclave at SP. Let us explain log-sealing in the context of WiFi connectivity data. The enclave reads the encrypted sensor data (❼ in Figure 3) and executes the three steps: (*i*) decrypts the data, (*ii*) checks the data against pre-notified valid data-capture rules, and (*iii*) cryptographically seals the data and store *appropriate secured logs*.

Below we explain our log sealing approach. To simplify the discussion, we first consider the case when all the sensor data satisfies some data-capture rule (*i.e.*, the state of all the sensor data is one), and hence, data is forwarded to and stored at SP §5.2.1. Then, we adapt the protocol to deal with the case when some sensor data satisfies some data-capture rule (*i.e.*, the state of some sensor data is one, and hence, data is forwarded to and stored at SP), while remaining sensor data does not satisfy any rule (*i.e.*, the state of the remaining sensor data is zero, and hence, data is forwarded to SP) 5.2.3.

5.2.1 *Sealing Entire Sensor Data*

Informally, the sealing function executes a hash function on each sensor reading (or value), whose output is used to create a chain of hash digests. At the end of the sensor readings/values, the sealing function generates an authenticated proof-of-integrity by mixing a computationally-hard secure string. For example, consider four values: v_1, v_2, v_3, and v_4. The sealing function works as follows:

Value	Hash output
v_1	$h_1 \leftarrow H(v_1 \| H(0))$
v_2	$h_2 \leftarrow H(v_2 \| h_1)$
v_3	$h_3 \leftarrow H(v_3 \| h_2)$
v_4	$h_4 \leftarrow H(v_4 \| h_3)$
Proof-of-integrity	$\langle sign_{PR_E}(SS \oplus h_4) \rangle$

In this example, all the values are hashed while including the hash digest of the previous value, except the first value. Note that only the first value is hashed with the hash digest of zero. In the end, the proof-of-integrity is prepared by signing XORed-value of the hash digest of the last value and a secret random string, SS. Note that the secret random string is generated for each chunk in a specific manner, which will be clear in the detailed description of the protocol; please see below.

The sealing operation consists of the following three phases: (*i*) chunk creation, (*ii*) hash-chain creation, and (*iii*) proof-of-integrity creation; described below.

PHASE 1: Chunk creation. The first phase of the sealing operation finds an appropriate size of a chunk (to speed up the attestation process). Note that the incoming encrypted sensor data may be large, and it may create problems during verification, due to increased communication between SP and the verifier. Also, the verifier needs to verify the entire data, which have been collected over a large period of time (*e.g.*, months/years). Further, creating cryptographic sealing over the entire sensor data may also degrade the performance of $Sealing()$ function, due to the limited size of SGX enclave. Thus, we first determine an appropriate chunk size, for each of which the sealing function is executed.

The chunk size depends on time epochs, the enclave size, the computational overhead of executing sealing on the chunk, and the communication overhead for providing the chunk to the verifier. A small chunk size reduces the communication overhead and maintains the log minimality property, thereby during the log verification phase, a verifier retrieves only the desired log chunks, instead of retrieving the entire sensor data. Consequently, SP stores many chunks.

PHASE 2: Hash-chain creation. Consider a chunk, $\mathcal{C}_x$, that can store at most n sensor readings, each of them of the format: $\langle d_i, s_j, t_k \rangle$. The sealing function checks each sensor reading against data-capture rules and adds sensor state to each reading, as: $\langle d_i, s_j, s_j.state, t_k \rangle$. Since in this section we assumed that all sensor data will be stored, the sensor state of each sensor reading is set to 1. The sealing function starts with the first sensor reading of the chunk $\mathcal{C}_x$, as follows:

First sensor reading. For the first sensor reading of the chunk, the sealing function computes a hash function on value zero, *i.e*, $H(0)$. Then, the sealing function mixes $H(0)$ with the remaining

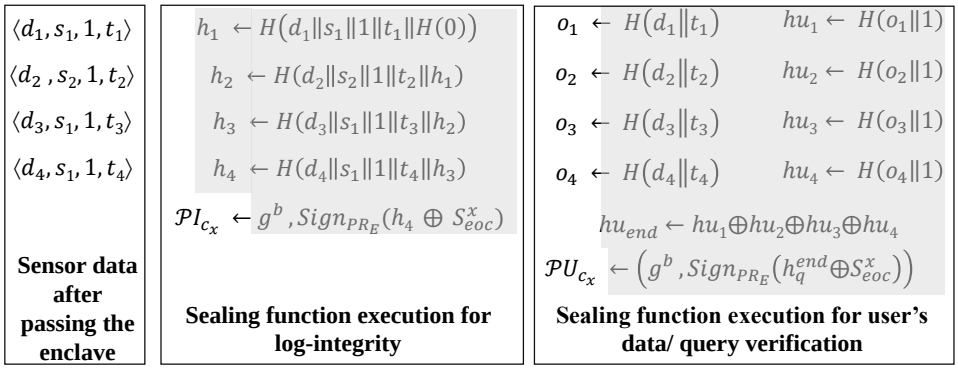

Figure 6: Cryptographically sealing procedure executed on a chunk, $\mathcal{C}_x$. Gray-shaded data is not stored on the disk. White-shaded data is stored on the disk and accessible by SP. Figure shows proof-of-integrity only for one chunk, $\mathcal{C}_x$; hence, some notations are abusively used. Note that we used h_i to denote a hash digest computed for verifying the entire sensor data, while hu_i denotes a hash digest computed for verifying the user data or query results.

values of the sensor reading, *i.e.*, sensor-id, device-id, sensor state, and time, at which it computes the hash function, denoted by $H(d_1||s_j||s_j.state||t_k||H(0))$ that results in a hash digest, denoted by h_1^x. After processing the complete first sensor reading of the chunk $\mathcal{C}_x$, the enclave writes cleartext first sensor reading of $\mathcal{C}_x$, *i.e.*, $\langle d_1, s_j, s_j.state, t_k\rangle$ on the disk, which can be accessed by SP.

Second sensor reading. Let $\langle d_2, s_j, s_j.state, t_{k+1}\rangle$ be the second sensor reading. For this, the sealing function works identically to the processing of the first sensor reading. It computes a hash function on the second sensor values, while mixing it with the hash digest of the first sensor reading, *i.e.*, $H(d_2||s_j||s_j.state||t_{k+1}||h_1^x)$ that results in a hash digest, say h_2^x. Finally, the enclave writes the second sensor reading in cleartext on the disk.

Processing the remaining sensor readings. Likewise, the second sensor reading processing, the sealing function computes the hash function on all the remaining sensor readings of the chunk $\mathcal{C}_x$. After processing the last sensor reading of the chunk $\mathcal{C}_x$, the hash digest h_n^x is obtained.

PHASE 3: Proof-of-integrity creation. Since each sensor reading is written on disk, SP can alter sensor readings, to make it impossible to verify log integrity by an auditor. Thus, to show that all the sensor readings are kept according to the pre-notified data-capture rules, the sealing function prepares an immutable proof-of-integrity for each chunk, as follows:

For each chunk $\mathcal{C}_i$, the sealing function generates a random string, denoted by g^j, where $i \neq j$. Let $\mathcal{C}_v$, $\mathcal{C}_x$, and $\mathcal{C}_y$ be three consecutive chunks (see Figure 7), based on consecutive sensor readings. Let g^a, g^b, and g^c be random strings for chunks $\mathcal{C}_v$, $\mathcal{C}_x$, and $\mathcal{C}_y$, respectively. The use of random strings will ensure that any of the consecutive chunks have not been deleted by SP (will be clear in §5.3). Now, for producing the proof-of-integrity for the chunk $\mathcal{C}_x$, the sealing function: (*i*) executes XOR operation on g^a, g^b, g^c, whose output is denoted by S_{eoc}^x, where eoc denotes the end-of-chunk; (*ii*) signs h_n^x XORed with S_{eoc}^x with the private key of the enclave; and (*iii*) writes the proof-of-integrity for log verification of the chunk $\mathcal{C}_x$ with the random string g^b, as follows:

$$\mathcal{PI}_{\mathcal{C}_x} = (g^b, Sign_{PR_E}(h_n^x \oplus S_{eoc}^x))$$

Note. We do not generate the proof for each sensor reading. The enclave writes only the proof and the random string for each chunk to the disk, which is accessible by SP. Further, the sensor readings having the state one are written on the disk, based on which SP develops services.

Example. Please see Figure 6, where the **middle box** shows PHASE 2 execution on four sensor readings. Note that the hash digest of each reading is passed to the next sensor reading on which a hash function is computed with the sensor reading. After computing h_4, the proof-of-integrity, $\mathcal{PI}$, is created that includes signed $h_4 \oplus S_{eoc}^x$ and a random string, g^b.

Note. g^* for the first chunk. The initialization of log sealing function requires an initial seed value, say g^*, due to the absence of 0^{th} chunk. Thus, in order to initialize the secure binding for the first chunk, the seed value is used as a substitute random string.

5.2.2 *Sealing Data for User Data/Service Verification*

While capturing *user-associated data*, users may wish to verify their user-associated data against notified messages. Note that *the protocol presented so far requires entire cleartext data to be sent to the verifier to attest log integrity* (it will be clear soon in §5.3). However, such cleartext data transmission is not possible in the case of user-associated data verification, since it may reveal other users' privacy. Thus, to allow verification of user-associated data (or service/query result[3] verification), we develop a new sealing method, consists of the three phases: (*i*) chunk creation, (*ii*) hash-generation, and (*iii*) proof-of-integrity creation. Chunk creation phase of this new sealing method is identical to the above-mentioned chunk creation phase 1; see §5.2.1. Below, we only describe PHASE 2 and PHASE 3.

PHASE 2: Hash-generation. Consider a chunk, $\mathcal{C}_x$, that can have at most n sensor readings, each of them of the format: $\langle d_i, s_j, s_j.state, t_k\rangle$. Our objective is to hide users' device-id and its frequency-count (*i.e.*, which device-id is prominent in the given chunk). Thus, on the i^{th} sensor reading, the sealing function mixes d_j with t_k, and then, computes a hash function over them, denoted by $H(d_j||t_k)$ that results in a digest value, say o_i. Note that hash on device-ids mixed with time results in two different digests for more than one occurrence of the same device-id. Note that o_i helps the user to know his presence/absence in the data during attestation, but it will not prove that tampering has not happened with the data. Then, the sealing function mixes o_i with the sensor state (to produce a proof of sensor state) of the i^{th} sensor reading, and on which it computes the hash function,

[3]The users, who access services developed by SP (as mentioned in §1), may also wish to verify the query results, since SP may tamper with the data to show the wrong results.

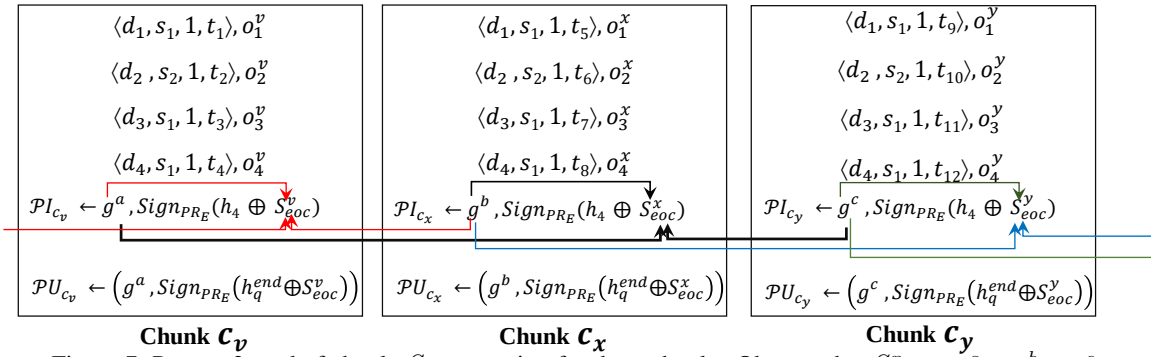

Figure 7: PHASE 3: end of chunk, S_{eoc}, creation for three chunks. Observe that $S_{eoc}^x = g^a \oplus g^b \oplus g^c$.

denoted by $H(o_i||s_j.state)$ that results in a hash digest, denoted by hu_i^x. After processing the i^{th} sensor reading of the chunk $\mathcal{C}_x$, the enclave writes o_i on the disk. After processing all the n sensor readings of the chunk $\mathcal{C}_x$, the sealing function computes XOR operation on all hash digests, hu_i^x, where $1 \leq i \leq n$: $hu_1^x \oplus hu_2^x \oplus \ldots \oplus hu_n^x$, whose output is denoted by hu_{end}^x. (Reason of computing hu_{end}^x will be clear in §5.3).

PHASE 3: Proof-of-integrity creation for the user. The sealing function prepares an immutable proof-of-integrity for users, denoted by $\mathcal{PU}$, for each chunk and writes on the disk. Likewise, proof-of-integrity for entire log verification, $\mathcal{PI}$ (§5.2.1), for each chunk, the sealing function obtains S_{eoc}; refer to PHASE 3 in §5.2.1. Now, for producing $\mathcal{PU}$ for the chunk $\mathcal{C}_x$, the sealing function: (*i*) signs hu_{end}^x XORed with S_{eoc}^x with the private key of the enclave, and (*ii*) writes the signed output with the random string of the chunk, g^b, as $\mathcal{PU}_{\mathcal{C}_x}$.

$$\mathcal{PU}_{\mathcal{C}_x} = (g^b, Sign_{PR_E}(hu_{end}^x \oplus S_{eoc}^x))$$

Note. The enclave writes hash digests, o_i for each sensor reading, the proof for user verification, and the random string for each chunk on the disk. Of course, the sensor readings having the state one are written on the disk.

Example. Please see Figure 6, where the **last box** shows PHASE 2 execution on four sensor readings to obtain the proof-of-integrity for the user, $\mathcal{PU}$.

5.2.3 *Sealing Mixed State Sensor Data*

The protocol so far has assumed that all data has the sensor state of one. We next consider how it can be generalized to a situation when some sensor readings may not satisfy the data capture rules (and hence, have the sensor state of zero). Recall that (as mentioned in §4.2), the enclave decrypts the sensor data received from the IFD and checks against the pre-notified data-capture rules, and if a sensor reading captured by a sensor s_i does not satisfy the data-capture rules, then the sensor state of the sensor reading becomes zero. Please note that the sensor state of zero does not indicate that the sensor is turned off.

Example 5.2.3.1. In a chunk, assume the following six sensor readings, produced by two sensors s_1 and s_2.

$$\langle d_1, s_1, 1, t_1\rangle$$
$$\langle d_2, s_2, 0, t_2\rangle$$
$$\langle d_2, s_2, 0, t_3\rangle$$
$$\langle d_3, s_2, 0, t_4\rangle$$
$$\langle d_3, s_2, 1, t_5\rangle$$
$$\langle d_1, s_1, 1, t_6\rangle$$

In this case, there is no need to store all sensor readings having state zero. However, doing it carelessly may provide an opportunity to the adversary to delete all the sensor readings and prove that such readings are deleted due to sensor state to be zero, due to not satisfying data-capturing rules. Thus, to avoid storing sensor readings having sensor state zero, we provide a method below.

Informally, the sealing function checks each sensor reading and deletes all those sensor readings for which sensor state is `passive`, due to not satisfying data-capture rules. In this case, it is not mandatory to seal each sensor readings, as mentioned in §5.2.1. Thus, the sealing function provides a *filter operation* that removes sensor readings whose device state is `passive`, while storing sensor readings whose device states are `active`. But, the sealing function cryptographically stores the data with minimal information of sensor readings whose device states are `passive`. Below we describe PHASE 2 for sealing mixed state sensor data. Note that PHASE 1 (chunk creation) and PHASE 3 (proof-of-integrity creation) are identical to the method described in §5.2.1.

PHASE 2: Sealing operation. Formally, to create hash-chain for this case, the sealing function will do the following: For the first i^{th} sensor reading (for example, $\langle d_j, s_k, 0, t_i\rangle$) whose $state = 0$, the enclave executes two operations: (*i*) sealing function that computes a hash function, as: $H(s_j||s_j.state||t_k||h_{i-1})$, whose output is denoted by h_i, and (*ii*) filter operation that deletes the device-id d_j and stores $\langle s_k, 0, t_i\rangle$ on the disk. Now for all the successive sensor readings until encountering a sensor reading, say l, with $state = 1$, all the sensor readings are discarded (not stored on the disk), as well as, the hash function is not executed. However, the sealing function computes the hash function on the l^{th} sensor reading ($\langle d_x, s_k, 1, t_l\rangle$), as: $H(d_x||s_j||1||t_l||h_i)$ and stores l^{th} sensor reading on the disk.

Example 5.2.3.2. Now, we apply the above-mentioned PHASE 2 on the sensor readings given in Example 5.2.3.1. Here, the enclave does not store the second, third, and fourth sensor readings on disk and delete them, while the remaining sensor readings will be stored on the disk. In addition, the enclave will store only the sensor device and its state of the first reading (*i.e.*, the second sensor reading) for which state was zero and generate verifiable logs, such that the verifier can verify that the three entries has been deleted due to not satisfying data-capturing rules. The sealing function computes the hash-chain and proof-of-integrity as follows:

$$h_1 \leftarrow H(d_1||s_1||1||t_1||H(0))$$
$$h_2 \leftarrow H(s_2||0||t_2||h_1)$$
$$h_3 \leftarrow H(d_3||s_2||1||t_5||h_2)$$
$$h_4 \leftarrow H(d_1||s_1||1||t_6||h_3)$$
$$\mathcal{PI} \leftarrow \langle secret\ string, sign_{PR_E}(secret\ string \oplus h_4)\rangle$$

Note that here we compute a hash function on the first, second,

fifth, and last sensor readings, while do not seal the third and fourth sensor readings, due to their passive (or zero) sensor states.[4]

5.2.4 Log-size Optimization

The above-mentioned procedure given in §5.2.3 regards the sensor states and stores less amount of cryptographically sealed data on the disk, by not storing consecutive sensor readings having sensor state of zero. However, such improvement is limited, if the sensor readings have state zero and one in an alternative sequence.

Example 5.2.4.1. Consider the following seven sensor readings obtained from two sensors s_1 and s_2. Here, the state of sensor s_2 is zero, due to not satisfying the data-capturing rules; however, the state of sensor s_1 is one.

$$\langle d_1, s_1, 1, t_1\rangle$$
$$\langle d_2, s_2, 0, t_2\rangle$$
$$\langle d_2, s_1, 1, t_3\rangle$$
$$\langle d_3, s_2, 0, t_4\rangle$$
$$\langle d_3, s_1, 1, t_5\rangle$$
$$\langle d_1, s_2, 0, t_6\rangle$$
$$\langle d_1, s_1, 1, t_7\rangle$$

In such scenario, the method given in §5.2.3 is not efficient, since it will store all the sensor readings and produce hash digest for each sensor reading, as follows:

$$h_1 \leftarrow H(d_1||s_1||1||t_1||H(0))$$
$$h_2 \leftarrow H(s_2||0||t_2||h_1)$$
$$h_3 \leftarrow H(d_2||s_1||1||t_3||h_2)$$
$$h_4 \leftarrow H(s_2||0||t_4||h_3)$$
$$h_5 \leftarrow H(d_3||s_1||1||t_5||h_4)$$
$$h_6 \leftarrow H(s_2||0||t_6||h_5)$$
$$h_7 \leftarrow H(d_1||s_1||1||t_7||h_6)$$

Thus, in order to reduce the size of secured log for the case when sensor readings have state zero and one in an alternative sequence, below, we propose a method that store logs for each sensor or each user-device.

Per sensor-based logging. We implement log-sealing procedure (given in §5.2.1 and §5.2.3) on per sensor, *i.e.*, we group the sensor readings produced by the same sensor before producing cryptographically sealed logs. This optimization method works as follows, for n sensor readings and x sensors:

1) The enclave creates x buffers, one buffer for each of the x sensors.
2) The enclave reads the n sensor readings from the disk and decrypts them.
3) Each of the n sensor readings is allocated to one of the buffers based on the sensor.
4) On each sensor reading allocated to a buffer, the enclave either executes the method given in §5.2.1 if all the sensor readings have state one, or executes the method given in §5.2.3 if the sensor readings have state one and zero.
5) At the end, the enclave writes cryptographically secured logs and proof-of-integrity for each buffer's sensor log, on the disk.

Example 5.2.4.2. Now, we show how one can reduce the size of secured logs for the seven sensor readings given in Example 5.2.4.1 by keeping logs per sensor based. Here, the enclave maintains two buffers: one for s_1 and another for s_2. All the sensor readings of s_1 are sealed using the method of §5.2.1, and the sensor readings of s_2 are sealed using the method of §5.2.3. Thus, the enclave executes the following computation:
The first buffer dealing with the sensor readings of sensor s_1:

$$h_1 \leftarrow H(d_1||s_1||1||t_1||H(0))$$
$$h_2 \leftarrow H(d_2||s_1||1||t_3||h_1)$$
$$h_3 \leftarrow H(d_3||s_1||1||t_5||h_2)$$
$$h_4 \leftarrow H(d_1||s_1||1||t_7||h_3)$$
$$\mathcal{PI}_{s_1} \leftarrow \langle secret\ string, sign_{PR_E}(secret\ string \oplus h_4)\rangle$$

The second buffer dealing with the sensor readings of sensor s_2:

$$h_5 \leftarrow H(s_2||0||t_2||H(0))$$
$$\mathcal{PI}_{s_2} \leftarrow \langle secret\ string, sign_{PR_E}(secret\ string \oplus h_5)\rangle$$

Note that, here, we used the notations $\mathcal{PI}_{s_1}$ and $\mathcal{PI}_{s_2}$ to indicate the proof-of-integrity generated for sensors s_1 and s_2, respectively. The enclave writes cleartext sensor readings corresponding to the sensor s_1, the tuple $\langle s_2, 0, t_2\rangle$, hash digests $h_1, h_2, \ldots h_5$, and proof-of-integrity for both sensors $\mathcal{PI}_{s_1}$ and $\mathcal{PI}_{s_2}$, on the disk.

Issues. The above-mentioned method, while reduces the size of cryptographically sealed logs, it faces two issues, as follows:

1) *More sensors.* It may happen that the number of sensors are significantly more, thereby it is not easy to maintain buffers for each sensor, due to a limited memory of the secure hardware. In this case, we can still use the above method; however, one buffer is responsible for more than one sensor. For example, if there are four sensors, $s_1, s_2, \ldots, s_4$, and the secure hardware can hold only two buffers, then the first buffer might be responsible for sensor readings corresponding to two sensors s_1 and s_2, and another buffer might be responsible for sensor readings corresponding to the remaining two sensors. To allocate sensor readings to buffers, one can use a hash function on the sensor-id to know the buffer id.
2) *Different-sized log.* Note that when we create multiple buffers for sensors, the enclave writes multiple chunks and proof-of-integrity corresponding to multiple buffers. The size of each chunk may not be identical, due to having a different number of hash digests. It may reveal information to the adversary that which sensor's state is zero. To avoid such leakage, the enclave may pad output produced for each buffer with some fake values, and may include this information in proof-of-integrity to show how many fake values are added. For instance, in Example 5.2.4.2, the output corresponding to the second buffer has only one hash digest (*i.e.*, h_5) while the output corresponding to the first buffer will have four hash digests (*i.e.*, h_1, h_2, h_3, h_4). Thus, to write the same size data for each buffer, the enclave may pad the output corresponding to the second buffer with three fake hash digests and write this information in the proof-of-integrity, as follows: $\mathcal{PI}_{s_2} \leftarrow \langle secret\ string, sign_{PR_E}(secret\ string \oplus h_5, 3)\rangle$. Note that while adding fake hash digests, the user does not need to verify any fake digest, and thus, verifying only desired data will reduce the verification time, as we will see in Experiment 7 in §6.

Note: Per user-based logging. We can also apply the same procedure for each user device-id to reduce the size of the cryptographically sealed data, by creating buffers for each user device id. This may reduce the verification time for user-related data.

[4] We can only compress $x > 1$ continuous sensor readings, say $\langle *_d, *_s, *_s.state = 0, *_t\rangle$ (where $*_d$, $*_s$, and $*_t$ denote any device-id, sensor-id, and time, respectively) to produce a proof that such x readings have been deleted. However, we cannot compress x sensor readings having $*_s.state = 1$, since it disallows to verify service integrity (*e.g.*, a user query, how many time a user has visited a location, cannot be verified, if x readings with $*_s.state = 1$ have been deleted).

5.3 Attestation Phase

The attestation phase contains two sub-phases: (*i*) key establishment between the verifier and service provider to retrieve logs (§5.3.1), and (*ii*) verification of the retrieved logs (§5.3.2).

5.3.1 *Key Establishment*

The secure log retrieval is crucial for proof validation and non-trivial in the presence of a de-centralized verifier model, where anyone can execute a remote request to attest the secured logs against the data-capture rules. Thus, before the log transmission from the service provider to the verifier (*i.e.*, a user or an auditor), for each attestation request, the verifier's identity must be authenticated prior to the session key establishment.

Our log retrieval scheme is based on Authenticated Key Exchange (AKE) protocol [7], where a verifier and the service provider (*i.e.*, prover) dynamically establish a session key (shown in Figure 8). This dynamic key establishment provides forward secrecy for all future sessions, such that, for any compromised session in future, all sessions in the past remain secure. To achieve these properties, we use SIGMA [7] protocol, which is the cryptographic base for Internet Key Exchange (IKE) protocol. The family of SIGMA protocols is based on Diffie-Hellman key exchange [29]. We only show a 3-round version of SIGMA, as it provides verifier/sender-identity protection during the key establishment process. Recall that in our solution, the prover is a centralized service provider, but the verifier can be anyone, and therefore, we use this identity protection method to achieve verifier's identity privacy during the session.

Without the loss of generality, we, first, define the Computational Diffie Hellman (CDH) [29]. During the session, the verifier and the prover must execute the computations within a cyclic group $G = \langle g \rangle$ that has a generator g of a prime order q. According to the CDH assumption, the computation of a discrete logarithm function on public values (g, g^x, g^y) is hard within the cyclic group G. In particular, given the publicly known value g, it is hard to distinguish g^{xy} from g^x and g^y without knowing the ephemeral secrets x and y.

Figure 8 depicts SIGMA-based communication flow between a verifier and the service provider/prover. Initially, a verifier selects a secret value x, computes g^x as a public exponent, and sends it to the prover. Similarly, the prover selects a secret value y, computes g^y, and receives the public exponent g^x from the verifier. Next, the prover computes a joint secret value as: $e = g^{xy}$, and also uses it to derive a message authentication code (MAC) key MAC_k.

The prover composes a message structure as: $[g^y, SP_{id}, MAC_k(SP_{id}||g^x||g^y)]$, where g^y is the prover's public exponent for session key generation, SP_{id} is the identity of sender/prover, and $MAC_k(SP_{id}||g^x||g^y)$ is the message authentication code generated on sender's identity and all public exponents used for session key derivation. Note that in order to generate this MAC, a separate key MAC_k is derived from the session key e.

Next, the verifier receives this message and retrieves the public exponent g^y to generate a local copy of session key e, as well as, the message authentication code generating key MAC_k derived from e. At this stage, both parties have locally computed a secure session key e; however, there is still an identity disclosure required from the verifier to prove that the freshly generated key e binds to an authentic identity holder v_{id}, where v_{id} is the verifier's identity.

Thus, the verifier sends a message, consisting of $[v_{id}, MAC_k(v_{id}||SP_{id}||g^x||g^y), \langle log \quad query \rangle_e]$, where

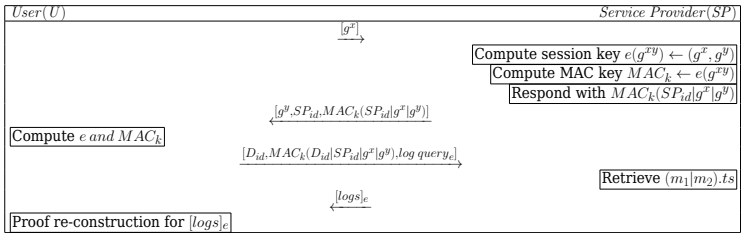

Figure 8: Secure log retrieval by the verifier.

$MAC_k(v_{id}||SP_{id}||g^x||g^y)$ is the message authentication code on verifier's identity v_{id}, responder's identity SP_{id}, and all public exponents exchanged so far, and $\langle log\ query \rangle_e$ is the log query request protected by the freshly generated session key e. This step binds all public key exponents exchanged with the claimed identity holders together and marks the end of the authenticated session key exchange process. The service provider, then, retrieves the corresponding logs according to the log query request in the last message and sends the log encrypted using the session key e to the verifier.

5.3.2 *Verification of Logs*

This section presents procedures for log verification at the auditor and a user.

Verification process at the auditor. Recall that the auditor can verify any part of the sensor data. Suppose the auditor wishes to verify a chunk $\mathcal{C}_x$; see Figure 7. Hence, entire sensor data (the data written in first box of Figure 6) of the chunk $\mathcal{C}_x$, random strings g^a, g^b, and g^c (corresponding to the previous and next chunks of $\mathcal{C}_x$; see Figure 7), and proof-of-integrity $\mathcal{PI}_{\mathcal{C}_x}$ are provided to the auditor. The auditor performs the same operation as in PHASE 2 of §5.2.2. Also, the auditor computes the end-of-chunk string $S^x_{eoc} = g^a \oplus g^b \oplus g^c$. At the end, the auditor matches the results of $h^x_n \oplus S^x_{eoc}$ against the decrypted value of received $\mathcal{PI}_{\mathcal{C}_x}$, and if both the values are identical, then it shows that the entire chunk is unchanged.

Note that since SP transfers sensor readings of the chunk $\mathcal{C}_x$, random strings (g^a, g^b, and g^c) and $\mathcal{PI}_{\mathcal{C}_x}$ to the user, SP can alter any transmitted data. However, SP cannot alter the signed $Sign_{PR_E}(h^x_n \oplus S^x_{eoc})$, due to unavailability of the private key of the enclave, PR_E, which was generated and provided by the trusted authority to the enclave. Thus, by following the above-mentioned procedure on the sensor readings of $\mathcal{C}_x$, any inconsistency created by SP will be detected by the auditor.

Verification process at the user. If the user wishes to verify his data in a chunk, say $\mathcal{C}_x$, the user is provided all hash digests computed over device-id and time (o_i, see the last box in Figure 6), time, sensor state, random strings g^a, g^b, and g^c (see Figure 7), and the proof $\mathcal{PU}$ by SP. Since, the user knows her device-id, first, the user verifies her occurrences in the data by computing the hash function on her device-id mixed with provided time values and compares against received hash digests. This confirms the user's presence/absence in the data. Also, to verify that no hash-digest is modified/deleted by SP, the user computes the hash function on the sensor state mixed with the received o_i ($1 \leq i \leq n$, where n in the number of sensor readings in $\mathcal{C}_x$) and computes $hu^x_{end} = h^x_1 \oplus h^x_2 \oplus \ldots \oplus h^x_n$. Finally, the user computes $hu^x_{end} \oplus S^x_{eoc}$ and compares against the decrypted value of $\mathcal{PU}$. The correctness of this method can be argued in a similar manner to the correctness of the verification at the auditor.

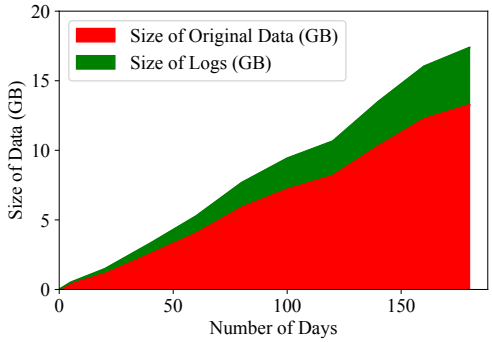

Figure 9: Exp 1: Storage overhead.

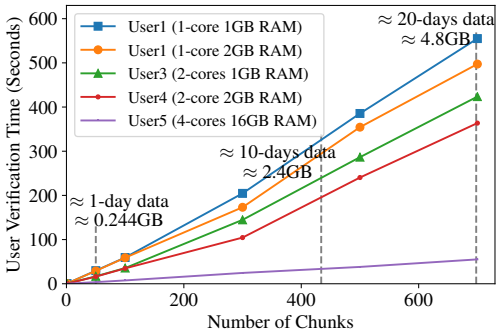

Figure 10: Exp 4: Verification time.

6 Experimental Evaluation

This section presents our experimental results on live WiFi data. We execute IoT Notary on a 4-core 16GB RAM machine equipped with SGX at Microsoft Azure cloud.

Setup. In our setup, the IT department at UCI is the trusted infrastructure deployer. It also plays the role of the trusted notifier (notifying users over emailing lists). At UCI, 490 WiFi sensors, installed over 30 buildings, send data to a controller that forwards data to the cloud server, where IoT Notary is installed. The cloud keeps cryptographic log digests that are transmitted to the verifier, while sensor data, qualifies data-capture rules, is ingested into realtime applications supported by TIPPERS. We use SHA-256 as the hashing algorithm and 256-bit length random strings in IoT Notary. We allow users to verify the data collected over the last 30minutes (min).

Dataset size. Although IoT Notary deals with live WiFi data, we report results for data processed by the system over 180 days during which time IoT Notary processed 13GB of WiFi data having 110 million WiFi events.

Data-capture rules. We set the following four data-capture rules: (*i*) *Time-based*: always retain data, except from t_i to t_j each day; (*ii*) *User-location-based*: do not store data about specified devices if they are in a specific building; (*iii*) *User-time-based*: do not capture data having a specific device-id from t_x to t_y ($x \neq i$, $y \neq j$) each day; and (*iv*) *Time-location-based*: do not store any data from a specific building from time t_x to t_y each day. The validity of these rules was 40 days. After each 40-days, variables i, j, x, y were changed.

Exp 1. Storage overhead at the cloud. We fix the size of each chunk to 5MB, and on average, each of them contains $\approx$ 37K sensor readings, covering around 30min data of 30 buildings in peak hours. Based on 5MB chunk size, we got 3291 chunks for 180 days. For each chunk, the sealing function generates two types of logs: (*i*) for auditor verification that produced proof-of-integrity $\mathcal{PI}$ of size 512bytes, and (*ii*) for user verification that produces hashed values (see Figure 6) and proof-of-integrity for users $\mathcal{PU}$ of size 1.05MB. Figure 9 shows 180-days WiFi data size without having sealed logs (red color) and with sealed logs (green color).[5]

Exp 2. Performance at the cloud. For each 5MB chunk, the sealing function took around 310ms to seal each chunk. This includes time to compute $\mathcal{PI}$, $\mathcal{PU}$ and encrypt them.

Exp 3. Auditor verification time. The auditor at our campus has a 7th-Gen quad-core i7CPU and 16GB RAM machine. It downloads the chunks from the cloud and executes auditor verification. The auditor varied the number of chunks from 1 to 3000; see Table 1. Note that to attest one-day data across 30 buildings, the auditor needs to download at most 50 chunks, which took less than 1min to verify. Observe that as the number of chunks increases, the time also increases, due to executing the hash function on more data.

Number of Chunks	1	50	100	500	1000	3000
$\approx$ duration (day)	30-60min	1-2	2-5	8-18	35-55	175
Verification time (seconds)	1	49	102	544	1160	4400

Table 1: The auditor verification time. Duration varies due to different class schedules in buildings and working hours.

Exp 4: Verification at a resource-constrained user. To show the practicality of IoT Notary for resource-constrained users, we considered four types of users, differing on computational capabilities (*e.g.*, available main memory (1GB/2GB) and the number of cores (1 or 2 cores)). Each user verified 1/10/20-days data; see Figure 10. Note that verifying 1-day data, which is $\approx$ 50 blocks, at resource-constrained users took at most 30s. As the number of blocks increases, the computational time also increases, where the maximum computational time to verify 20-days data was $<$ 10min. As the days increase, so does data transmitted to the user, which spills over to disk causing an increased latency. Also, we execute the same experiment on a powerful user having 4-core and 16GB machine. Note that as the number of core and memory increase, it results in parallel processing and absence of disk data read. Thus, the computation time decreases (see user 5 in Figure 10).

Chunk Size (KB)	5	100	1,000	5,000	10,000	25,000	40,000
Time (millisecond)	30	38	120	246	579	1,789	3,389

Table 2: Exp 5: Impact of the chunk size on log sealing execution.

Exp 5. Impact of the chunk size. We have selected chunk size to be 5MB that can hold at most 30min WiFi data. Now, we investigate the impact of chunk size on the sealing function execution time, the verification time, and latency in obtaining the most recent data. Table 2 shows that as the chunk size increases, the chunk holds more data, and hence, executing sealing function on a large-sized data takes more time. Similarly, a large-sized chunk also increases the latency in obtaining the most recent data (see Figure 11), since unless filling the chunk, the enclave cannot produce the proof-of-integrity. We can also create a chunk having only a single row; however, it will increase the size of secured logs (see Figure 11). Observe that (in Figure 11), when the chunk size is 5KB, the secured log size is $\approx$ 7GB, while when the chunk size is 5MB, the secured log size is $\approx$ 4GB. Figure 12 shows verification time also increases as the chunk size increases.

[5]The reason of getting more chunks is that during non-peak hours 5MB chunk can store sensor readings for more than one hour. However, as per our assumption, we allow the user to verify the data collected over the last 30min. Hence, regardless of the chunk is full or not, we compute the sealing function on each chunk after 30min.

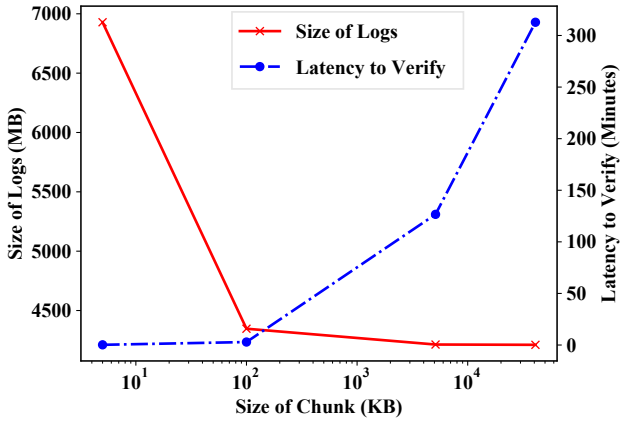

Figure 11: Exp 5: A tradeoff between log size and latency.

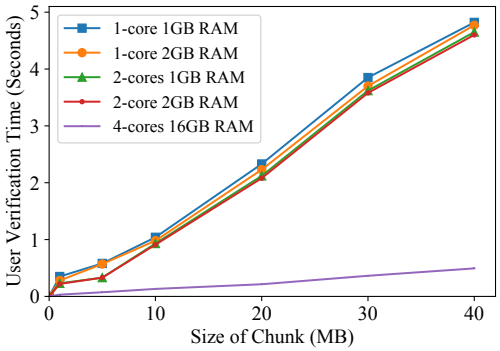

Figure 12: Exp 5: User verification time with different log chunk sizes.

Exp 6: Impact of communication. We measured the communication impact when a verifier downloaded the sensor data and/or sealed log for attestation. Consider a case when the verifier attests only one-hour/one-day data. The average size of one-hour (one-day) data in a peak hour was 14MB (250MB) having 103K (1.2M) connectivity events, while in a non-peak hour, it was 2MB (50MB) having 13.5K (320K) connectivity events. When using slow (100MB/s), medium (500MB/s), and fast (1GB/s) speed of data transmission, the data transmission time in case of 1-hour/1-day data was negligible.

Exp 7: Impact of parallelism. The processing time at each server can be reduced by parallelizing the computation. We investigated the impact of parallel processing to seal a 5MB chunk when having the number of threads 2 or 4 that took 322ms and 310ms, respectively. Increasing more threads did not provide speed-up, since the execution time increases due to thread maintenance overheads. Note that we only parallelized the hash function computation for $\mathcal{PU}$, (while $\mathcal{PI}$ cannot be computed in parallel, due to the formation of hash chains).

Exp 8. Impact of log optimization. We compare the impact of per sensor- and per-user-device-id-based optimization methods (denoted by OPT-SENSOR and OPT-USER in Figure 12, given in §5.2.4) against the non-optimized method (denoted by NON-OPT in Figure 12, given in §5.2.1). Since there were 490 WiFi access points in our experiments, we pre-allocate 490 buffers in SGX memory for 490 sensors, one buffer for each sensor. As a result, each buffer size was ≈ 85KB. We also created buffers for groups of devices to implement OPT-USER. Here, we again created 490 buffers and an identical group of user devices is allocated to a buffer. Particularly, the enclave extracted the user device's MAC

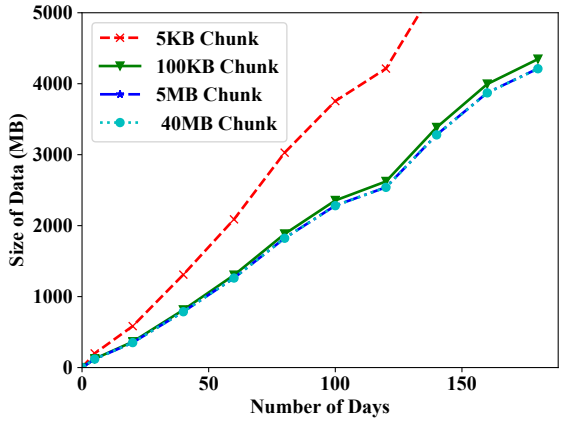

Figure 13: Exp 8: Size of the secured log when using different log sealing methods

address, hashed to get the buffer identity, and placed the sensor reading to corresponding buffer. When the buffer got full, the data in the buffer is cryptographically sealed and written on the disk. Here, we compare storage overhead and verification time.

Figure 12 shows the storage overhead to store only cryptographically-secured logs, when using OPT-SENSOR, OPT-USER, and NON-OPT methods. Since OPT-SENSOR and OPT-USER methods produce chunks of size 85KB, for a fair comparison, we set the log chunk size to be 85KB in NON-OPT method. Figure 12 shows that OPT-SENSOR and OPT-USER save ≈6.5% and ≈2.1% space, respectively, compared to NON-OPT method.

We also investigated the benefit in the performance improvement of user data verification, when using OPT-SENSOR, OPT-USER, and NON-OPT methods. Note that when using OPT-SENSOR and NON-OPT methods, a chunk may store data associated with other users. Thus, it needs to verify additional data, which does not belong to the user. In contrast, when using OPT-USER methods, a user has to verify only the desired data that belongs to him/her. It is clear that OPT-USER method requires to verify less amount of data, and hence, less verification time, compared to OPT-SENSOR and NON-OPT methods. Table 3 shows verification time and number of chunks required to verify one-day data at a resource-constrained user (1-core 1GB RAM). Observe that OPT-USER method takes significantly less time compared to NON-OPT method.

Method	# chunks	Verification time
NON-OPT	3012	31.2s
OPT-SENSOR	86	0.89s
OPT-USER	57	0.71s

Table 3: Exp 8: User verification performance when using different methods.

7 CONCLUSION

This paper presented a framework, IOT NOTARY for sensor data attestation that embodies cryptographically enforced log-sealing mechanisms to produce immutable proofs, used for log verification. Our solution improves the naïve end-to-end encryption model, where retroactive verification is not provable. The service verification mechanism on failing at users allows users to revoke services of the concerned IoT space. Therefore, a user is not required to blindly trust in the IoT space, and we empower

the users with the right-to-audit instead of right-to-own the data captured by sensors. IoT NOTARY is a part of a real IoT system (TIPPERS) and provides verification on live WiFi data with almost no overheads on users.

In addition, we are exploring the following improvement to the current implementation of IoT NOTARY: *Supporting data-capturing rules where conditions depend on the value of other sensors*: The current implementation does not support context-dependent rules, where the context is determined based on data captured by other sensors. Extending the system to handle such data-capture rules, *e.g.*, "Do not capture my WiFi connectivity data if I am the only person connected to the access point," is a non-trivial challenge.

REFERENCES

[1] N. Panwar *et al.*, "IoT Notary: Sensor data attestation in smart environment," *CoRR*, vol. abs/1908.10033, 2019. [Online]. Available: http://arxiv.org/abs/1908.10033

[2] S. Mehrotra *et al.*, "TIPPERS: A privacy cognizant IoT environment," in *PerCom Workshops*, 2016, pp. 1–6, http://tippersweb.ics.uci.edu/.

[3] E. Fernandes *et al.*, "Security implications of permission models in smart-home application frameworks," *IEEE Security & Privacy*, vol. 15, no. 2, pp. 24–30, 2017.

[4] A. Rao *et al.*, "Expecting the unexpected: Understanding mismatched privacy expectations online," in *SOUPS*, 2016, pp. 77–96.

[5] S. Madakam *et al.*, "Security mechanisms for connectivity of smart devices in the internet of things," 2016.

[6] S. K. Aikins, "Connectivity of smart devices: Addressing the security challenges of the internet of things," in *Connectivity Frameworks for Smart Devices: The Internet of Things from a Distributed Computing Perspective*, 2016.

[7] H. Krawczyk, "Sigma: The 'SIGn-and-MAc' approach to authenticated diffie-hellman and its use in the IKE protocols," in *CRYPTO*, 2003.

[8] A. Ibrahim *et al.*, "AID: autonomous attestation of IoT devices," in *SRDS*, 2018.

[9] N. Asokan *et al.*, "Seda: Scalable embedded device attestation," in *CCS*, 2015, pp. 964–975.

[10] A. Ibrahim *et al.*, "Darpa: Device attestation resilient to physical attacks," in *WiSec*, 2016, pp. 171–182.

[11] M. Ambrosin *et al.*, "SANA: secure and scalable aggregate network attestation," in *CCS*, 2016, pp. 731–742.

[12] X. Carpent *et al.*, "Remote attestation of iot devices via smarm: Shuffled measurements against roving malware," in *HOST*, 2018, pp. 9–16.

[13] M. Conti *et al.*, "Radis: Remote attestation of distributed iot services," in *Sixth International Conference on Software Defined Systems (SDS)*, 2019, pp. 25–32.

[14] J. Wang *et al.*, "Enabling security-enhanced attestation with Intel SGX for remote terminal and iot," *TCDICS*, vol. 37, no. 1, pp. 88–96, 2018.

[15] D. C. G. Valadares *et al.*, "Achieving data dissemination with security using FIWARE and Intel software guard extensions," in *ISCC*, 2018.

[16] J. Frankle *et al.*, "Practical accountability of secret processes," in *USENIX*, 2018, pp. 657–674.

[17] S. Eskandarian *et al.*, "Certificate transparency with privacy," *PoPETs*, vol. 2017, no. 4, pp. 329–344, 2017.

[18] W. Jiang *et al.*, "Transforming semi-honest protocols to ensure accountability," *Data Knowl. Eng.*, vol. 65, no. 1, pp. 57–74, 2008.

[19] B. R. Waters, D. Balfanz, G. Durfee, and D. K. Smetters, "Building an encrypted and searchable audit log." in *NDSS*, vol. 4, 2004, pp. 5–6.

[20] S. A. Crosby *et al.*, "Efficient data structures for tamper-evident logging." in *USENIX*, 2009, pp. 317–334.

[21] S. Zawoad *et al.*, "Towards building forensics enabled cloud through secure logging-as-a-service," *IEEE TDSC*, vol. 13, pp. 148–162, 2016.

[22] E. Boyle *et al.*, "Function secret sharing," in *EUROCRYPT*, 2015.

[23] Y. Zhang *et al.*, "vSQL: Verifying arbitrary SQL queries over dynamic outsourced databases," in *IEEE SP*, 2017, pp. 863–880.

[24] V. Costan *et al.*, "Intel SGX explained," *IACR Cryptology ePrint Archive*, vol. 2016, p. 86, 2016.

[25] https://newsroom.intel.com/newsroom/wp-content/uploads/sites/11/2017/09/8th-gen-intel-core-product-brief.pdf.

[26] W. Wang *et al.*, "Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX," in *CCS*, 2017, pp. 2421–2434.

[27] L. Lamport, "Password authentication with insecure communication," *Commun. ACM*, vol. 24, no. 11, pp. 770–772, 1981.

[28] D. M'Raihi *et al.*, "TOTP: Time-based one-time password algorithm," in *RFC 6238 Internet Engineering Task Force*. [Online]. Available: https://doi.org/10.17487/RFC6238

[29] W. Diffie *et al.*, "New directions in cryptography," *IEEE Trans. Information Theory*, vol. 22, no. 6, pp. 644–654, 1976.

APPENDIX A
FUTURE PASSWORD-BASED NOTICE MESSAGES

In §5.1, we described two procedures for notifying users about data-capturing rules. Now, we describe a time-based notification method. The authorization and validation of data-capture rule declaration is based on Time-based One-Time Passwords (TOTP) [28] such that the users are pre-initialized to receive the notifications regarding any sensitive data collection until an upcoming reference time t_{max} happens in the future. All notifications from the time of initialization t_{init} until the future reference time are chained through exactly in the timeline sequence rounded over the equally distant interval of an epoch size $\mathcal{E}$. Once the chain of these pre-initialized notifications nears the end, *i.e.*, the current time t_{cur} is same as the future reference time t_{max} (as declared in the beginning); the service provider initializes a new chain for the next set of notifications until a new reference time t'_{max} in future such that

$$t'_{max} > t_{init} > t_{init}$$

Notification phase: The service provider selects an upcoming reference time t_{max} for which all registered users will be notified. The service provider computes a local secret passphrase such as x and compute $y_{init} = \mathcal{H}^{length}(x)$ where $length$ is the length of the chain as $t_{max} - t_{init}/\mathcal{E}$ and $\mathcal{H}^{length}$ is the $length$ successive iterations of hash function $\mathcal{H}$ over the value x. The registered users receive a tuple $\langle y_{init}, t_{init}, t_{max}\rangle$ to compute the hash-chain at each interval during $t_{max} - t_{init}$. Let us assume that the service provider dispatch a notification during ith epoch $\mathcal{E}_i$ in the hash-chain. The service provider computes $y_i = \mathcal{H}^{length-i}(x)$ and send it to all registered users along with the message contents and a HMAC over passphrase and the message contents.

$$\langle y_i, msg, HMAC(y_i, msg)\rangle$$

Subsequently, all registered users retrieve the previously validated y_{i-1} (which is y_{init} at the beginning of the hash-chain) and use it to validate the current passphrase y_i. Note that the users must compute $\mathcal{H}(y_i)$ and compare it to previously validated passphrase y_{i-1}. If $\mathcal{H}(y_i) = y_{i-1}$ the user replace previously validated passphrase y_{i-1} with the currently validated passphrase y_i and use it for next passphrase validation. It must be noted that x represent the passphrase at the end of the hash-chain where $t_{cur} = t_{max}$.

It must be noted that the usage of same hash function for the passphrase computation is vulnerable to birthday attacks. In addition the passphrase (hence the notification) remain valid and irrevocable for an indefinite duration which makes it vulnerable to leakage attacks. Therefore, to allow ephemeral passphrases and the notifications that are exposed for a short duration only a time-based counter is required. In particular, a primary hash function and a time-based counter can be used here together to derive as many separate hash functions as the length of the hash-chain $length$. In that case the initial passphrase would be:

$$y_{init} = \mathcal{H}_{length}(\mathcal{H}_{length-1}(\ldots(\mathcal{H}_1(x))\ldots))$$

Also, for any notification validation during $t \in (t_{init}, t_{max}]$ the service provider must yield $\langle y_t, msg, HMAC(y_t, msg)\rangle$ such that

$$y_t = \mathcal{H}_{t_{max}-t}(\mathcal{H}_{t_{max}-t-1}(\ldots(\mathcal{H}_1(x))\ldots))$$

In order to validate the passphrase and the notifications at time $t_{cur} > t_{prev}$, the registered users must compute the hash-chain from $(t_{max} - t_{prev})$ to $(t_{max} - t_{cur} + 1)$.