Senopra: Reconciling Data Privacy and Utility via Attested Smart Contract Execution

DAT LE TIEN, University of Oslo
FRANK ELIASSEN, University of Oslo

ABSTRACT

The abundance of smart devices and sensors has given rise to an unprecedented large-scale data collection. While this benefits various data-driven application domains, it raises numerous security and privacy concerns. In particular, recent high-profile data breach incidents demonstrate security dangers and single point vulnerability of multiple systems. Moreover, even if the data is properly protected at rest (i.e., during storage), data confidentiality may still be compromised once it is fed as input to computations. In this paper, we introduce Senopra, a privacy-preserving data management framework that leverages trusted execution environment and confidentiality-preserving smart contract system to empower data owners with absolute control over their data. More specifically, the data owners can specify fine-grained access policies governing how their captured data is accessed. The access policies are then enforced by a policy agent that operates in an autonomous and confidentiality-preserving manner. To attain scalability and efficiency, Senopra exploits Key Aggregation Cryptosystem (KAC) for key management, and incorporates an optimisation that significantly improves KAC's key reconstruction cost. Our experimental study shows that Senopra can support privacy-preserving data management at scale with low latency.

1 INTRODUCTION

The emerging ubiquity of "smart devices", which are essentially electronic devices that can capture, exchange and process data so as to operate autonomously and interactively, has enabled unprecedented large-scale data collection. These devices can either be installed in typical living and working spaces (e.g., automatic light switch, voice-assisted electronic appliances such as TV or microwaves), or be worn and used on a daily basis by individuals (e.g., mobile phones, smart watches, etc) enables large-scale data collection. The resulting datasets are typically stored on a cloud network, and utilised for various purposes including service customization [29, 36], emergency notification systems, or training of complex machine learning models that facilitate a host of Internet services such as image and speech recognition or products and services recommendation. In all of these scenarios, activities of or data generated by individual users – their voice commands, online and offline transactions, photo, locations, preferences, health data – are exploited to enhance some utilities.

Nonetheless, this de facto standard of data collection, storage and processing raises various security and privacy concerns. On the one hand, a number of recent high-profile data breach incidents [5, 18] demonstrate the security dangers and single point vulnerability of multiple systems. These data breaches have serious privacy implications. For example, given an individual's social identification number and race, an adversary can perpetrate a large array of attacks such as physical stalking, identify theft, or inferring the individual's political and religious view. On the other hand, even if the data are properly protected at rest (i.e., during storage), there exists numerous attacks that compromise data confidentiality once it is fed as input to a computation [53, 54]. In the wake of these concerns, we argue that it is of

Authors' addresses: Dat Le Tien, University of Oslo, dattl@ifi.uio.no; Frank Eliassen, University of Oslo, frank@ifi.uio.no.

utmost importance to protect the confidentiality and privacy of individuals' data that is captured by or generated via their usage and interaction with the smart devices. Access rights for such data could be made possible for data mining or other analysis tasks, but they must be strictly adhere to the access policy specified by the data owners.

Active interest in protecting individuals' data privacy has motivated a number of works, ranging from regulations and governance legislation [4, 58] to academic research [21, 34, 61], to tackle this issue. Nonetheless, there remain challenges in adopting these solutions. Regulations and governance legislation are often implemented by centralized parties, whose trustworthiness is questionable at times (e.g., Facebook data scandal [5]). Technical approaches to enforce such regulatory measures are lacking in general. While security research provides techniques to tackle privacy concerns focused on personal data, implementing them is not trivial. In particular, one may attempt to conceal personally identifiable or sensitive data using approaches such as *k-anonymity* [21], *l-diversity* [43] or *t-closeness* [41]. Nonetheless, it has been demonstrated that anonymized datasets employing these techniques can be de-anonymized [48]. Other privacy-preserving techniques include differential privacy [34] and fully homomorphic encryption (FHE) [35, 57]. The former attains privacy by perturbing data or adding noise to the computations, while the latter enables computations directly on encrypted data, albeit at a prohibitive computational overhead.

In light of the aforementioned needs of protections for data privacy, and challenges that existing techniques are struggling to address, we propose a practical data management framework, called Senopra, that reconciles privacy and utility. Senopra gives data owners complete control over their personal or sensitive data. More specifically, it allows the data owners to specify a *fine-grained access policy* governing how the captured data are accessed. An access policy dictates (i) which data consuming services can access the data, and (ii) the permissible functions that the said services could perform over the data. Of equal importance is the confidentiality of the access policy and the key materials required to grant access to the protected data. Senopra keeps the policy internals (e.g., the white-list of permitted services) private, exposing it only as a black box API. To avoid a single point of failure and trust issues of centralised party, Senopra implements the agent that enforces the access control policy as a confidential smart contract (elaborated in Section 2.2) in a decentralized, confidentiality-preserving smart contract platform, called Oasis [12, 25].

Abstractly, users' data, once captured, are encrypted (under a symmetric-key encryption scheme, such as AES) using user-specific encryption/decryption key and maintained in a key-value store residing outside of the smart contract blockchain (i.e., off-chain). The pointer to the data and the key materials required to decrypt the data are retained by an on-chain confidentiality-preserving smart contract, which we shall refer to as `PolicyContract`, in its private state. A data consuming service that requests access to the protected data must run inside a trusted execution environment (TEE). The service has to attest itself to the `PolicyContract`. This attestation covers the functions to be executed on the data, and the constraint that the key materials shall reside exclusively inside the TEE, and be discarded after the execution finishes. If the attestation is successful, and the requesting service is eligible to access the data, `PolicyContract` communicates the pointer and the decryption key for the requested data to the requesting service's TEE.

To attain scalability, Senopra needs to handle the key management efficiently at scale. Since users' data is encrypted with user-specific key, the number of keys to be managed grows proportionally with the number of users participating in the system. Moreover, an user may split her data into different categories, each of which is to be protected by a unique key, further expanding the key pool. Recall that `PolicyContract` has to keep all key materials on-chain, maintaining a large key pool is challenging, for on-chain storage is typically expensive. Besides, a data consuming service may request for data from multiple users at the same time, incurring transmission of a large number of keys at once. Senopra overcomes these challenges by building on Key Aggregation Cryptosystem (KAC) [26] – a public key cryptosystem that can aggregate any set of decryption keys into a constant size decryption key. With KAC incorporated, `PolicyContract` has to maintain only one master-secret key, and transmit to the requesting service a single "aggregated" key, regardless of the number of users participating in the systems, and the diversity of the data that requested by the service. Finally, we introduce an algorithmic enhancement that enables the data consuming services to reconstruct the original set of decryption keys from the aggregated key efficiently.

We implement a prototype of SENOPRA, deploy `PolicyContract` on the Oasis Devnet [12], and evaluate its performance. Our experiments show that `PolicyContract`'s end-to-end latency in responding to a data consuming service's request is in seconds, sufficiently responsive to support real-time operations. The experimental study also shows that by incorporating a fast reconstruction technique, SENOPRA attains upto 56× speed-up compared to naive implementation of the original KAC's reconstruction.

In summary, we make the following contributions:

- We present a privacy-preserving data management framework named SENOPRAthat empowers data owners with complete control over their data. Access control management is implemented via a confidentiality-preserving smart contract, thereby avoiding single point of failure and unnecessary trust assumptions.
- We incorporate in SENOPRA an efficient and scalable key management system, allowing the system to operate at scale.
- We empirically evaluate the performance of SENOPRA, and show that our design can support scalable privacy-preserving data management, attaining near real-time latency.

The rest of the paper is structured as follows. Section 2 provides preliminaries on trusted execution environment, confidentiality-preserving smart contracts, and Key Aggregation Cryptosystem. Subsequently, Section 3 state our problem definition, describing challenges we face in designing SENOPRA. We then present SENOPRA in Section 4, and arguing about its security in Section 5. Section 6 reports experimental results on a working prototype of SENOPRA, while Section 7 reviews the related works, before Section 8 concludes our work.

2 PRELIMINARIES

This section give a brief overview on primitives that we adopt in building SENOPRA. In particular, Section 2.1 summarizes key characteristic of a trusted execution environment (TEE), and in particular Intel SGX. Section 2.2 introduces a confidentiality-preserving smart contract system that combines blockchains and trusted hardware. Finally, Section 2.3 reviews the KAC scheme.

2.1 Trusted Execution Environment

A trusted execution environment (TEE) is a secure area wherein integrity and confidentiality of computations and data are protected. The TEE can be provisioned either by trusted hardware [28, 46, 59] or a combination of hardware and software [24, 44, 45]. In the following, we focus our discussion on a particular TEE provider, namely Intel SGX [46]. Nonetheless, we emphasize that these characteristics also extend to other TEE implementation mechanisms in general.

Intel SGX [46] provides a CPU-protected TEE (or *enclave*) for generic computations. A host can instantiate multiple enclaves simultaneously. An enclave is assigned a guarded address space which is accessible only by the code running inside the enclave. All attempts to access the enclave memory made by an external process, including the operating systems and hypervisor, are blocked by the CPU. When data moves out of the enclave memory onto DRAM or external storage, it is encrypted using the processor's key. It is worth mentioning that the enclaves cannot directly execute

OS-provided services such as I/O or networking, and thus have to rely enclave boundary interface communicate with the OS or untrusted application that services OS-provided functions.

Intel SGX enables attestation mechanisms via which one can verify if a specific enclave has been correctly instantiated [19]. More specifically, after being instantiated, the enclave in question requests the CPU to compute a its measurement (i.e., the hash of its initial state). The measurement is then signed using CPU's private key under the Enhance Privacy ID (EPID) scheme [38]. A remote user obtaining the signed attestation verifies the signature using Intel's Attestation Service (IAS), and compares the measurement against a known value to determine the enclave's correct instantiation. Attestation mechanisms also allow the validator and the attesting enclave to establish a secure, authenticated channel via which sensitive data and application secrets can be communicated.

Intel SGX supports data sealing, a mechanism via which the enclave can securely persist its state onto non-volatile memory. The enclave first requests the CPU for a enclave-specific key, it then uses such key to encrypt data before storing it on persistent storage. Data sealing mechanism guarantees that the sealed data can only be accessed by the enclave that sealed it.

2.2 Confidentiality-Preserving Smart Contract Execution

Smart Contract. A *smart contract* (or contract) is an "autonomous agent" encoded in a decentralised blockchain. Once successfully created, a contract is uniquely identified by a contract address. A contract features a predefined executable code, and has its own storage. A contract can manipulate its state (e.g., variables, or data contained in its own storage) similar to a traditional imperative program. One can invoke a contract execution by sending a transaction to the contract address. Such transaction typically includes input data for the invocation, and a transaction fee paid for the execution.

Due to the decentralization nature of the blockchain, a contract execution (i.e., its computation and the computation result) is replicated across a network of participants (or miners) who reach agreement on the contract's state. Public blockchains, such as NEO [11] or Ethereum [3], implement full replication on all nodes, providing high level of fault tolerance at a cost that is orders of magnitude higher than conventional cloud computing platforms like AWS EC2 [1]. In addition, these platforms require the contract state and user input to be publicly visible, lacking privacy and confidentiality protection that sensitive application may requires.

Indeed, there are various applications that could benefit from a combination of decentralization and confidentiality protections. For instances, credit scoring industry attracts an annual revenue exceeding ten billions USD[1], but is concentrated among a few bureaus. This renders the industry prone to single point of failures and trustworthiness issues. Other prominent examples include private data sharing, health care analytics, voting and auctions applications. The above mentioned usecases motivate a decentralized confidentiality-preserving smart contract system.

Confidentiality-Preserving Smart Contract. Cheng et al. [25] propose a confidentiality-preserving smart contract system, called Ekiden, which leverages TEE for private contract execution. A confidential smart contract in Ekiden is similar to a traditional smart contracts in many ways, but features two distinguished properties. First, the contract code is executed inside an attested TEE that generates attestations proving the correctness of the code execution and state updates. This enables confidential transactions, as users can interact with the contract via a secure channel. Second, the contract's private state and data remain encrypted outside of the TEE, inaccessible to the public. The encryption key is contract-specific, and is only known to the TEE. This gives the contract an absolute control over its private state,

[1]http://clients1.ibisworld.com/reports/us/industry/ataglance.aspx?entid=1475

allowing it to dictate a policy under which the protected data is shared with other party (e.g., how, when and to whom the data is released).

Ekiden architecture decouples blockchain consensus from contract execution, allowing the system to scale horizontally. To enable fault-tolerance, it suffices to replicate the contract execution (and its state) on a few TEE-enabled compute nodes, as opposed to the entire network of miners as in public blockchain. This does not compromise the security, for integrity of contract execution has already been accounted for by the TEE-generated attestations. The Oasis Lab[2] implements Ekiden architecture in its Oasis Platform.

Recall that contract's private state and data are encrypted using a contract-specific key known only to the TEE, replicating the contract execution on different TEE-enabled compute nodes entails a careful key management procedure that guarantees the availability of the key in the event of TEE failure or compute nodes leaving the network. To facilitate the key management, each compute node maintains a key manager service inside a TEE. All key managers share a master key k_{master}. A newly instantiated key manager may retrieve k_{master} from existing key managers, the latest list of the latter is maintained on the blockchain. Communication between key managers is protected via secure and authenticated channels established through remote attestations. All contract-specific keys are encrypted using k_{master}, the resulting ciphertext are stored on the blockchain. The key manager TEE enforces contract-level key isolation, wherein a contract-specific key k_i of contract $\mathcal{E}_i$ is only accessible to contract TEEs with identity $\mathcal{E}_i$[3].

2.3 Key Aggregation Cryptosystem

Key-Aggregate Encryption (KAC) [26] is a public key cryptosystem that supports flexible delegation wherein any subset of ciphertexts (encrypted under the same public key PK) can be decrypted using a constant-size decryption key. Such decryption key can only be generated by a party who owns the master-secret key SK corresponding to PK. More specifically, given a plaintext x and an index $i \in [1, n]$, one can encrypt x under the public key PK to obtain a ciphertext c_i that is associated with the index i.

Subsequently, given any set of indices $S \subseteq \{1, 2, \ldots, n\}$, a party who holds a master-secret key SK can generate single constant-size decryption key K_S that can be used to decrypt any ciphertext c_i whose index i belongs to the set S. In addition, K_S must not reveal any information about SK, nore any ciphertext c_j whose index j does not belong to the set S. KAC's security is based on the decisional Bilinear Diffie-Hellman Exponent problem (BDHE)[22].

This KAC cryptosystem comprises of five basic polynomial-time algorithms, namely *Setup*, *KeyGen*, *Encrypt*, *Extract* and *Decrypt*:

- $param \leftarrow \textsc{Setup}(1^\lambda, n)$: Given security parameter λ and the number of ciphertext indices n, the *Setup*() algorithm randomly picks a bilinear group $\mathbb{G}$ of prime order p where $2^\lambda \leq p \leq 2^{\lambda+1}$, a generator $g \in \mathbb{G}$ and a random number $\alpha \in_R \mathbb{Z}_p$. It then computes and returns $param = \langle g, g_1, g_2, \cdots, g_n, g_{n+2}, \cdots, g_{2n} \rangle$ where $g_i = g^{\alpha^i}$.
- $(PK, SK) \leftarrow \textsc{KeyGen}()$: The algorithm picks a value $\gamma \in_R \mathbb{Z}_p$, and outputs the public and master-secret key pair: $(PK = v = g^\gamma, SK = \gamma)$.
- $\zeta \leftarrow \textsc{Encrypt}(PK, i, x)$: Given a public key PK, an index $i \in \{1, 2, ..., n\}$ and a message $x \in \mathbb{G}_\mathbb{T}$, the *Encrypt*() algorithm picks $t \in_R \mathbb{Z}_p$ uniformly at random. It then computes and returns a ciphertext $\zeta = \langle g^t, (vg_i)^t, x \cdot e(g_1, g_n)^t \rangle$.

[2]https://www.oasislabs.com/

[3]The contract TEE's identity is the hash of the contract code

- $K_S \leftarrow$ Extract(SK, S): Given a set S of indices j's and the master-secret key SK, the *Extract*() algorithms outputs the aggregated decryption key $K_S = \prod_{j \in S} g_{n+1-j}^{\gamma}$.
- $\{x, \perp\} \leftarrow$ Decrypt$(K_S, S, i, \zeta = \langle c_1, c_2, c_3 \rangle)$: If $i \notin S$, the *Decrypt*() algorithm outputs $\perp$. Otherwise, it returns the plaintext $x = c_3 \cdot e(K_S \cdot \rho, c_1)/e(\hat{\rho}, c_2)$ where $\rho = \prod_{j \in S, j \neq i} g_{n+1+i-j}$ and $\hat{\rho} = \prod_{j \in S} g_{n+1-j}$

The aggregated key K_S consists of one group element. As such, its size depends only on the security parameter λ . In another word, the size of the aggregated key K_S does not depends on the number of indices n, or the cardinality of the index set S. On the other hand, decrypting cost for each ciphertext grows proportionally to the cardinality of S. In particular, given a set of ciphertext C whose indices form an index set S, and the aggregated key K_S corresponding to S, decrypting each ciphertext $c \in C$ requires $O(|S|)$ group operations. This translates to a cost of $O(|S|^2)$ group operations to decrypt the entire ciphertext set C.

3 PROBLEM STATEMENT

In this section, we state the problem of protecting data privacy while still enabling data utility in IoT context. Section 3.1 gives an overview of privacy guarantees our design seek to offer, and challenges in achieving our goal. Next, Section 3.3 specifies the adversary model we consider. Finally, we introduce a baseline solution and analyze its shortcomings so as to motivate and justify the design choices that we make in constructing Senopra.

3.1 Problem Overview

Our work seeks to provision a data management framework that reconciles data privacy and utility. The framework enables an interaction between *data owners* and *data consumers*, wherein the former allows the latter to "use" their data for some specific tasks such as analytics or training of machine learning models, subject to the trust assumptions made between them. Our framework abstracts away from details of these trust assumptions, rather focusing on enforcing the access control requirement resulted from such trust assumptions.

The data management framework comprises of the following main components, namely *sensors*, *encrypted storage*, *policy agent* and *data consuming services*. Figure 1 depicts an overview of the framework architecture and its workflow.

- **Sensors** are IoT devices that are capable of capturing and pre-processing users' information, as well as sending and receiving data. The required pre-processing operations are lightweight, for examples AES [37] or SHA-2 [23]. The sensors interact with physical environment to capture users' information in real-time. The captured data are then encrypted using data owner-specific key before being sent off to be persisted on a storage.
- **Encrypted storage** is a key-value store that maintains the protected sensitive data. The framework is agnostic to the implementation details of the key-value store (e.g., LevelDB [9], RocksDB [15]). It is only assumed that the storage exposes two functions, which are `put(k, v)` and `get(k)`. The former persists the data contained in `v`, indexing it by `k`, while the latter retrieves data that is indexed by `k` if it has been persisted earlier.
- **Policy agent** enforces the access control policy set forth by data owners with respect to their sensitive data. It acts as a gateway via which eligible data consuming services could gain access to the protected data. In addition to the access control rules to be applied on the protected data, the policy agent also keeps track of the data pointers, and its decryption key materials. Given a data request from a data consuming service, the policy agent

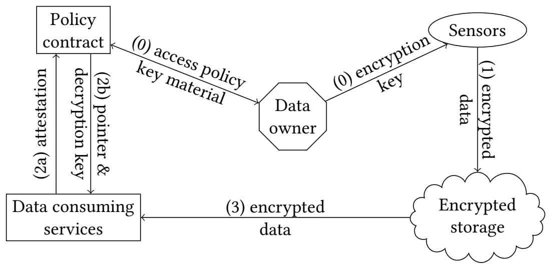

Fig. 1. An overview of a privacy-preserving data management framework.

first attests the eligibility of the service against the access control restrictions[4]. Upon successful verification, it communicates the pointer and the decryption key for the requested data to the requesting service in a secure manner.

- **Data consuming services** are programs belong to the data consumer, and wish to access the protected data. It is assumed that the service's program specification is known a priori, based on which data owners decide whether they should grant the service access to their data. The data consuming services are required to feature attested execution (i.e., they are able to prove that their execution correctly follow the predetermined program specifications). In practice, this can be realized by implementing the data consuming program inside a TEE that offers remote attestation, such as SGX enclave [46]. Upon receipt of the pointer and the decryption key for the requested data, the services retrieved encrypted data from the storage, and decrypt the data using the decryption key.

3.2 Goals

In designing a privacy-preserving data management framework, we seek to attain the following goals:

- **Data Ownership**: The framework ensures that data owners have absolute control over their data. Services that require access to the protected data must seek permissions from the data owners. These permissions are codified into an access control policy. The data owner is the sole entity that could alter the access control policy, either to grant new services access to her data, or to revoke access previously granted.
- **Fine-grained Access Control**: The access control policy should support fine granularity, allowing the data owner to share any subset of her data. Data to which access is not explicitly granted by the data owners should remain confidential.
- **Autonomous Access Control Management**: The access control management must be implemented as a smart contract whose execution is autonomous and decentralized, so as to avoid single point of failure and unnecessary involvement of a trusted third party.
- **Confidential policy agent**: The access policy internals (e.g., the white-list of permitted services) must be kept confidential. Requesting services only learn whether they are granted access permissions, but not the criteria by which the policy agent's decisions are made.

[4]The eligibility criteria include not only the requesting service's function execution, but also mechanism by which it handles the data's decryption key

- **Efficiency and Scalability**: As the sensors can be operating on low-powered devices, it is important to keep their computation load. Moreover, the framework should support a large number of data owners and services, wherein each data owner may specify different access control policy with respect to different service and subset of her data.

The resulting framework incorporating all the aforementioned goals will allow privacy-preserving data management at scale. Throughout the rest of the paper, we shall use the following running example to motivate and illustrate our design.

Running Example. Let us consider activity tracking, wireless-enabled wearable devices. Such a device captures various information of its user (i.e., individual that wears the device) such as her heart rate, number of steps walked or climbed, calories burned, routes traveled and locations visited during her daily routines. A fitness service provider (which might be a party independent from the wearable device manufacturer) would like to exploit data captured by the wearable devices to construct a machine learning model that facilitate exercise recommendations or workout personalisations. In order to attain high accuracy, the model requires a training dataset comprising data records collected from numerous users. Due to privacy concerns, the users are only willing to share with the above-mentioned fitness service provider data regarding their heart rate and calories burned, but not the routes they traveled or locations they visited.

Under the framework described in Section 3.1, the wearable devices are the sensors, and the fitness service provider is the data consumer service. The storage that stores the data captured by the devices can be provided by the wearable device manufacturer, or a third party (e.g., Google Cloud Healthcare [7]). Data breach incidents of major wearable device manufacturer and fitness service provider (e.g., Fitbit [6] and MyFitnessPal [10]) prompt the need for data encryption on the storage. Moreover, misuses of users' data by untrustworthy centralised party (e.g., Facebook Cambridge Analytica data scandal [5]) incite a decentralized approach to implementing access control management.

3.3 System and Adversary Models

This sections details the trust assumptions and threat model considered in our design and security analysis.

Sensor Assumptions. We assume that the sensors capturing users' data are correctly manufactured and free of backdoor. In addition, we also assume that their software does not deviate from its pre-programmed logic, and that there exists mechanism that allows user to securely provision secret (e.g., encryption key) to the sensors. While we acknowledge that assumptions are strong, they are necessary in our work. Hardening the sensors and other IoT devices against backdoor or potential software vulnerabilities are an interesting and indepdent line of works [47].

TEE Assumptions. A practical approach to implement attested execution for data consuming services and confidentiality-preserving smart contract is to leverage security properties offered by trusted execution environment (TEE). We assume that the mechanism provisioning the TEE (e.g., trusted hardware such as Intel SGX [46] or Sanctum [28]) is correctly implemented and securely manufactured. Recent attacks on SGX enclaves show potential threat to confidentiality, especially side-channel leakages [40, 56]. We discuss techniques to mitigate these attacks in Section 5.

Smart Contract Assumptions. To avoid involvement of a centralized party in implementing access control management, the access policy can be encoded into a smart contract whose executable code resides on a decentralised blockchain. We assume that the blockchain is trusted for integrity and availability. In another words, it is assumed that smart contract faithfully follows its prescribed logic, and is constantly available.

Threat Model. In our security analysis, we pay our primary attention to an adversary whose main goal is to violate the *data ownership* property set forth in the framework. In particular, the adversary aim to bypass the access control policy specified by the data owner with respect to her data, so as to learn the content of the encrypted data without permission granted by the data owner.

We assume that the adversary can take control over the encrypted storage. Moreover, it can also pose as data consuming services, requesting access to different portions of the protected data. In this case, it is critical to ensure that the adversary cannot "collude" these requests to reveal additional information beyond the portions of data it is explicitly granted access to. Revisiting the running example in Section 3.2, if a data consuming service is granted access to heart rate information of user U_A, and calories burned of user U_B, then it must not be able to obtain heart rate information of user U_B or calories burned of user U_A. This guarantee is formally modeled under a notion of *collusion resistance*. We refer readers to [22] for the formal model.

Nonetheless, we assume that the adversary is computationally bounded, and it cannot break the cryptographic primitives employed in our framework (e.g., encryption schemes such as AES and KAC, or EPID signature scheme). Further, the adversary is not able to subvert any security guarantee offered by the TEEs and the smart contract system. In addition, although we assume that sensors are trusted, in case an adversary manages to compromise or leak secrets a sensor holds, confidentiality of data captured by other sensors must not be compromised. Finally, we leave DoS attacks against the system beyond scope.

3.4 Baseline Approach

We now describes a baseline approach to privacy-preserving data management. We then discuss limitations of the baseline approach, in so doing we identify challenges we face in achieving the goals specified in Section 3.2. The baseline approach may make additional assumptions beyond those mentioned in Section 3.3. We shall clarify them as needed.

Workflow. A user U first generates an encryption key sk uniformly at random, and securely provision the key to her sensor. For every data record x_i that the sensor captures, it encrypts x under key sk using a symmetric-key encryption scheme, such as AES [42], obtaining $c_i \leftarrow \mathsf{Enc}(sk, x_i)$. The ciphertexts are then stored at a cloud storage CS.

The baseline solution relies on a centralised, trusted party TP to serve the role of the policy agent[5]. U informs TP on an access control policy of her data, and entrusts her secret key sk to TP. For simplicity, we assume that the data consuming program Prog can be uniquely identified by its measurement $\mathsf{M}_{\mathsf{Prog}}$(i.e., the hash of its initial state). Hence, the access control policy simply contains measurements of programs that are eligible to access U's data.

To ensure confidentiality of the sensitive data during the data consuming program Prog's computation, Prog must run inside a TEE, in particular a SGX enclave. The SGX architecture supports remote attestation, allowing TP to attest correct substantiation of Prog enclave, and to establish a secure communication channel to the enclave. Upon successful attestation, TP checks Prog's measurement $\mathsf{M}_{\mathsf{Prog}}$ against U's access control policy, and sends sk to the TP enclave via the secure channel if the access can be granted Prog enclave fetches the encrypted data from the storage CS. It then decrypts and processes the data inside the enclave. Once the computation finishes, Prog discards sk.

[5]This trust assumption regarding TP is not included in the system and adversary models specified in Section 3.3.

Limitations. The baseline approach described earlier features two key limitations. The first limitation is that the trust assumption made upon the centralised policy agent `TP` significantly weakens the security of the system. Indeed, `TP` becomes a single point of failure in the system. Moreover, the trustworthiness of `TP` should not be taken for granted without questions. Various incidents have demonstrated devastating consequences of placing too high a stake in a single hand [5, 6]. Nonetheless, "decentralizing" the role of `TP` by leveraging public smart contract platforms such as Ethereum [3] or NEO [11] is not trivial, as these platforms do not (yet) offer any privacy protection for the smart contract's data. Naively implementing `TP` as a smart contract would expose all decryption keys on a public ledger, defeating the privacy and data ownership protections. Zyskind et al. [61] proposed a similar platform wherein the smart contract system only handles pointers to the data, while decryption keys are shared directly between the users and the data consumers. It is unclear, nevertheless, how the shared keys, and by their extension, accesses to the encrypted data, can be efficiently revoked.

The second limitation of the baseline approach lies in its key management. Recall each data owner generates a unique key for her sensor, the number of keys that the policy agent has to manage grows proportionally with the number of data owners. To further exacerbate the issue, a data owner may use different keys to encrypt different data types. Repeating our running example, a user can configure her wearable device to encrypt her heart rate information, number of steps climbed, calories burned, and locations visited with different keys. In situation where a data consuming service acquires access to data records (e.g., heart beat and calories burned) of numerous users, the number of keys to be communicated between the policy agent and the data consuming service is significant.

4 SENOPRA DESIGN

This section presents Senopra– a privacy-preserving data management framework that augments the baseline approach introduced in Section 3.4 with confidentiality-preserving smart contract [25] and KAC [26], thereby addressing the baseline approach's limitations. Section 4.1 details how we implement the policy agent in a decentralised and secure manner by leveraging confidentiality-preserving smart contract, while Section 4.3 discusses Senopra's approach to handle key management efficiently at scale.

4.1 Decentralized & Confidentiality-Preserving Policy Agent

Senopra sidesteps the trust assumption that the baseline approach makes upon the policy agent by codifying its logic into a confidentiality-preserving smart contract, called `PolicyContract`whose logic is described in PseudoCode 1. On the one hand, the smart contract operates "autonomously" and its execution integrity is enforced by consensus protocol. This ensures that the policy agent never deviates from its prescribed protocol, faithfully enforcing the access control policy set forth by the data owners. On the other hand, `PolicyContract` keeps its internal state and private data (e.g., decryption key material) *encrypted* on-chain, only decrypting them when running inside a secure environment. Our description below is restricted to the most popular confidentiality-preserving smart contract platform called Oasis, but our design is applicable to other smart contract systems that provide similar capabilities and security guarantees.

Confidential private state. The contract maintains two private variables, namely `keyMaterials` and `ACPolicy`. The former consists of key materials used to devise decryption keys for eligible data consuming services, while the latter records data owners-defined access control policy. Notice that both variables are marked *confidential* using the `private` modifier. With Oasis platform, the `private` modifier states that the value of marked variables is not only inaccessible through transactions, but also stored encrypted on the blockchain, thereby preventing any node/worker to read it.

PseudoCode 1 `PolicyContract`

```
private keyMaterials;
private ACPolicy;
procedure SetPolicy($\mathsf{pk}_U$, k, wl)
    if Owner(k) = null then
        Owner(k) = $\mathsf{pk}_U$;
    if Owner(k) = $\mathsf{pk}_U$ then
        ACPolicy.Update(k, wl);
procedure HandleDataRequest($\mathsf{AttCert}_{\mathsf{Prog}}$, k)
    $\mathsf{M}_{\mathsf{Prog}}$ ←GetMeasurement ($\mathsf{AttCert}_{\mathsf{Prog}}$);
    if isValid($\mathsf{AttCert}_{\mathsf{Prog}}$) $\wedge$ $\mathsf{M}_{\mathsf{Prog}} \in$ ACPolicy[k] then
        GrantAccess($\mathsf{AttCert}_{\mathsf{Prog}}$, k);
procedure GrantAccess($\mathsf{AttCert}_{\mathsf{Prog}}$, k)
    dk ← getDecryptionKey(keyMaterials, k);
    $\mathsf{pk}_{\mathsf{Prog}}$ ← getPK($\mathsf{AttCert}_{\mathsf{Prog}}$);
    channelID ← EstablishSecureChannel($\mathsf{pk}_{\mathsf{Prog}}$);
    Send(channelID, dk);
```

The latter requirement (i.e., being stored encrypted on-chain) differentiates Oasis from Ethereum and other platforms without confidentiality, wherein the `private` modifier only restricts access from transactions, but does not provide any confidentiality, for the value of the field is stored in plaintext on-chain [2]. These private variables are encrypted using contract-specific key which resides exclusively inside the contract TEE. TEE-enabled compute nodes that replicate execution of the same contract manage the contract-specific key via key manage enclaves at discussed in Section 2.2. Due to space contrainst, we refer readers to Oasis platform's specifications [2, 25] for further details on confidentiality protection of contract's private state.

Setting Access Policy. Without loss of generality, the `PolicyContract` in Senopra identifies a data record v by its hash digest $\mathsf{k} = H(\mathsf{v})$, where $H(\cdot)$ is a cryptographic hash function [42], a data owner U by her public key pk_U, and a data consuming program `Prog` by its measurement $\mathsf{M}_{\mathsf{Prog}}$. An access control policy for a data record with identifier k is simply a white list `wl` containing measurements of eligible data consumer services. U can set an access control policy her data by sending a transaction that invokes `PolicyContract`'s SetPolicy function with appropriate payload.

Attesting Data Consuming Service. Similar to the baseline approach, Senopra requires the data consuming service `Prog` to run inside an SGX enclave. The `Prog` enclave has its own (unique) public and private key pair ($\mathsf{pk}_{\mathsf{Prog}}$, $\mathsf{sk}_{\mathsf{Prog}}$) generated uniformly at random during the enclave instantiation. Once `Prog` enclave is instantiated, the data consumer (i.e., a party who runs `Prog`) obtains from the SGX processor `Prog`'s remote attestation $\pi_{\mathsf{Prog}} = \langle \mathsf{M}_{\mathsf{Prog}}, \mathsf{pk}_{\mathsf{Prog}} \rangle_{\sigma_{TEE}}$, in which $\mathsf{M}_{\mathsf{Prog}}$ is the enclave's measurement, and σ_{TEE} is a group signature signed by the processor's private key under the Enhance Privacy ID (EPID) scheme. The attestation π_{Prog} substantiates the correct instantiation of the `Prog` enclave and its public key. Under the current SGX architecture, verifying σ_{TEE} requires involvement of the Intel's Attestation Service (IAS) [8]. Instead of requiring `PolicyContract` to contact the IAS to verify the remote attestation π_{Prog}, Senopra converts π_{Prog} into a publicly verifiable certificate, thereby allowing `PolicyContract` to verify the certificate internally. More specifically, the data consumer contact the IAS for verification of π_{Prog}, which then responds with $\mathsf{AttCert}_{\mathsf{Prog}} = \langle \pi_{\mathsf{Prog}}, \mathsf{validity} \rangle_{\sigma_{IAS}}$, in which σ_{IAS} is the IAS's publicly verifiable signature [14], and `validity` flag indicates if π_{Prog} is valid.

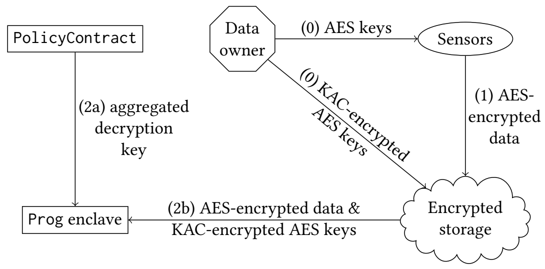
PolicyContract
Data owner
(0) AES keys
Sensors
(0) KAC-encrypted AES keys
(1) AES-encrypted data
(2a) aggregated decryption key
Prog enclave
(2b) AES-encrypted data & KAC-encrypted AES keys
Encrypted storage

Fig. 2. An overview of SENOPRA's KAC-incorporated scalable key management.

The attestation certificate $\texttt{AttCert}_{\texttt{Prog}}$ allows `PolicyContract` to attest correctness of and establish a secure channel to the `Prog` enclave. If `Prog` is eligible to access the requested data, `PolicyContract` derives a decryption key `dk` from its private data `keyMaterials`, and communicates `dk` to the `Prog` enclave via a secure channel established earlier. We discuss in the next section details on what constitutes `keyMaterials` and how the key derivation process is carried out so as to attain both efficiency and scalability in key management.

4.2 Scalable Key Management

Recall that in the baseline approach, `keyMaterials` essentially comprises encryption/decryption keys of all data owners. In order to operate at scale, SENOPRA has to support several thousands, if not millions, of data owners. Given that each data owner register at least one key with the system, a trivial key management approach such as one featured in the baseline approach will yield a significantly large `keyMaterials`. This is undesired in SENOPRA, for and on-chain storage is expensive (e.g., storing 1KB of data (split into several blocks) costs 0.00128BTC on Bitcoin chain and 0.032 ETH on Ethereum chain[6], which is several orders of magnitude higher than conventional cloud computing platforms like AWS EC2). Moreover, when data consuming service acquires access to data records from numerous users, the number of keys communicated to the data consuming service is an important factor worthy of consideration.

SENOPRA incorporates KAC [26] to attain scalability in key management. As introduced in Section 2.3, KAC allows flexible delegation wherein any subset of ciphertexts (encrypted under the same public key PK) can be decrypted using a constant-size decryption key, which is derived from the master-secret key SK associating with PK. By encrypting data owners' key under KAC and storing the encrypted keys along with the encrypted data, SENOPRA reduces the `keyMaterials` that `PolicyContract` must maintain to a single master-secret key SK. Moreover, `PolicyContract` can grant the data consuming service access to a large number of data records from multiple data-owners by issuing a single aggregated decryption key. In the following, we elaborate on this intuition.

For clarity of exposition, our discussion assumes that the the sensors use encrypt its captured data using AES encryption [42], and refer to the encryption keys in use as AES keys. Figure 2 depicts SENOPRA's approach to scalable key management via KAC.

During the system bootstrap, the `PolicyContract` execute KAC's SETUP() and KEYGEN() algorithms, obtaining a pair of public and master-secret key (PK, SK). The contract keeps the master secret-key SK in its private `keyMaterials` variable, while putting the public key PK on an off-chain storage with integrity protection. After securely generating and provisioning the AES keys to the sensors, the data owners encrypt these keys under KAC, and store the ciphertexts of the AES keys at the storage `CS` (i.e., the same storage that their encrypted data are stored). We remark that each

[6]https://ethereum.stackexchange.com/questions/872/what-is-the-cost-to-store-1kb-10kb-100kb-worth-of-data-into-the-ethereum-block

ciphertext in KAC is associated with an index i, based on which the aggregated decryption key is computed. Here, it suffices to assume that each AES key is associated with a unique index. Section 4.3 elaborates on how data owners derive these indexes.

Let us denote by `EncryptedData` a set of AES-encrypted data records that the `PolicyContract` is to grant the data consuming service `Prog` access to. The ciphertexts of the AES keys used to decrypt `EncryptedData` constitutes a set C. Let S be the set of indexes associated with ciphertexts in C. The `PolicyContract` derives the aggregated decryption dk by executing KAC's Extract() algorithm with inputs $\langle SK, S\rangle$. dk is then sent to the `Prog` enclave. In addition to the AES-encrypted data, the `Prog` enclave also fetches C from the storage CS. It then reconstructs the AES keys from C by repeatedly executing KAC's Decrypt() algorithm with inputs $\langle \mathsf{dk}, S, i, c_i\rangle$ for every $c_i \in C$ whose associated index is i. After obtaining the AES keys, the `Prog` enclave uses them to decrypt and process `EncryptedData`.

4.3 Efficient Key Reconstruction

Incorporating KAC allows Senopra to keeps the size of both `PolicyContract`'s `keyMaterials` and the amount of keys communicated to the data consuming program `Prog` small, regardless of the number of data owners or volume of data `Prog` acquires access to. This allows Senopra to operate at scale. Nonetheless, as we briefly mentioned in Section 2.3, the computational cost incurred by decrypting all encrypted keys in the set C is $O(|C|^2)$, assuming each ciphertext $c_i \in C$ is associated with an unique index i (i.e., $|S| = |C|$). This poses a significant expense on `Prog`, for it has to C prior to accessing `EncryptedData`. We present in this section an algorithmic enhancement that reduces the decryption cost of C to $O(|C|)$, provided that the set of associated indexes S adheres to specific structures stipulated below.

Quadratic Key Reconstruction Cost. For clarity, let us revisit KAC's Encrypt and Decrypt algorithms. On input $\langle PK, i, x\rangle$ comprising the public key, the index i to be associated with the plaintext x, Encrypt returns:

$$c_i \leftarrow \langle g^t, (vg_i)^t, x \cdot e(g_1, g_n)^t\rangle$$

where v is essentially the PK, $\langle g, g_1, g_i, g_n\rangle$ are drawn from the public parameter $param$, and t is chosen uniformly at random. On the other hand, given the input $\langle \mathsf{dk}, S, i, c_i\rangle$ with $i \in S$, Decrypt returns:

$$x = c_{i[2]} \cdot e(K_S \cdot \rho_i, c_{i[0]})/e(\hat{\rho}, c_{i[1]})$$

where

$$\rho_i = \prod_{j\in S, j\neq i} g_{n+1+i-j}$$

and

$$\hat{\rho} = \prod_{j\in S} g_{n+1-j}$$

While $\hat{\rho}$ is independent of i and can be computed once and then reused for all ciphertexts in C, ρ_i is index specific, and its computation requires $O(|C|)$ group multiplications (as $|S| = |C|$). Consequently, decrypting all ciphertexts $c_i \in C$ incurs quadratic overhead of $O(|C|^2)$.

Key Observation. To reduce the cost of decrypting C, we exploit a key observation that should indexes in S bear a special recurrent relation, there exists an algorithm that computes *all* ρ_is in linear time. To clarify this observation, let us consider an index set $S = [1, m]$ for some m, observing a recurrent relation that $S_{[i+1]} = S_{[i]} + 1, \forall i \in [0, m-1]$. Let us define $\hat{g}_t$ and R_i as follows:

$$\hat{g}_t = g_{n+1+t}$$

and

$$R_i = \prod_{j \in S} \hat{g}_{i-j}$$

We can then rewrite ρ_i (where $i \in S$) as $\rho_i = \hat{g}_i^{-1} R_i$. By exploiting the recurrence relation:

$$R_{i+1} = (\hat{g}_{i-m})^{-1} \cdot R_i \cdot \hat{g}_i$$

we can obtain R_{i+1} from R_i using only two extra multiplications. This yields an algorithm that recursively computes all R_is (and by their extensions ρ_is and decryptions of ciphertexts in C) in linear time, thereby offering a significant speed-up over quadratic time incurred by an naive approach of computing all R_is independently.

Index Pattern for Fast Reconstruction. We now generalise the key observation explained earlier, defining an index pattern that allows fast reconstruction. The original KAC construction [26] assumes that indexes associated with the ciphertexts form an integer interval $[1, n]$. In another word, each index is assumed to be an integer. Senopra on the other hand, allows data owners to pick a d-dimensional point from a d-dimensional lattice L_Δ to serve as an index for their AES key, enabling the data owners to define their access control policy in finer-grained. For example, an AES key can be indexed by a three-dimensional point representing a 3-tuple $\langle$ data owner's ID, data type, service type $\rangle$.

The d-dimensional lattice L_Δ is defined as $L_\Delta = [1, T_1] \times [1, T_2] \times \ldots \times [1, T_n]$, with bounds $T_1, T_2, \ldots, T_d$. Put differently, an interval $[1, T_k]$ represents possible values in L_Δ's k^{th} dimension. A d-dimensional point p^d can be "flattened" into an integer p^1 as follows:

$$p^1 = p_1^d \times \prod_2^d T_j + p_2^d \times \prod_3^d T_j + \ldots + p_d^d$$

wherein p_k^d is the k^{th} dimension component of p^d.

A ciphertext set C can be decrypted in linear time $O(|C|)$ if its corresponding index set S forms either a d-dimensional hyper-rectangle $\mathcal{R} = r_1 \times r_2 \times \ldots \times r_d$ where each r_k is an interval in the k^{th} dimension of L_Δ, or a t-dimensional down-sampled lattice $\mathcal{L} = \{\sum_{i=1}^t a_i v_i | a_i \in \mathbb{Z}\} \cap \mathbb{L}_\Delta$ where $t \leq d$, each of the v_i is a d-dimensional vector and the basis $\{v_1, v_2, .., v_t\}$ is independent. This is so because such the d-dimensional indices in such S, when being "flattened", would yield into an integer array that observe the recurrent relation described in the key observation.

Linear Key Reconstruction Time. We first discuss how data owners should index their AES keys so that the data consuming services could efficiently reconstruct the encrypted keys. To this end, Senopra identifies a set of common data types and services that are interested in accessing those data types. For example, in our running example, a fitness service provider is typically interested in data concerning users' physical activities, for examples calories burned, heart rate and routes travelled. Data owners are more likely to share calories burned and heart rate information, but are more conservative about revealing their trajectory due to privacy concern. Senopra then suggests to a data owner U to encrypt all data records containing her calories burned and heart rate information with one AES key, while her trajectory information with another AES key. U then indexes the former with a three-dimensional point representing a 3-tuple $\langle$ U's ID, *non-sensitive, FitnessService* $\rangle$, and the former with $\langle$ U's ID, *sensitive, TrustedService* $\rangle$ wherein only a select few, if not none, services are granted *TrustedService* label. With this approach, data records that are typically requested together would have their encrypted AES-keys observing the special index pattern described earlier. In particular, indexes associated with the AES-keys would form a d-dimensional hyper-rectangle.

We now explain how data consuming service efficiently reconstruct AES-keys whose indexes form a d-dimensional hyper-rectangle. Let us consider a two-dimensional lattice with bound n in both dimensions (i.e., $\mathrm{L}_\Delta = [1, n] \times [1, n]$, and a two-dimensional hyper-rectangle $S = [1, m] \times [1, m]$. As such, the indices are two-dimensional vectors. We denote by $\sigma(x_1, x_2) = x_1(n-1) + x_2$ the function that maps two-dimensional point into an integer. Decrypting a ciphertext with the index (i_1, i_2) requires the following value:

$$\rho_{(i_1,i_2)} = \prod_{(j_1,j_2)\in S,(j_1,j_2)\neq(i_1,i_2)} g_{n^2+1+\sigma(i_1,i_2)-\sigma(j_1,j_2)}$$

We simplify the formula by defining $\hat{g}_{(i_1,i_2)}$ and $R_{(i_1,i_2)}$:

$$\hat{g}_{(i_1,i_2)} = g_{n^2+1+\sigma(i_1,i_2)}$$

and

$$R_{(i_1,i_2)} = \prod_{(x,y)\in S} \hat{g}_{(i_1,i_2)-(x,y)} = \prod_{x=1}^{m}\prod_{y=1}^{m} \hat{g}_{(i_1,i_2)-(x,y)}$$

Clearly, obtaining $\rho_{(i_i,i_2)}$ from the corresponding $R_{(i_1,i_2)}$ is trivial. By exploiting the observation on the special recurrence relation explained earlier, we derive the following equation:

$$R_{(i_1+1,i_2)} = R_{(i_1,i_2)} \prod_{y=1}^{m} \hat{g}^{-1}_{(i_1,i_2)-(i_1-m,y)} \prod_{\widetilde{y}=1}^{m} \hat{g}_{(i_1,i_2)-(i_1,\widetilde{y})} \tag{1}$$

Denoting the product of the first sequence $T_{(i_1,i_2)}$ and the second $\widetilde{T}_{(i_1,i_2)}$, Equation 1 becomes:

$$R_{(i_1+1,i_2)} = R_{(i_1,i_2)} T_{(i_1,i_2)} \widetilde{T}_{(i_1,i_2)} \tag{2}$$

Since both $T_{(i_1,i_2)}$ and $\widetilde{T}_{(i_1,i_2)}$ can also be expressed by recurrence relations, their computations take linear time. Consequently, all $R_{(i_1,i_2)}$s can be evaluated in linear time. By extending these derivations to higher dimensions, we need only $O(d|S|)$ group multiplications to compute all necessary ρ_i of ciphertexts whose indexes form a d-dimensional hyper-rectangle. As the number of dimensions d is a constant, this approach runs in linear time.

In our prototype implementation of Senopra, we assume all indexes are three-dimensional lattice. It is trivial to "lift" one or more uni-dimensional component to multi-dimensions, should the application scenario in reality so require. We also remark that in Senopra, data owners retain sole discretion on indexing their AES keys. The suggested pattern discussed earlier serve as an optimization that allow efficient key reconstruction at the data consuming service. As we shall elaborate in the next section, KAC's security ensures that index configuration of one AES key does not affect confidentiality of another AES key (which may belongs to either the same or different data owner).

5 SECURITY ANALYSIS

In designing Senopra, our contributions focus primarily on pragmatic aspect of the system, while relying on various existing primitives to achieve security. Therefore, we present in this section an informal security analysis of our design, and refer the readers to formal model of the employed security primitives when necessary.

5.1 Data Ownership & Fine-grained Access Control

Senopra ensures that data owners have absolute control over their data by encrypting sensitive data with AES encryption under user-specific keys. To enable fine-grained access control, Senopra allows data owners to encrypt their data under different keys, so that access to each subset of data is governed by a separate key, and therefore independent. To attain scalable and efficient key management, the AES keys are then encrypted under KAC and stored on the encrypted storage along with the encrypted data. Thus, provided that the AES and KAC encryption schemes are not compromised, Senopra secures users' sensitive data at rest. Formal treatments on security definition and proofs of AES can be found in [42], and those of KAC are presented in [26]. We remark that the algorithmic enhancement that reduces decryption cost that we discussed in Section 4.3 purely focuses on how to evaluate components required in KAC's Decrypt() algorithm, neither alter nor affecting any of its security guarantee.

To protect data confidentiality once it is fed as input to the eligible and approved data consuming services, Senopra require these services to run in an attested TEE, in particular Intel SGX enclave. The enclaves offer confidentiality and integrity protections for data and code loaded inside the enclave memory, ensuring that the data consuming services faithfully do not leak sensitive data to any unauthorised party. Formal foundation of enclave execution, and SGX specifications are discussed in [27, 55].

SGX's enclave execution is susceptible to side-channel leakages [40, 56]. More specifically, the access pattern incurred by data (or code page) moving in and out of the enclave (due to the hard limit of enclave memory [17] may leak sensitive information about the code or data processed inside the enclave. Various defences techniques to harden and protect the enclave execution against these side-channel leakage exist in the literature [32, 49]. These techniques render the enclave execution is data oblivious, wherein the access pattern do not depend on, and therefore do not reveal any information about, the input data. Data consuming services in Senopra can incorporate these protections to mitigate side-channel leakage, albeit at a certain computation overhead [32].

5.2 Policy Agent

Senopra implements the policy agent as a private smart contract named `PolicyContract` on Oasis platform. By exploiting autonomous and decentralized nature of the smart contract, Senopra avoid single point of failure and unnecessary involvement of a trusted third party. It is also worth noting that the confidentiality-preserving property of Oasis platform is essential to Senopra for `PolicyContract` is tasked to maintain key materials necessary to devise decryption keys for eligible data consuming services, and records access control policy defined by the data owners. Implementing `PolicyContract` on conventional smart contract platform that does not offer any privacy protection would expose the key materials to the public, thereby compromising Senopra's security.

In the Oasis platform, the availability and fault tolerance of `PolicyContract` is provisioned by the underlying consensus protocol run among a number of nodes in the network, while `PolicyContract`'s execution integrity and confidentiality of its private state (i.e., `keyMaterials` and `ACPolicy`) are safeguarded by the use of TEEs on the compute nodes. We refer readers to [25] for formal security model and protocol specifications of the Oasis smart contract platform.

6 EVALUATION

In this section, we evaluate the performance of Senopra prototype implementation. Recall that in Senopra, data records are encrypted under AES with keys generated uniformly at random; the AES-keys are then encrypted under KAC. Once a data consuming service is granted access to the protected data, it first decrypts the KAC-encrypted AES-keys using the aggregated decryption key `dk` obtained from the `PolicyContract`, and then use the AES-keys to access the encrypted data. Our empirical evaluation aim to quantify the cost of adopting KAC in Senopra design. In particular, we measure the encryption time and ciphertext size of KAC. Moreover, we compare the size of the key materials that the

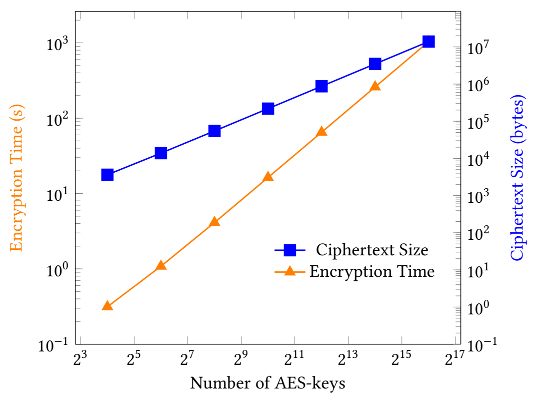

Fig. 3. KAC encryption time and ciphertext size

policy agent has to manage in the baseline solution and Senopra. Finally, we report the key reconstruction time of the data consuming services, showing significant performance gain offered by the fast reconstruction technique discussed in Section 4.3.

Experimental Setup. We conduct the experiments in on a system equipped with Intel i7-6820HQ 2.70GHz CPU, 16GB RAM, and running Ubuntu 16.04 Xenial Xerus. The KAC implementation is written in C and employs the Pairing-Based Cryptography (PBC) Library [13] for Type-A elliptic-curve group and pairing operations. It is configured with 160-bit Solinas prime, offering 1024-bit of discrete-logarithm security. The AES keys used in our experiments are 256-bit keys. We implement the `PolicyContract` in Rust [16] and deploy it on Oasis Devnet [2]. Our experiments use Intel SGX to provision TEE, and Intel SGX SDK to implement TEE's codebase. We repeat all experiments over 10 runs, and report the average results.

KAC encryption time. Figure 3 reports encryption time and ciphertext size of KAC in log scale against varying number of AES keys. In our experiments, each KAC encryption takes approximately 10 ms, producing a 210 bytes ciphertext (i.e., KAC-encrypted AES key). In reality, we expect each data owner in Senopra to utilize no more than a few hundreds different AES keys to encrypt her data. For such a number of AES keys, the per-data owner overhead incurred by adopting KAC in key management is negligible.

`keyMaterials` size. Figure 4 contrasts the size of the key materials the policy agent has to handle in the baseline approach against that in Senopra. In the baseline solution, the key materials comprise all the data owners' AES keys. On the other hand, key material in Senopracontains a single KAC's master-secret key *SK* regardless of the number of data owners and their AES keys involved in the system. This clearly shows the superiority of Senopra over the baseline solution in enabling key management at scale.

KAC key derivation time. Figure 5 plots the cost of deriving the aggregated decryption key dk with respect to the number of AES-keys that dk grants access to. We measure this cost in three settings, namely non-SGX (compute only), SGX (compute only), and Oasis Devnet. The first two settings involve running KAC's Extract() algorithm in an untrusted execution environment and SGX enclave, respectively. In the last setting, we deploy `PolicyContract` on Oasis Devnet and measures the end-to-end latency of data consuming service's request.

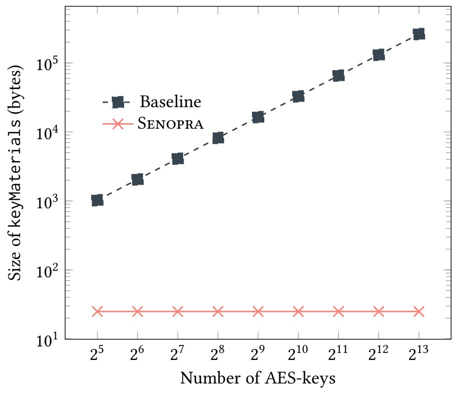

Fig. 4. Size of `keyMaterials`

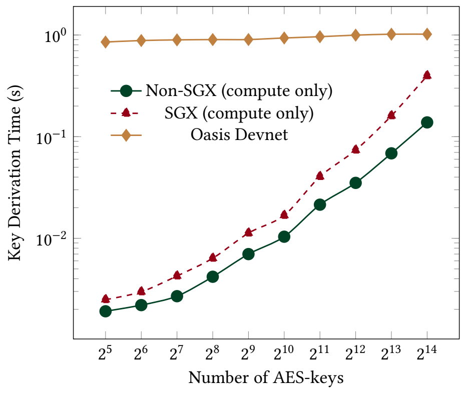

Fig. 5. Key Derivation Time

We observe that running EXTRACT() in SGX enclave incurs an overhead ranging from 35% to 150% compared to non-SGX setting. This overhead is primarily due to enclave switching needed for I/O operations. While the relative overhead factor appears significant, the actual running time is in sub-second. The end-to-end latency of `PolicyContract` observed on the Oasis Devnet remains roughly the same for varying request size, for the latency is dominated by the cost of the underlying blockchain consensus protocol. As the blockchain consensus engine is further improved, we expect the end-to-end latency of `PolicyContract` to further reduces. Even with the current latency of a few seconds, we believe that it is sufficiently efficient to support real-time operations.

Key reconstruction time. Finally, we quantify the cost of reconstructing KAC-encrypted AES-keys given the aggregated decryption key dk. Figure 6 plots the reconstruction time of the original KAC (i.e., sequentially invoking KAC's DECRYPT() on each encrypted AES-key in C), and that of our optimization described in Section 4.3. For comparison, we include running time observed in two settings, namely non-SGX (i.e., untrusted execution environment) and SGX (i.e., enclave execution). The experiment results show that our optimization indeed achieves linear reconstruction time. This leads to significant performance gain when the number of AES-keys to be shared is large. For instance, with 2^{14} AES-keys, our optimization offers upto 56× speed-up over the original KAC's reconstruction.

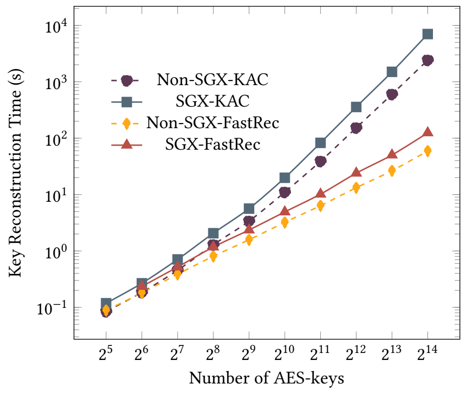

Fig. 6. Key Reconstruction Time

In this set of experiments, we observe the overhead incurred by SGX enclave execution over the non-SGX settings ranges from 40% to 125%. While our prototype implementation of SGX-compatible codebase (e.g., KAC's Extract() and Decrypt() algorithms) do not incorporate techniques that minimize the overhead of enclave execution [50], they can be employed in Senopra to reduce the cost of enclave execution. We plan to implement this extension in our future work.

7 RELATED WORK

Trusted Hardware-assisted Systems. Various works have advocated exploiting trusted hardware, most notably Intel SGX, to reconcile the trade-off between security and performance of different applications domains. Examples include machine learning [49], data analytics [32, 52], outsourced storage [30], or computation marketplace [33]. Senopra follows a similar spirit, leveraging trusted hardware to provision attested TEE that offers confidentiality and integrity protections for trusted codebase.

SGX has also been widely adopted in blockchain systems for various purposes, including scaling blockchain transaction throughput [31], implementing authenticated data feed for smart contract [60], or providing confidentiality and privacy protections on public blockchain [25]. Senopra leverages the Oasis platform to implements the confidential `PolicyContract` smart contract that securely and autonomously enforces data owners' access control policy.

Blockchain-powered Data Management. Zyskind et al. [61] is among the first to propose the use of blockchain in data management. The authors describe an architecture where in sensitive data are stored encrypted in a distributed hash table, and pointers to the data are logged on the blockchain. Unlike Senopra, this architecture assumes decryption keys are shared directly between the users and the data consumers, and does not consider revoking access that is previously granted to a data consuming service.

Various works have since followed the approached proposed in [61]. They focus mainly on healthcare and medical data [20, 39], and IoT applications [51]. Nonetheless, they either not fully address the privacy aspect of data sharing and exchange, or requires a strong assumption on certain trusted party. In contrast, Senopra offers privacy protection in data management without relying on any trusted third party.

8 CONCLUSION

We have presented Senopra- a privacy-preserving data management framework. The proposed framework gives the data owners absolute control over their personal and sensitive data, allowing them to specify a fine-grained access policy governing how the captured data are accessed. The access policy is enforced by a confidential smart contract named `PolicyContract` implemented on the Oasis platform. As demonstrated by our experiments, Senopra attains both scalability and efficiency in key management by incorporating KAC with an algorithmic enhancement that allows linear time key reconstruction.

REFERENCES

[1] Amazon AWS EC2. https://aws.amazon.com/ec2/.

[2] Developing Confidential Smart Contracts in Oasis. https://docs.oasiscloud.io/en/latest/confidentiality-develop/.

[3] Ethereum: Blockchain App Platform. https://www.ethereum.org/.

[4] EU General Data Protection Regulation. https://ec.europa.eu/commission/priorities/justice-and-fundamental-rights/data-protection/2018-reform-eu-data-protection-rules_en.

[5] Facebook Cambridge Analytica data scandal. https://en.wikipedia.org/wiki/Faceboo-Cambridge_Analytica_data_scandal.

[6] Fitbit hack. https://www.databreachtoday.com/fitbit-hack-what-are-lessons-a-8793.

[7] Google Cloud Healthcare. https://cloud.google.com/healthcare/.

[8] Intel Software Guard Extensions: Intel Attestation Service API. https://software.intel.com/sites/default/files/managed/7e/3b/ias-api-spec.pdf.

[9] LevelDB. http://leveldb.org//.

[10] MyFitnessPal hack. https://www.cnet.com/how-to/150-million-myfitnesspal-accounts-were-hacked-heres-what-to-do/.

[11] Neo: An Open Network For Smart Economy. https://neo.org/.

[12] Oasis Labs. https://www.oasislabs.com/.

[13] Pairing-Based Cryptography Library. http://crypto.stanford.edu/pbc.

[14] Public Key for Intel Attestation Service. https://software.intel.com/en-us/sgx/resource-library.

[15] RocksDB. https://rocksdb.org/.

[16] Rust. https://www.rust-lang.org/.

[17] SGX Developer Reference. https://download.01.org/intel-sgx/linux-1.9/docs/Intel_SGX_SDK_Developer_Reference_Linux_1.9_Open_Source.pdf.

[18] SingHealth Data breach. https://en.wikipedia.org/wiki/SingHealth#Data_breach.

[19] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. 2013. Innovative technology for CPU based attestation and sealing. In *HASP*.

[20] Asaph Azaria, Ariel Ekblaw, Thiago Vieira, and Andrew Lippman. 2016. Medrec: Using blockchain for medical data access and permission management. In *OBD*.

[21] Roberto J Bayardo and Rakesh Agrawal. 2005. Data privacy through optimal k-anonymization. In *ICDE*.

[22] Dan Boneh, Craig Gentry, and Brent Waters. 2005. Collusion resistant broadcast encryption with short ciphertexts and private keys. In *CRYPTO*.

[23] Ricardo Chaves, Georgi Kuzmanov, Leonel Sousa, and Stamatis Vassiliadis. 2006. Improving SHA-2 hardware implementations. In *CHES*.

[24] Xiaoxin Chen, Tal Garfinkel, E Christopher Lewis, Pratap Subrahmanyam, Carl A Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan RK Ports. 2008. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. *SIGOPS OSR* (2008).

[25] Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah Johnson, Ari Juels, Andrew Miller, and Dawn Song. 2018. Ekiden: A Platform for Confidentiality-Preserving, Trustworthy, and Performant Smart Contract Execution. *arXiv preprint arXiv:1804.05141* (2018).

[26] Cheng-Kang Chu, Sherman SM Chow, Wen-Guey Tzeng, Jianying Zhou, and Robert H Deng. 2014. Key-aggregate cryptosystem for scalable data sharing in cloud storage. *TPDS* (2014).

[27] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *IACR Cryptology ePrint Archive* 086 (2016).

[28] Victor Costan, Ilia A Lebedev, and Srinivas Devadas. 2016. Sanctum: Minimal Hardware Extensions for Strong Software Isolation.. In *USENIX Security*.

[29] Hung Dang and Ee-Chien Chang. 2015. PrAd: enabling privacy-aware location based advertising. In *GeoPrivacy*.

[30] Hung Dang and Ee-Chien Chang. 2017. Privacy-preserving data deduplication on trusted processors. In *IEEE CLOUD*.

[31] Hung Dang, Anh Dinh, Ee-Chien Chang, and Beng Chin Ooi. 2018. Chain of Trust: Can Trusted Hardware Help Scaling Blockchains? *arXiv preprint arXiv:1804.00399* (2018).

[32] Hung Dang, Tien Tuan Anh Dinh, Ee-Chien Chang, and Beng Chin Ooi. 2017. Privacy-preserving computation with trusted computing via Scramble-then-Compute. *PETs* (2017).

[33] Hung Dang, Dat Le Tien, and Ee-Chien Chang. 2018. Fair Marketplace for Secure Outsourced Computations. *arXiv preprint arXiv:1808.09682* (2018).

[34] Cynthia Dwork. 2008. Differential privacy: A survey of results. In *TAMC*.

[35] Craig Gentry et al. 2009. Fully homomorphic encryption using ideal lattices. In *STOC*.

[36] Saikat Guha, Bin Cheng, and Paul Francis. 2011. Privad: Practical privacy in online advertising. In *NSDI*.

[37] Panu Hamalainen, Timo Alho, Marko Hannikainen, and Timo D Hamalainen. 2006. Design and implementation of low-area and low-power AES encryption hardware core. In *DSD*.

[38] Simon Johnson, Vinnie Scarlata, Carlos Rozas, Ernie Brickell, and Frank Mckeen. 2016. Intel Software Guard Extensions: EPID Provisioning and Attestation Services. (2016).

[39] Tsung-Ting Kuo, Hyeon-Eui Kim, and Lucila Ohno-Machado. 2017. Blockchain distributed ledger technologies for biomedical and health care applications. *Journal of the American Medical Informatics Association* (2017).

[40] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. 2017. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *USENIX Security*.

[41] Ninghui Li, Tiancheng Li, and Suresh Venkatasubramanian. 2007. t-closeness: Privacy beyond k-anonymity and l-diversity. In *ICDE*.

[42] Yehuda Lindell and Jonathan Katz. 2014. *Introduction to modern cryptography*. Chapman and Hall/CRC.

[43] Ashwin Machanavajjhala, Johannes Gehrke, Daniel Kifer, and Muthuramakrishnan Venkitasubramaniam. 2006. l-Diversity: Privacy Beyond k-Anonymity. In *ICDE*.

[44] Jonathan M McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. 2010. TrustVisor: Efficient TCB reduction and attestation. In *IEEE S&P*.

[45] Jonathan M McCune, Bryan J Parno, Adrian Perrig, Michael K Reiter, and Hiroshi Isozaki. 2008. Flicker: An execution infrastructure for TCB minimization. In *SIGOPS OSR*.

[46] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. 2013. Innovative instructions and software model for isolated execution. *HASP@ISCA* (2013).

[47] Mujahid Mohsin, Zahid Anwar, Ghaith Husari, Ehab Al-Shaer, and Mohammad Ashiqur Rahman. 2016. IoTSAT: A formal framework for security analysis of the internet of things (IoT). In *CNS*.

[48] Arvind Narayanan and Vitaly Shmatikov. 2006. How to break anonymity of the netflix prize dataset. *arXiv preprint cs/0610105* (2006).

[49] Olga Ohrimenko, Felix Schuster, Cédric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. 2016. Oblivious Multi-Party Machine Learning on Trusted Processors. In *USENIX Security*.

[50] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. 2017. Eleos: ExitLess OS services for SGX enclaves. In *EuroSys*.

[51] Aafaf Ouaddah, Anas Abou Elkalam, and Abdellah Ait Ouahman. 2016. FairAccess: a new Blockchain-based access control framework for the Internet of Things. *Security and Communication Networks* (2016).

[52] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. 2015. VC3: Trustworthy data analytics in the cloud using SGX. In *IEEE S&P*.

[53] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. 2017. Membership inference attacks against machine learning models. In *IEEE S&P*.

[54] Congzheng Song, Thomas Ristenpart, and Vitaly Shmatikov. 2017. Machine learning models that remember too much. In *CCS*.

[55] Pramod Subramanyan, Rohit Sinha, Ilia Lebedev, Srinivas Devadas, and Sanjit A Seshia. 2017. A formal foundation for secure remote execution of enclaves. In *CCS*.

[56] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel {SGX} Kingdom with Transient Out-of-Order Execution. In *USENIX Security*.

[57] Marten Van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. 2010. Fully homomorphic encryption over the integers. In *EUROCRYPT*.

[58] Rolf H Weber and Evelyne Studer. 2016. Cybersecurity in the Internet of Things: Legal aspects. *Computer Law & Security Review* (2016).

[59] Johannes Winter. 2008. Trusted computing building blocks for embedded linux-based ARM trustzone platforms. In *ACM STC*.

[60] Fan Zhang, Ethan Cecchetti, Kyle Croman, Ari Juels, and Elaine Shi. 2016. Town crier: An authenticated data feed for smart contracts. In *CCS*.

[61] Guy Zyskind, Oz Nathan, et al. 2015. Decentralizing privacy: Using blockchain to protect personal data. In *SPW*.