SEAL: Capability-Based Access Control for Data-Analytic Scenarios

Hamed Rasifard
CISPA Helmholtz Center for Information Security
hamed.rasifard@cispa.de

Rahul Gopinath
University of Sydney
rahul.gopinath@sydney.edu.au

Michael Backes
CISPA Helmholtz Center for Information Security
director@cispa.de

Hamed Nemati
CISPA Helmholtz Center for Information Security
Stanford University
hnnemati@stanford.edu

ABSTRACT

Data science is the basis for various disciplines in the Big-Data era. Due to the high volume, velocity, and variety of big data, data owners often store their data in data servers. Past few years, many computation techniques have emerged to protect the security and privacy of such shared data while enabling analysis thereon. Hence, access-control systems must provide a fine-grained, multi-layer mechanism to protect data. However, the existing systems and frameworks fail to satisfy all these requirements and resolve the trust issue between data owners and analysts.

In this paper, we propose SEAL as a framework to protect the security and privacy of shared data. SEAL enables computations on shared data while they remain under the complete control of data owners through pre-defined policies. Our framework employs the capability-object model to define flexible access policies. SEAL's access-control system supports delegating and revoking access privileges and other access-control customizations. In addition, SEAL can assign security labels to privacy-sensitive data and track them to enable data owners to define where and when a data analyst can access their data. We demonstrate the practicability of our approach by presenting a prototype implementation of SEAL. Furthermore, we display the flexibility of our framework by implementing multiple data-analytic scenarios, which cover different applications.

CCS CONCEPTS

• **Security and privacy → Access control**; • **Information systems → Process control systems**; *Computing platforms.*

KEYWORDS

capability-based access control; secure data-sharing framework

ACM Reference Format:
Hamed Rasifard, Rahul Gopinath, Michael Backes, and Hamed Nemati. 2023. SEAL: Capability-Based Access Control for Data-Analytic Scenarios. In *Proceedings of the 28th ACM Symposium on Access Control Models and Technologies (SACMAT '23), June 7–9, 2023, Trento, Italy.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3589608.3593838

SACMAT '23, June 7–9, 2023, Trento, Italy

ACM ISBN 979-8-4007-0173-3/23/06…$15.00
https://doi.org/10.1145/3589608.3593838

1 INTRODUCTION

Data science is the basis for various disciplines across academia and industry, such as health-care, banking, insurance, social networks, and e-commerce [23]. As data science becomes prevalent, academia and industry have recognized several security and privacy issues concerning sharing data between data *owners* and data *analysts*, which limit the possibility of data sharing [30, 45, 55].

Data owners desire to cooperate with analysts to discover hidden insights from their data. Due to the high *volume*, *velocity*, and *variety* of big data, they often have to store their data in data servers and let analysts access them [48]. However, security experts control access at the edge of servers' networks, enabling adversaries to access sensitive data by penetrating servers' networks [22, 59].

Despite analysts obtaining data legitimately, they may violate data privacy or use them for another purpose than the allowed one. For example, previous works [6, 51] demonstrated how attackers could leverage shared genomic data to identify individuals or perform facial recognition. One solution to the problem is to bring computation to data instead of bringing data to computation [1, 7, 35]. Nevertheless, current frameworks need to be more flexible. For instance, data owners must provide data and computations [5, 7]. Furthermore, data owners cannot specify which analysts can invoke which computation or when analysts can access computed results.

Existing Solutions. Over the past many years, there have been a plethora of approaches to address concerns associated with sharing sensitive data, each with different pros and cons. These approaches fall into four main categories: *privacy-preserving access control* [11], *data anonymization* [33, 54], *differential privacy* (DP) [13, 14], and *homomorphic encryption* [19, 38]. Differential privacy [14] is the current de facto standard to protect privacy-sensitive data. This technique provides provable guarantees which programming languages [17, 31, 34] as well as privacy-preserving [18, 37, 46], and query-based [24, 39] frameworks have used to ensure data privacy. For example, the programming platform *PINQ* [34] provides DP primitives to perform privacy-preserving analyses. It trusts developers do not write malicious programs, and requires that program developers rewrite existing applications to meet privacy requirements. Similarly, DPella [31] enables data analysts to reason about the required privacy and accuracy trade-offs. Nevertheless, it only

supports a few statistical analysis functions, such as counting. Finally, GUPT [37] is a DP-based framework that permits analysts to run unmodified programs, and preserves data privacy using the the sample-and-aggregate technique. However, it does not allow analysts to use the result of one computation as the input for another computation.

Despite their usefulness, existing approaches fail to resolve the *trust* issue between data owners and analysts and provide fine-grained access control over computations by analysts. This deficiency implies that data owners must balance the security and privacy risks of lending their data with the benefits of previously overlooked insights. For example, PINQ [34] does not presume analysts act as adversarial. Finally, data-publishing systems, e.g, [60], usually presume that analysts will use published data for pre-defined purposes, which is inherently dangerous.

Our Approach. In this paper, we address these challenges by introducing an access-control system for data-analytic computations. We present SEAL, a computation framework that serves as a middle ground in data-analytic scenarios to resolve this issue. SEAL enables data owners to control the computations that an analyst can invoke, the precondition of an invoked computation, the order of the invocations, and when data can be published to the analyst (see §7). In addition, we designed SEAL to be a general access-control framework for data-analytics scenarios. Therefore, computing architecture can adopt and integrate our framework into their designs.

Our approach ensures two main notions: first, we build a platform that permits performing a sequence of computations on data based on a *stateful access-control scheme*. Our access-control system operates based on the *capability* concept [10, 16, 36], which allows defining flexible policies to control analysts' access to computations and (raw or processed) data. Furthermore, our access-control system supports delegating and revocating user privileges. Moreover, to support data owners in defining and writing access-control policies, we propose a flexible language by extending Rei policy language [25]. Our extension introduces new constructs which facilitate encoding security and privacy requirements in policies.

Second, we build our work upon the critical insight that data sensitivity may evolve during computation steps. For example, the ages of patients are sensitive data in a medical dataset. However, an analyst can access a slated copy of such data (i.e, after applying a sensitization mechanism). Therefore, we enable data owners to define *when* and *where* a data analyst can access their data [47] based on data sensitivity. In doing so, we introduce a labeling mechanism to assign security labels to data.

Security labels are metadata that can be updated either according to applied computations on data or based on directives. Data owners provide directives as function contracts. Such directives are especially helpful in cases where it is unclear how a computation may transform security labels, e.g., a machine learning algorithm. Then, the decision to publish a computed result or to invoke a specific sanitization mechanism depends on the data's current security label and analysts' capabilities, which define their clearance levels. To make this process tractable, we leverage the standard *taint-tracking* technique [50, 52, 53], which tracks the sensitivity level of the data throughout applied computations on the data.

SEAL provides a generic access-control mechanism, which data owners can employ in different data-analytic scenarios. For example, data owners may employ the framework to enforce their desired privacy requirements on their data. SEAL facilitates it by allowing data owners to express their requirements as policies, e.g., they can specify under which circumstances SEAL can publish computed results. We demonstrate the usefulness and flexibility of our proposed framework for covering different applications by using it to implement a few data-analyses scenarios(§6).

To summarize, we make the following main contributions:

- We propose, to our knowledge, the first stateful access-control mechanism based on capabilities for data-analytic scenarios.
- We design a policy language to define capability-based access-control policies and requirements based on Rei.
- we implement our approach into SEAL, a multi-purpose capability-based access-control framework for data owners.
- We evaluate the efficacy of SEAL in real-world scenarios, showing how it protects the security and privacy of shared datasets.

2 SEAL OVERVIEW

The main goal of SEAL is to enable data owners to define how analysts can process their data as a set of processing steps. In addition, SEAL enables them to define when the result of a processing step is publishable.

We refer to a processing step as an *action*. A set of consecutive actions is then called a *processing path*. This approach permits data owners to assign different access privileges to analysts based on the actions and status of the data. SEAL provides a policy language to enable data owners to express their desires as a policy set. Moreover, the framework uses a capability-based access-control model to enforce policies.

Figure 1 depicts the main components of SEAL and the connections between them. SEAL operates in two phases: (1) an *initialization* phase, wherein the data owner initializes the framework and provides the datasets (steps A-D); and (2) an *execution* phase, wherein the framework receives requests from analysts and performs the actual computations (steps 1-9).

2.1 Initialization Phase

In this phase, the data owner prepares the framework to perform requested computations. This phase consists of four steps as follows.

Policy definition. The data owner defines the processing paths as a policy set using our policy language (step Ⓐ). The data owner expresses the actions and states for each path, along with the states where the framework can publish a computed result. The *policy manager* loads the policy set and checks analysts' requests against it. In addition, it updates the status of data after executing an action. §4.4 presents details of the policy definition.

Parameter setting. In this step, the data owner assigns initial values to parameters, such as the dataset's *privacy level* or the *privacy budget* for analysts when DP is used (step Ⓑ). The policy manager uses these parameters while enforcing the policies.

Rights delegation. The data owner assigns rights to analysts to determine their clearance level and the actions they can perform on each path (step Ⓒ). The *capability manager* encodes analyst rights

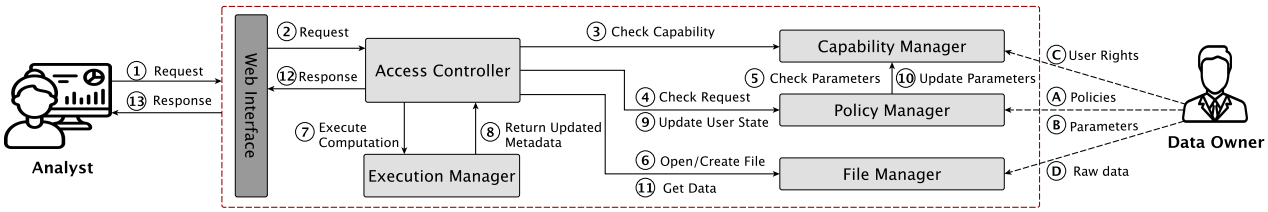


Figure 1: SEAL Overview.

into capabilities and manages them according to our capability model (see §4.2).

Data transfer. The data owner uploads the raw data to the framework (step Ⓓ). The *file manager* stores the uploaded raw data into the appropriate files. Furthermore, the file manager provides input/output file handlers for every action. When the framework invoke an action, the action reads the input data through the input file handler(s) and writes back the result via the output file handler(s).

2.2 Execution Phase

In this phase, analysts request their desired actions. SEAL receives requests from analysts and returns the computed results through a *web interface*. The web interface provides basic functionalities such as a graphical user interface (GUI) and secure communication. Additionally, it acts as a proxy between analysts and the rest of the framework by relaying requests and displaying responses to analysts. The *access controller* intercepts a request when the web interface forwards it. Upon receiving the request, it verifies that (1) the analyst has the appropriate access rights and (2) the execution of the requested action will not violate policies. The access controller achieves these goals in three steps:

Validity check. First, the access controller asks for capability checking from the capability manager (step ③ in Figure 1). If the analyst owns the required capability, the access controller asks the policy manager to check the request against the policy set (steps ④ and ⑤). Then, the access controller obtains the name of the requested computation, its parameters, and a list of metadata, such as the taint of the input data, from the policy manager. In addition, it requests the input and output file handlers from the file manager (step ⑥). The access controller passes all these parameters to the *execution manager*, who performs the actual computation (step ⑦). If any of these checks fails, the access controller stops going further and sends an error message to the web interface.

Computation. The execution manager performs the requested computation in an isolated environment. In addition, it tracks metadata, such as data taints, and updates them according to the performed computation (§4.3 presents the details of our taint-tracking approach). SEAL handles communications with the outside world through input/output file handlers. Communication through the file handlers eliminates the possibility of leaking sensitive data. When the computation terminates and writes the result to output file(s), the computation manager notifies the access controller and forwards the updated metadata to it (step ⑧).

State transformation. The policy manager updates the analyst's state as soon as it receives the notification from the access controller (step ⑨). Furthermore, if needed, the policy manager updates the analyst's capability-related parameters (step ⑩). In the next step, the access controller receives the current state of the analyst and the result of the computation (in the case that the requested action is to publish the data) and sends them to the web interface (steps ⑪ and ⑫). Finally, the analyst receives the response to the request through the web interface (step ⑬).

3 PRELIMINARIES

In this section, we explain the requisite preliminaries to understand the details of our proposed approach.

3.1 Capability-Object Models

SEAL leverages the capability-based access-control model to support fine-grained control over data-processing steps. The capability-based access control provides a robust and flexible mechanism to enforce the *least-privilege* principle [10, 16].

Miller et al. [36] introduced the capability-object model as a security measure for controlling access to particular system parts. The capability-object model models the system resources and subjects as objects. A capability is an unforgeable token and contains a reference to specific objects. When an object possesses a capability, it can communicate with the referenced objects inside the capability. In addition, capabilities encode access rights to determine allowed interactions between objects.

The capability-object model enables two objects that do not possess a direct connection to communicate with each other through intermediate objects. Such mediated communication prevents objects from having direct access to each other and facilitates the revocation operation. For example, suppose three objects exist: A, B, and C, as depicted in Figure 2. Object A possesses a capability that refers to C and desires to delegate the capability to B. To this end, (1) object A creates two intermediate objects, namely R and F; (2) It delegates the capability that refers to C to R; (3) It delegates the capability that refers to R to F. This enables B to communicate with C through these intermediate objects. A asks object R to stop forwarding B's messages to break this communication link. Miller refers to R and F as *revoking* and *forwarding* facets, respectively. We follow the Miller's approach to design our capability-based access-control system.

3.2 Capsicum

To enforce the least-privilege principle for computations running on behalf of data analysts, we use Capsicum [58], an OS-level capability and sandboxing framework. Capsicum adds new primitives to the UNIX API to support the compartmentalization of user-space applications. It constrains the available namespaces an application can access and heavily restricts its permissions. A Capsicum capability

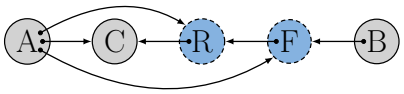


Figure 2: The Capability-Object Model.
Legend: F: Forwarding Facet; R: Revoking Facet.

provides a wrapper around a file descriptor. User programs can not change these capabilities because file descriptors are unforgeable and read-only tokens.

Note that Capsicum capabilities differ from the capabilities we introduce in our framework. A Capsicum capability restricts the access of a process to specific namespaces and system calls. In contrast, the framework's capability defines an analyst's clearance level and the list of actions the analyst can invoke.

3.3 Differential privacy

Differential privacy is a powerful tool for quantifying and solving practical problems related to data privacy. It is a technique which is designed for the setting where there is a **trusted data curator**, which gathers data from individual users, processes the data in a way that satisfies DP, and then publishes the results. Intuitively, the DP notion requires that any single element in a dataset has only a limited impact on the output.

Definition 3.1 ((ϵ, δ)-Differential Privacy). An algorithm satisfies (ϵ, δ)-differential privacy ((ϵ, δ)-DP), where $\epsilon > 0, \delta \geq 0$, if and only if for any two neighboring datasets D and D', we have:

$$for\ all\ T \subseteq Range() :\ \Pr(D) \in T \leq e^{\epsilon}\ \Pr(D') \in T + \delta,$$

where $Range()$ denotes the set of all possible outputs of .

Two datasets D and D' are *neighbors*, denoted as $D \simeq D'$, if and only if either $D = D' + r$ or $D' = D + r$, in where $D + r$ denotes the generated dataset from adding the record r to the dataset D.

3.4 Thread Model

We assume analysts act as adversaries and try to breach the security of our framework and data privacy. We trust the framework's hosting machine, and assume the connections between the framework and analysts are secured using existing mechanisms, such as TLS. We assume the machine's OS prevents network-based attacks. Furthermore, We trust Capsicum [58] and assume adversaries cannot bypass its sandboxing mechanism. We assume adversaries cannot physically access the framework's hosting machine. Moreover, adversaries cannot alter or forge new capabilities; they can only request permitted computations in their capabilities. However, adversaries can upload malicious programs into the framework when they possess proper capabilities.

Adversaries can attack data privacy by requesting trained models or feeding their datasets during computations. We follow Nasr et al. [40] approach and divide adversaries into three categories regarding data privacy, including *weak*, *medium*, and *strong* adversaries. Weak adversaries can only train models and employ models to evaluate their data. Medium adversaries have the capabilities of weak adversaries and request models. Strong adversaries have the above capabilities and can apply their datasets during training models.

4 SEAL DESIGN

In this section, we present details of our framework design. In particular, §4.1 describes how we model data-processing steps as a finite state machine. §4.2 elaborates on our capability model, and §4.3 presents the details of our taint tracking. Finally, §4.4 presents the notion of security policies.

4.1 Stateful System Model

We employ standard structures from automaton theory to model our system. In addition, we apply this technique in a way that incorporates capabilities. In order to treat a wide variety of scenarios, we define an abstract notion of the system. We provide a set of theoretic models as a generalized automaton. We refer to the set as the *capability system*. In addition, the capability system includes a capability machine, which keeps track of analysts permissions.

Let Val and Var denote the set of values and variables, respectively. We define a *state* of our system $s \in S$ as a tuple $\langle \sigma, \Gamma, \Xi, pc \rangle \in S$, where $\sigma : Val \cup Var \mapsto Val$ is a set of local stores that maps locals to values. A local store represents a snapshot of data in a specific state. Γ is a security-relevant variable, and it refers to the taint context $\Gamma : Val \cup Var \mapsto T$ of data being processed, with T being the set of security labels. It tracks security types associated with locals. Ξ denotes the system stack, containing a frame (fr) for each function call. A frame (pc, σ) is a pair of callers' next step program counter and its saved stores. Finally, pc represents the program counter.

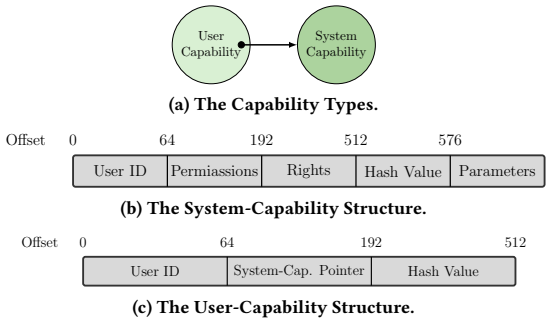
We model the transition as a function $\delta(s_i, a, p)$, wherein s_i is the initial state, a represents an action, and p denotes a security parameter determining when the framework can execute the operation. Function δ yields the reachable state s_j or halts if the operation is not allowed. For example, the capability system can enforce restrictions by consulting a user's capabilities and the current taint of the data. We will get back to this in §4.4. An analyst's request triggers transitions $req = (a \in \Sigma, c \in C, parameters)$, which expresses their desired action and capability, and the parameters for this action.

We model our system as a *deterministic finite-state machine* by adding an initial state. More formally, we model our system as a tuple of $(S, \Sigma, C, \delta, s_0, F)$. S and Σ are finite set of states and action symbols, respectively. C is a non-empty set of capabilities, and $\delta : S \times \Sigma \hookrightarrow_p S \cup \{\perp\}$ is the transition relation with $\hookrightarrow_p$ denoting a transition restricted by policy checking. Finally, $s_0 \in S$ is the initial state, and $F \subseteq S$ is a set of possible final states.

Policies are restrictions over possible transitions in the system, and the system can enforce them through the transition function δ. The primary purpose of policies in our system is to define permitted information flows or sequences of actions. Our transition function consults such policies (denoted by p) for each request. The policy-checking function maps security-relevant parameters, e.g., analyst capabilities and security label of data, to a boolean value determining if the requested operation is allowed, i.e., $p : C \times \Gamma \rightarrow \{true, false\}$.

4.2 Capability Model

We build SEAL's capability model based the *capability-object model* presented in §3.1. It comprises two capability types: *system capability* and *user capability*, as depicted in Figure 3a. The system and



(a) The Capability Types.

(b) The System-Capability Structure.

(c) The User-Capability Structure.

Figure 3: The Capability Model

user capabilities are revoking and forwarding facets in our model, respectively. The capability manager is responsible for creating all the system capabilities and their corresponding user capabilities. While the capability manager keeps system capabilities inside the framework, it hands over user capabilities to analysts.

4.2.1 System Capability. The capability manager creates a system capability (Figure 3b) when a data owner assigns a set of rights to an analyst. It comprises several fields, such as rights, permissions, the corresponding user-capability digest, and parameters.

Permissions. Permissions are possible operations on a capability, such as *delegation* and *revocation*. The delegation permission allows the capability owner to delegate a subset of their rights to another analyst. The revocation permission indicates that the owner can revoke a delegated capability. By revoking a capability, the capability manager revokes all its re-delegation too. Furthermore, SEAL can control the number of re-delegation of a capability

Rights. This list defines the rights that an analyst can request. When creating a capability, the capability manager initializes the list with the provided rights by the data owner for the analyst.

Parameters. This field carries analyst-specific data and helps the policy manager to enforce policies and make decisions. For example, it can specify the maximum number of times the analyst can use the capability.

4.2.2 User Capability. A user capability (Figure 3c) comprises a user identifier, a pointer to the corresponding system capability, and a hash field. The user identifier field binds a capability to a specific analyst. Hence,an analyst cannot obtain and use other analysts' capabilities on their behalf. SEAL can quickly locate the corresponding system capability and perform the required checks because the pointer points to a unique system capability.

The capability manager stores the hash value of the first two fields in the hash field. Afterward, it generates the hash of the whole user capability and stores it inside the hash field of the system capability. The hash guarantees that an analyst cannot maliciously alter a capability. The capability manager stores the latter hash value before releasing the capability to the analyst.

4.3 Security Labels Tracking

Our framework supports more fine-grained access policies in addition to providing capabilities. SEAL provides security labeling of data and tracking these labels throughout requested computations. We leverage these security labels to better control where and when a computation result is publishable, or the framework can use it as the input for the next operation.

SEAL updates security labels (i.e., downgraded or upgraded) either automatically according to applied computations on the data or based on directives stated as function contracts, e.g., by the data owner. Such directives are especially helpful when it is unclear how the applied computations will transform security labels.

SEAL tracks both direct and indirect propagation of data taints. Our tainting rules are standard [50], in which assignments and conditional branches propagate taints to the involved variables. Araujo and Hamlen [2] introduced *flow-sensitive* taint tracking for LLVM intermediate representation with memory pointers. Compared to their approach, our tainting approach is also *context-sensitive*, meaning that we incorporate taint of the current context (i.e., taint due to control flow) into the taint tracking rules.

Figure 4 presents tainting rules for variable assignment, conditional branch, memory store (e.g., array assignment), function call, and return statements. The statement $\textbf{store}(e_1, e_2)$ stores the expression e_2 into the memory location pointed by the expression e_1, and the others statements have a standard meaning. An expression is either a constant $n \in \mathit{Val}$, a variable $r \in \mathit{Var}$, or a standard binary operation ranged over by **bop** (i.e., e_1 **bop** e_2). We keep track of the security type associated with locals by the taint context $\Gamma : \mathit{Val} \cup \mathit{Var} \mapsto \{\mathbf{L}, \mathbf{H}\}$, where $\mathbf{L}, \mathbf{H}$ are the security label of *low* and *high* privacy-sensitive data, respectively. Each local is assigned a security type $\ell \in \{\mathbf{L}, \mathbf{H}\}$, where $(\{\mathbf{L}, \mathbf{H}\}, \sqcup, \sqsubseteq)$ is a lattice of security levels and $\mathbf{L} \sqsubseteq \mathbf{H}$. Moreover, the program counter (pc) taint $\ell_{pc} = \Gamma(pc)$ indicates the current security context. Furthermore, $[e \mapsto e']$ is applied to indicate the update of state components. We use $e \Downarrow_{\langle\, \sigma,\Gamma \,\rangle} (n, \ell)$ to indicate the evaluation of the expression e under the given locals and taint states. The evaluation function represents a large-step transition and yields a pair of values n and taint ℓ for the expression e.

Tainting rules for assignment, branching, and storing instructions are straightforward. We compute the taints of the input expressions for a function call with a formal list of parameters $\overline{params_f}$, such as $\textbf{call}\, f(e_1, \ldots, e_m)$ and update the taints accordingly. In addition, we store the current context in the stack. When the invoked function terminates (rule **ret** e), we compute the taint of returned value by incorporating the security label of the current context (i.e., pc taint) and return to the callee site by restoring the saved context from the stack.

4.4 Security Policies

SEAL processes requests and transforms system states when an action takes place based on policies. In addition, policies specify when SEAL can grant (or revoke) analysts' access. Therefore, the framework requires a simple policy language that is expressive enough to define system states and their relations, required preconditions to invoke actions, and the effects of each action on the system. Moreover, the language should enable data owners to define possible rights of analysts and conditions under which the framework can grant them. To this end, we choose the *Rei* policy language [25] and adapt it to meet our needs.

Rei is a policy language based on deontic logic [57] with constructs for policy objects, including rights, prohibitions, obligations,

$$\frac{e \Downarrow_{\langle \sigma, \Gamma \rangle} (n, \ell) \quad \sigma' = \sigma[r \mapsto n] \quad \Gamma' = \Gamma[r \mapsto \ell \sqcup \ell_{pc}]}{\langle \sigma, \Gamma, \Xi, pc \rangle \rightarrow \langle \sigma', \Gamma', \Xi, pc + 1 \rangle} \; (r := e)$$

$$\frac{e \Downarrow_{\langle \sigma, \Gamma \rangle} (n, \ell) \quad \Gamma' = \Gamma[pc \mapsto \ell_{pc} \sqcup \ell] \quad e_{(n?1:2)} \Downarrow_{\langle \sigma, \Gamma' \rangle} (n', \ell')}{\langle \sigma, \Gamma, \Xi, pc \rangle \rightarrow \langle \sigma, \Gamma', \Xi, n' \rangle} \; (\textbf{if } e \textbf{ then } e_1 \textbf{ else } e_2)$$

$$\frac{e_1 \Downarrow_{\langle \sigma, \Gamma \rangle} (n_1, \ell_1) \quad e_2 \Downarrow_{\langle \sigma, \Gamma \rangle} (n_2, \ell_2) \quad \sigma' = [n_1 \mapsto n_2] \quad \Gamma' = \Gamma[n_1 \mapsto \ell_1 \sqcup \ell_2 \sqcup \ell_{pc}]}{\langle \sigma, \Gamma, \Xi, pc \rangle \rightarrow \langle \sigma', \Gamma', \Xi, pc + 1 \rangle} \; (\textbf{store}(e_1, e_2))$$

$$\frac{\begin{array}{c} e_1 \dots e_m \Downarrow_{\langle \sigma, \Gamma \rangle} (n_1 \dots n_m, \ell_1 \dots \ell_m) \quad \Gamma' = \Gamma[\overline{params_f} \mapsto \ell_1 \dots \ell_m] \\ \sigma' = \sigma[\overline{params_f} \mapsto n_1 \dots n_m] \quad fr = (pc + 1, \sigma) \end{array}}{\langle \sigma, \Gamma, \Xi, pc \rangle \rightarrow \langle \sigma', \Gamma', fr :: \Xi, pc_f \rangle} \; (\textbf{call } f(e_1, \dots, e_m))$$

$$\frac{\begin{array}{c} e \Downarrow_{\langle \sigma, \Gamma \rangle} (n, \ell) \quad fr = (pc', \sigma') \quad \sigma'' = \sigma'[r_{ret} \mapsto n] \\ \Gamma' = \Gamma[r_{ret} \mapsto \ell \sqcup \ell_{pc'}] \end{array}}{\langle \sigma, \Gamma, fr :: \Xi, pc \rangle \rightarrow \langle \sigma'', \Gamma', \Xi, pc' \rangle} \; (\textbf{ret } e)$$

Figure 4: Dynamic taint-tracking rules. In these rules, $\overline{params_f}$ denotes the formal parameters of the function definition.

and dispensations. We extend the language by introducing new constructs: `Action` and `StateObject`. An `action` definition comprises the action's name, the corresponding computation's name, its parameters, and action's requirements. The action construct enables data owners to define new computations. Furthermore, the `StateObject` construct enables data owners to define the states of their systems.

```
ACTION(action-name, computation-name,
  Paramset(paramset-name,
    params(param(param-name, param-type), ...)),
  Require(action-requirements)
```

- **Action and computation names.** The action-name specifies a unique identifier for the action. The computation-name denotes the specific computation implementation which SEAL executes when an analysts requests the associated action.
- **Parameters.** An action takes a set of parameters provided by an analyst as the input. Each parameter has a name and a type, where the type (param-type) includes: String, Integer, Double, Listkv_String, Listkv_Integer, Listkv_Double, and File. Type Listkv_VType defines a list of (key, value), where the VType specifies the values type. For example, the type Listkv_Integer defines a list of (*key*, *integer*) pairs with *integer* as the value's type.
- **Requirements.** The action-requirements refers to a list of requirements for an action. A requirement illustrates what action needs from the framework during its execution. For example, an action may request SEAL to provide dynamic taint tracking during its execution.

The `RIGHT` construct can define a right as follows:

```
RIGHT(right-name, action-name,
  StateObject(Src_State),
  StateObject(Dst_State),
  Obligation(right-conditions))
```

- **Right and action names.** The right-name defines a unique identifier for the right. The action-name specifies the corresponding action that the right owner can request.
- **Obligation.** The action-conditions refers to a list of conditions the data owner asks SEAL to check before executing an action. The data owner can define the list based on the properties of the data or metadata.

SEAL provides two particular actions: *Go_To* and *Publish_Data*. The *Go_To* action changes the current state of an analyst without any requirement. The *Publish_Data* action releases the data to an analyst. Additionally, we provide a few *predicate*s that data owners can employ to define conditions and obligations. For example, they can check if a metadata contains a specific value using metadata(k, v), wherein k represents the metadata's name, and v defines the value. Predicates are logical expressions and may connect to other predicates with logical connectives, including *Conjunction (AND)*, *Disjunction (OR)*, and *Negation (NOT)*.

5 SEAL IMPLEMENTATION

We implemented SEAL as a hosted platform, i.e. a web server, using *Flask* [41] toolkit and *Python* programming language. Analysts may connect to the SEAL as clients and request their desired computations while providing input parameters. Upon receiving a request, the web server sends the request along with the corresponding parameters and the user's capability to the access controller. Furthermore, analysts can upload their capabilities and view their current states in the finite state machine.

The access controller orchestrates a sequence of operations to produce the response: (1) The access controller asks the capability manager to check the analyst's capability. (2) It asks the policy manager to check the request against the policy set. (3) The access controller receives the file handlers for the requested data from the file manager. (4) It passes the action name, parameters, and file handlers to the execution manager. (5) The access controller asks the policy manager to update the analyst's state. In the following, we explain the implementation details of each component.

Capability Manager: The capability manager handles all capability-related operations, such as creating a new capability. Figure 3b illustrates the structure of a system's capability. The capability manager

uses a tree structure to store system capabilities. The tree comprises a *root* capability which includes all rights and permissions in the policy set and several delegated capabilities. This capability belongs to the data owner. The capability manager creates a delegated capability for each data analyst, in which the actions and permissions are a subset of the actions and permissions of the delegated capability's parent.

The capability manager handles delegating or revoking capabilities. It stores the delegated capabilities of a system capability in a doubly linked list as the children of the capability. The parent capability provides access to the head and tail of the list through two pointers. The capability manager delegates a capability by adding a child node at the end of this list. It revokes a capability by removing it from the list. It is worth noting that the tree's height can be greater than one, i.e., analysts can re-delegate their delegated capabilities by possessing the permission. Finally, we leverage *bcrypt* [43] to generate the user- and system-capability digests and use these digest for security reasons.

To check a capability, the capability manager first verifies the integrity of the capability by regenerating the hash values and comparing them with the hash-value field inside the system capabilities (Figure 3b). Then, it verifies the analyst's rights by searching for the requested right inside the rights field (Figure 3b). Finally, the capability manager informs the access controller about the result of these checks.

Policy Manager: The manager reads the specified policies from a description file during the initialization phase. It then loads the policies as a state machine into an internal data structure comprising *state objects*. A state object contains its unique name and several *action objects*. Each action object contains information related to an action, such as the name of the action and its requirement.

The policy manager stores analysts' information, such as their current state, in a database using the *LiteSQL* library. When the policy manager receives an analyst's request for a right, it retrieves the action and source state of the right (presented in §4.4) and the current state of the analyst. Then, the policy manager checks for this request: (1) equality of the states; (2) existence of the action inside the right and action in the current state; (3) fulfillment of the action's requirements; and (4) the current privacy budget of the analyst from the system capability (Figure 3b). Finally, the policy manager informs the access controller about the result of these checks.

Execution Manager. The execution manager provides an interface for receiving information about requested computations. The access controller delegates executing requested computations to the execution manager through the interface. During the execution, a computation must only access to provided data. Therefore, the framework executes the requested computation inside a Capsicum [58] sandbox.

The execution manager provides a taint-tracking tool to instrument the code of the computation function if the policy requires it. The transformed code handles both implicit and explicit taint tracking during an execution. The execution manager informs the access controller about the computation's success and its output's taint when it terminates. In the following, we explain how the taint-tracking tool accomplishes its task.

5.1 Taint Tracking

There are two main problems in the implementing taint-tracking mechanism for used libraries in data-analytic scenarios: (1) data-analytic libraries mix code snippets from different languages like python, **C**, and assembly for performance reasons; (2) they usually have huge code bases.

To resolve the first challenge, we employ Numba [27] to transfer a Python library into the LLVM [28] intermediate representation (IR). The transferring allows us to use off-the-shelf taint trackers, e.g., PhASAR [49], for static taint tracking. Furthermore, we combine static and dynamic taint tracking to increase the performance for large code bases. In particular, we use static taint tracking to analyze the code of the used function from data-analytic libraries (e.g., *NumPy* [56] and *Scikit-learn* [42]) while collecting data for dynamic taint tracking of the code used by an analyst for computation.

5.2 Dynamic Taint Tracking

Implementing a full-fledged taint-tracking module is out of the scope of this paper. However, the current implementation of SEAL can handle both direct and indirect information flow, memory operations and arrays, and function calls.

We support taint tracking without modifying the Python's interpreter. For this reason, we implement our dynamic taint tracking in two parts. The first part relies on object proxies for direct taint propagation through the data flow. The second part statically instruments the source code to keep track of indirect taint propagation due to control flow. Now, we describe these two parts.

5.2.1 Direct taint tracking through object proxies. For propagating taints based on direct data flow, we employ the described technique by Conti et al. [8]. This technique leverages wrapper objects for any input through the *taint sources*. Each wrapper object acts as a proxy for the object it wraps. Furthermore, the source marks a wrapper object with a taint.

The wrapper object employs the original (wrapped) object to obtain the result of an invocation of its methods. Afterward, it wraps the result in a new wrapper object. This technique computes the taint of the new wrapper based on the original wrapper's taint, the arguments' taints, and the context taint (discussed next). Hence, the wrapper object propagates the taint along the result. A key reason for choosing this technique is its ability to propagate direct taints without modifying the interpreter.

5.2.2 Indirect taint tracking through instrumentation. We leverage source instrumentation to incorporate indirect taint propagation due to control flow. We implement the indirect taint propagation during the control flow as a context-taint variable in the resulting branches of the control flow.

In Python, each control flow has a conditional expression that specifies which branch to take. We first evaluate the conditional expression. If the evaluation produces a tainted wrapper object, the context taint variable is set to the generated taint.

Contrary, the context-taint variable is set to *no taint* if the evaluation does not produce a tainted wrapper. The framework passes the context-taint variable to any operation that involves a tainted wrapper object. We implement this procedure by rewriting the program's

abstract syntax tree (AST) under taint tracking to incorporate the context variable.

6 CASE STUDY

SEAL's design suits various data analysis scenarios. Data owners can adapt the framework according to their applications. Concretely, a scenario may involve private data. Data owners can employ different privacy-preserving techniques, such as differential privacy, to ensure the framework will publish sanitized data to analysts.

Differential privacy guarantees that an adversary should not be able to distinguish between outputs of computation over two datasets that differ only in an individual record. Dwork [14] defines the ϵ-differential privacy as follows: For any two datasets $S, S' \in D^n$ differing only in one data record, computation C preserves ϵ-differential privacy if for any $R \subseteq Range(C)$

$$\Pr[C(S) \in R] \leq e^{\epsilon} \times \Pr[C(S') \in R]$$

Where ϵ is the *privacy budget*. A lower privacy budget indicates that computation C preserves privacy at a higher level. In comparison, a higher privacy budget means the computation preserves privacy at a lower level but produces more accurate results. Due to space constraints, we select four typical data-analytic scenarios that demonstrate the capabilities of SEAL. These scenarios leverage the differential-privacy technique to preserve data privacy and cover the cases in which data owners and analysts provide the input data.

Figure 5 illustrates our four analysis scenarios. We logically divide these scenarios into two phases: the *selection phase* and the *computation phase*. In the selection phase, analysts can only select a subset of data without seeing it. Afterward, analysts can leverage SEAL to analyze the selected data in the computation phase. After defining the policies, the data owner specifies a privacy level by setting a privacy budget for the whole dataset. In addition, the data owner defines analysts' rights and their maximum privacy budgets.

6.1 Scenario One: Statistical Analysis

The first scenario (dash-dotted lines in Figure 5) investigates how analysts can select a subset of data records and count them. Counting the selected data can provide analysts insight into the data or help them to select the subsequent adequate request. This scenario demonstrates how SEAL can provide statistical queries/operations.

As Figure 5 illustrates, analysts can invoke the `Count` action to count the selected data. However, they can observe the result if they first invoke the `Add_Noise` action. Action `Add_Noise` applies an appropriate DP technique to preserve data privacy. Concretely, it adds a *Laplacian noise* to the counted value with a sensitivity set to 1 [29].

6.2 Scenario Two: Model Training

The second scenario (bold lines in Figure 5) trains a machine-learning model. The data owner applies the following constraints in this scenario:

- Analysts can train their models using these datasets, with the restriction of maintaining the privacy level of the data.
- analysts can select the data from different sources, including their data.
- analysts can either have access to the trained model for querying new data or get the trained model itself.

The data owner first defines the `Linear_Regression` action to realize the machine-learning scenario. This action takes the assigned privacy budget and the hyperparameter of the model and then employs the *diffprivlib* library [21] to train a differential private linear-regression model. An analyst can publish the trained model using the `Publish_Model` action or test it using their data by invoking the `Test_with_Analyst_Data` action. Furthermore, analysts can combine the selected data with their data before model training by performing the `Get_Analyst_Data` action.

The data owner has multiple options for preserving privacy in this scenario. For instance, the data owner can provide the theoretical guarantees of DP for the whole pipeline by defining a strict privacy budget which the framework will reduce from the privacy budget of an analyst after invoking each action. However, there is a less restricted option in which data privacy can depend on empirical guarantees.

Recent work [40] empirically shows that achievable privacy highly depends on what an adversary can perform, i.e., how strong the adversary is. The data owner can define privacy budgets based on adversaries' types and strengths. For example, the data owner can define the following adversaries, ranked from the weakest to the strongest.

(1) The weakest adversary can only train the model and evaluate their data on it.
(2) The medium adversary can request the model itself, in addition to the ability of the weakest adversary.
(3) The strongest adversary can do all of the above and apply their data in the model's training.

SEAL supports these scenarios by enabling data owners to define appropriate rights and the maximum privacy budget for each type of adversary.

6.3 Scenario Three: Analyst Functions

The third scenario (dashed lines in Figure 5) demonstrates how data owners let analysts process data with their programs. Data analysts sometimes develop private functions and desire to evaluate data using them. However, data owners intend to provide this facility for only some analysts. For example, they may only want a subset of internal analysts to process data with their programs.

Data owners define such policy as follows: First, SEAL receives the functions from these users; Then, it uses the sample-and-aggregation technique [37] to execute the functions while preserving the data privacy. In this case, analysts can neither include their data nor use the results as inputs to further steps.

6.4 Scenario Four: Model Training with Taint Tracking

Finally, the fourth scenario (bold dashed-lines in Figure 5) describes how data owners can leverage the provided taint-tracking mechanism. This scenario is an extreme example in which SEAL tracks every bit of data during a computation to demonstrate the capabilities of our framework.

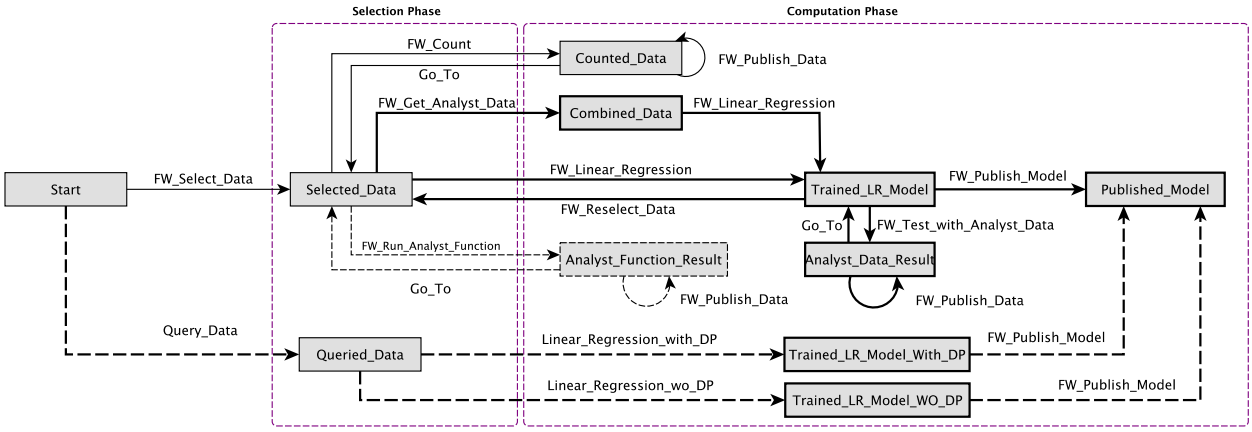


Figure 5: Data-analytic Scenarios.

In this scenario, data owners allow a subset of analysts to select any subset of the data and perform *linear-regression* model training over it. Afterward, based on the taint of the data, the data owner can decide when to publish the results.

When an analyst selects a subset of data, the execution manager activates the taint-tracking mechanism. When an analyst requests SEAL to publish the model, the policy manager checks the request against the policy set and the taint values. It only publishes the model if this check passes.

Use-case policy. The following policy demonstrates this scenario using our policy language. The data owner provides three actions and their corresponding function name in this scenario. These actions include selecting a subset of the data, executing the linear-regression model training with differential privacy, and executing the linear-regression model training without differential privacy. For example, the data owner defines the action for selecting a subset of data as follows:

```
ACTION(Query_Data, query_data_function,
  Paramset(query_data_parameters,
  params(param(any-of-these, Listkv_String),
         param(all-of-these, Listkv_String))),
  Require(taint-tracking)
```

where the taint-tracking mechanism is enabled for the action. The data owner defines analysts' rights based on actions and the data's taint. For example, an analyst can only train a model without applying differential privacy when the data taint is *Low* as follows:

```
RIGHT(LR_Learning_wo_DP, Linear_Regression_wo_DP,
  StateObject(Queried_Data),
  StateObject(Trained_LR_Model_WO_DP)
  Obligation(data − tain = Low))
```

6.5 Security and Privacy Analysis

SEAL is a hosted platform that interacts with untrusted and potentially malicious users. In doing so, it is essential to restrict analysts' privileges and ensure that they cannot directly access datasets and resources beyond their access rights.

SEAL performs access control at two levels: *analyst level* and *computation level*. To control analysts' access, SEAL employs the object-capability model. The object-capability model allows data owners to define fine-grained privileges per analyst. Moreover, our framework prohibits an analyst from misusing the capabilities of other analysts. For this reason, the capability manager binds a capability to an analyst, thus preventing unauthorized use of the capability. In addition, the capability manager supports capability revocation through system capabilities.

The framework employs Capsicum to isolate invoked computations by analysts. Such isolation enforces the least-privilege principle and protects datasets and system resources. Capsicum allows computations only access to a minimal set of system calls and prohibits the rest. For example, It forbids computations to access the network sockets. Thus, if an adversary provides a malicious code, Capsicum restricts it from accessing the rest of the framework and leaking data via network or file system inside a sandbox. In §7, we evaluate the overhead of Capsicum.

Data owners can preserve data privacy by applying appropriate sanitization techniques before publishing them. In addition, the framework enables data owners to define various restrictions based on use cases or how strong an adversary is. For example, consider the above machine learning scenarios. The data owner assigns a privacy budget to a dataset during the initialization phase, which the framework keeps inside the root capability. Furthermore, The data owner assigns the maximum privacy budget that an analyst can spend, which the framework keeps inside the system capability of the analyst. When a request arrives, Seal first checks the dataset's privacy budget and the analyst's budget. If the remaining budget of the dataset or the budget of the analyst does not suffice, SEAL will reject the action. Otherwise, SEAL will execute the action and decrease the action's budget from the budgets of the dataset and

the analyst. Due to the sequential composability of differential privacy, the final privacy budget of any dataset will never exceed the assigned one.

7 EVALUATION

SEAL adds runtime overhead when it processes a request. The framework introduces this overhead due to checking the analyst's capability, checking the request against the policy set, and running the computation inside the Capsicum's sandbox. Hence, we evaluate the performance of our framework by comparing its execution time for each scenario with the execution time of the same scenario in the native form. By native form, we refer to executing actions by calling them through the Python interpreter. We employ *cProfiler*, a Python profiler, to measure the performance, which gathers statistics about a function call, such as its execution time. To provide accurate results, we execute each measurement five times and average their runtimes as the final result of the measurement.

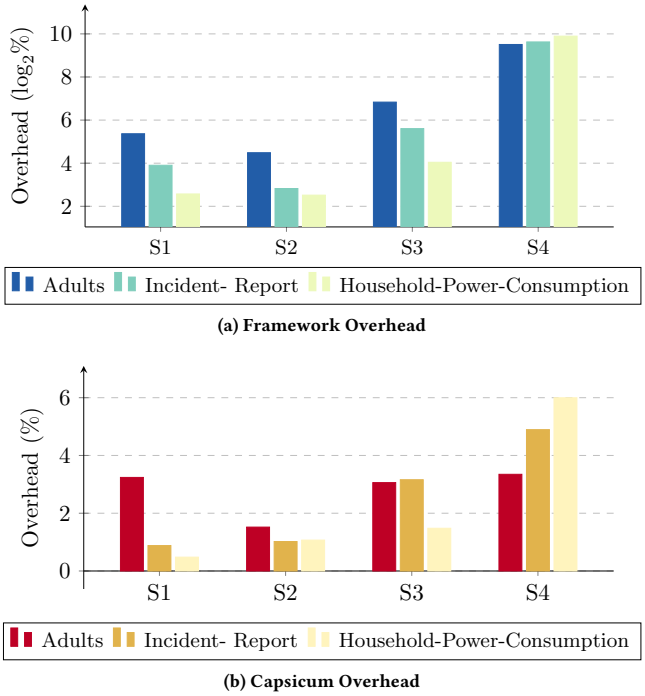
We first selected three datasets from *UCI Machine-Learning Repository* [12]: the *Adult* dataset, the *Incident-Report* dataset, and the *Household-Power-Consumption* (HPC) dataset. The datasets contain 32, 561, 141, 713, and 2, 075, 258 data entries, respectively. We repeat all the measurements for these three datasets.

We measure the execution time of the four scenarios (Table 1) to evaluate our framework's overhead. The execution time of a scenario starts when the framework receives the first request from an analyst and ends when the framework publishes the result to the analyst. Furthermore, we measure the overhead of Capsicum's sandboxing for in our framework. Hence, we disable the sandboxing and repeat the measurement for each scenario. Figure 6a and Figure 6b illustrate the framework's and Capsicum's overheads, respectively.

In the first scenario, the analyst selects all data entries, asks the framework to count them, and observes the result. Figure 6a illustrates the overhead of the framework for all three datasets. The framework's overhead for the Adult dataset is 41.27%, the Incident-Report dataset is14.98%, and the Household-Power-Consumption dataset is 5.95%. These overheads demonstrates that in this scenario, an overhead decreases as the number of data entries increases.

In the second scenario, the analyst selects all the data entries in a dataset. Then, the analyst provides the parameters and starts training the model. Finally, the analyst asks the framework to publish the model. The framework's overhead for the Adult dataset is 22.43%, the Incident-Report dataset is 7.09%, and the Household-Power-Consumption dataset is 5.73%, as Figure 6a depicts. Moreover, the framework's overhead decreases when data entries increase.

In the third scenario, the analyst selects all the data entries in a dataset, provides a computation function, and asks the framework to apply the function over the selected data. For the sake of simplicity, the user-provided function just computes the average of the data in a specific column for each dataset. As Figure 6a illustrates, the same pattern repeats similar to two previous scenarios. However, the produced overhead by this scenario is higher than the produced overheads for two previous scenarios. This higher overhead is due to the sample-and-aggregate technique. The framework divides a dataset to several smaller datasets based on this techniques. Then, the framework creates a child process for each smaller dataset, runs



(a) Framework Overhead

(b) Capsicum Overhead

Figure 6: Computation Overhead.

the computations inside the created processes, and aggregates the results. Hence, the creation of child processes and aggregating of their results produces a higher overhead compared to the overheads of two previous scenarios.

In the forth scenario, we divide the dataset into three sub-datasets. One sub-dataset includes data tagged with *High* taints, and the other two include data with *Low* taints. First, the analyst selects all data from three sub-datasets. In the next step, since the selected datasets include data with *High* taints, the analyst can only request the linear-regression model training with differential privacy.

The framework's overhead varies between 7 to 10 times the native execution times in the fourth scenario because SEAL tracks every bit of data during the computation. An execution time is related to the number of function calls during its execution. The number of function calls depends on the number of times a computation retrieves an object from a container data structure. Every time a computation retrieves an object from a container, the taint-tracking mechanism creates a tainted type of object, then initiates the new object with the value of the retrieved object and the taint of the container. Furthermore, whenever the computation combines several objects, the taint-tracking mechanism merges them into a container and updates its taint based on the objects' taints. There are a few optimizations to reduce the number of function calls, such as minimizing the number of object retrievals from a container.

8 RELATED WORK

We identified four main categories of work related to our approach in this paper and reviewed selected works in each category.

Programming languages and frameworks. PINQ [34] provides DP building blocks for programmers to write privacy-preserving

	Adult Dataset			*Incident Report*			*Household Power Consumption*		
		SEAL			*SEAL*			*SEAL*	
	native	*with Campsicum*	*without Capsicum*	*native*	*with Campsicum*	*without Capsicum*	*native*	*with Campsicum*	*without Capsicum*
● *S1 : Statistiacal Analysis*	1.284	1.814	1.757	6.067	6.976	6.915	88.077	93.324	92.87
● *S2 : Model Training*	10.151	12.428	12.241	80.916	86.654	85.779	496.548	525.01	519.45
● *S3 : Analyst Functions*	1.299	2.789	2.706	7.372	10.96	10.624	108.189	126.74	124.89
● *S4 : Model Training with TaintTracking*	10.151	84.1	81.375	81.051	721.973	688.286	497.381	5 012.714	4 729.055

Table 1: Execution Time (Second)

programs. Fuzz [20] and DFuzz [17] provide programming languages that ensure the return value of a query includes noise. HOARe^2 [4] introduces a programming language and uses program verification to provide differential privacy. DPella [31] proposes a programming framework with compositional reasoning about the accuracy of data. These frameworks do not address the trust issue between data owners and analysts. However, our framework does not trust analysts and programmers. Hence, it provides a mechanism to resolve the trust issue between them.

Privacy-preserving frameworks. Airavat [46] preserves data privacy and only executes unmodified Map-Reduce programs. GUPT [37] provides a framework that allows analysts to run unmodified programs. The GUPT framework employs the sample-and-aggregate technique to preserve data privacy. Privacy-preserving frameworks do not enable data owners to define compound computations. However, our framework enables data owners to express such compound computations in various orders for analysts.

Query-based frameworks. Djoin [39] defines a few join operations for distributed databases and applies differential privacy to their results. Flex [24] applies elastic sensitivity to provide data privacy in the results of SQL queries. *Apex* [18] proposes a framework that considers data accuracy and privacy. Although existing query-based frameworks and data-publishing systems preserve data privacy when publishing data, an analyst can use the published data for a different purpose rather than the proposed one. However, our framework brings computations to data. Hence, analysts can only receive data in the defined states by data owners.

Privacy-preserving data publishing. Rappor [15] introduces a mechanism to collect distributed end-user data and provide privacy-preserved statistical results using differential privacy. These frameworks preserve data privacy but do not provide any mechanism to control how analysts will use them. However, our framework enables data owners to define how their data should be processed and when a data should be published.

Policy Languages. *Platform for privacy preference* (P3P) [44], *P3P preference exchange language* (APPEL) [9] , and *enterprise privacy authorization language* (EPAL) [3] provide standards and tools for expressing privacy policies. However, they do not specify how the policies should be enforced. Lou et al. [32] introduced a framework for preserving privacy in distributed data-analytic scenarios. On the contrary, our framework can be employed to control access to computations that work with various kinds of data.

9 CONCLUSION AND FUTURE WORK

Summary and conclusions. In this paper, we proposed a capability-based access-control system for data-analytic scenarios. Our stateful system lets data owners control how analysts analyze their data by defining fine-grained data control policies.

Our approach enables complex computations on shared data while keeping them under the complete control of the data owners through access-control policies. We further suggest labeling privacy-sensitive data and tracking their transformation based on performed computations, which enable data owners to make more informed decisions on when, where, and who can access a result.

To facilitate defining policies, we extend the Rei policy language with the constructs which allow encoding security and privacy requirements. We implemented our approach into a prototype framework, that we called SEAL. To evaluating our scheme's efficacy and analysing its security and privacy guarantees, we tested SEAL on three real-world data-analytic scenarios, and measured its performance overhead in each scenario. Our results showed that SEAL can enhance the security and privacy of shared datasets.

Future work. In this work, we implemented a proof-of-concept system for our approach on a single server. However, we designed our approach with flexibility in mind, making it possible to adapt to other architectures. For example, disaggregated-memory architectures, such as *Memory-Driven Computing* [26], can expedite the data-processing speed and efficiency, and we plan to adapt SEAL's implementation to support these architectures in the future.

ACKNOWLEDGMENTS

This work was supported in part by the German Federal Ministry of Education and Research (BMBF) through funding for the CISPA-Stanford Center for Cybersecurity (FKZ: 13N1S0762). We would like thank Ahmed Ibrahim and Ahmed Salem for reading the draft of the paper and providing us with suggestions to improve the text.

REFERENCES

[1] Ryan P Abernathey, Tom Augspurger, Anderson Banihirwe, Charles C Blackmon-Luca, Timothy J Crone, Chelle L Gentemann, Joseph J Hamman, Naomi Henderson, Chiara Lepore, Theo A McCaie, et al. 2021. Cloud-native repositories for big scientific data. *Computing in Science & Engineering* 23, 2 (2021).

[2] Frederico Araujo, W Kevin, et al. 2015. Compiler-instrumented, dynamic secret-redaction of legacy processes for attacker deception. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*. 145–159.

[3] Paul Ashley, Satoshi Hada, Günter Karjoth, Calvin Powers, and Matthias Schunter. 2003. Enterprise privacy authorization language (EPAL). *IBM Research* 30 (2003), 31.

[4] Gilles Barthe, Marco Gaboardi, Emilio Jesús Gallego Arias, Justin Hsu, Aaron Roth, and Pierre-Yves Strub. 2015. Higher-order approximate relational refinement types for mechanism design and differential privacy. *ACM SIGPLAN Notices* 50, 1 (2015), 55–68.

[5] Peter Baumann, Andreas Dehmel, Paula Furtado, Roland Ritsch, and Norbert Widmann. 1998. The multidimensional database system RasDaMan. In *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*. 575–577.

[6] Luca Bonomi, Yingxiang Huang, and Lucila Ohno-Machado. 2020. Privacy challenges and research opportunities for genomic data sharing. *Nature genetics* 52, 7 (2020), 646–654.

[7] Paul G Brown. 2010. Overview of SciDB: large scale array storage, processing and analysis. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 963–968.

[8] Juan José Conti and Alejandro Russo. 2010. A taint mode for Python via a library. In *Nordic Conference on Secure IT Systems*. Springer, 210–222.

[9] Lorrie Cranor. 2002. A P3P preference exchange language 1.0 (APPEL1. 0). *http://www. w3c. org/TR/P3P-preferences. html* (2002).

[10] Jack B Dennis and Earl C Van Horn. 1966. Programming semantics for multiprogrammed computations. *Commun. ACM* 9, 3 (1966), 143–155.

[11] David DeWitt. 2004. Limiting disclosure in hippocratic databases. In *30th Int. Conf. on Very Large Databases, VLDB Endowment, Toronto, Canada*. 108–119.

[12] Dua, Dheeru and Graff, Casey. 2017. UCI Machine Learning Repository. http://archive.ics.uci.edu/ml.

[13] Cynthia Dwork, Krishnaram Kenthapadi, Frank McSherry, Ilya Mironov, and Moni Naor. 2006. Our data, ourselves: Privacy via distributed noise generation. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 486–503.

[14] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. 2006. Calibrating noise to sensitivity in private data analysis. In *Theory of cryptography conference*. Springer, 265–284.

[15] Úlfar Erlingsson, Vasyl Pihur, and Aleksandra Korolova. 2014. Rappor: Randomized aggregatable privacy-preserving ordinal response. In *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*. 1054–1067.

[16] Robert S. Fabry. 1974. Capability-based addressing. *Commun. ACM* 17, 7 (1974), 403–412.

[17] Marco Gaboardi, Andreas Haeberlen, Justin Hsu, Arjun Narayan, and Benjamin C Pierce. 2013. Linear dependent types for differential privacy. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 357–370.

[18] Chang Ge, Xi He, Ihab F Ilyas, and Ashwin Machanavajjhala. 2019. Apex: Accuracy-aware differentially private data exploration. In *Proceedings of the 2019 International Conference on Management of Data*. 177–194.

[19] Craig Gentry. 2009. Fully homomorphic encryption using ideal lattices. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*. 169–178.

[20] Andreas Haeberlen, Benjamin C Pierce, and Arjun Narayan. 2011. Differential Privacy Under Fire.. In *USENIX Security Symposium*, Vol. 33.

[21] Naoise Holohan, Stefano Braghin, Pól Mac Aonghusa, and Killian Levacher. 2019. Diffprivlib: the IBM differential privacy library. *arXiv preprint arXiv:1907.02444* (2019).

[22] Vincent C Hu, Tim Grance, David F Ferraiolo, and D Rick Kuhn. 2014. An access control scheme for big data processing. In *10th IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing*. IEEE, 1–7.

[23] Valentina Janev, Dea Pujić, Marko Jelić, and Maria-Esther Vidal. 2020. *Chapter 9 Survey on Big Data Applications*. Springer International Publishing, Cham, 149–164. https://doi.org/10.1007/978-3-030-53199-7_9

[24] Noah Johnson, Joseph P Near, and Dawn Song. 2018. Towards practical differential privacy for SQL queries. *Proceedings of the VLDB Endowment* 11, 5 (2018), 526–539.

[25] Lalana Kagal, Tim Finin, and Anupam Joshi. 2003. A policy language for a pervasive computing environment. In *Proceedings POLICY 2003. IEEE 4th International Workshop on Policies for Distributed Systems and Networks*. IEEE, 63–74.

[26] Kimberly Keeton. 2017. Memory-Driven Computing. USENIX Association, Santa Clara, CA.

[27] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. 2015. Numba: A LLVM-Based Python JIT Compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC* (Austin, Texas) *(LLVM '15)*. Association for Computing Machinery, New York, NY, USA, Article 7, 6 pages. https://doi.org/10.1145/2833157.2833162

[28] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 75–86.

[29] Ninghui Li, Min Lyu, Dong Su, and Weining Yang. 2016. *Differential Privacy: From Theory to Practice*. Morgan Claypool.

[30] Christoph Lippert, Riccardo Sabatini, M Cyrus Maher, Eun Yong Kang, Seunghak Lee, Okan Arikan, Alena Harley, Axel Bernal, Peter Garst, Victor Lavrenko, et al. 2017. Identification of individuals by trait prediction using whole-genome sequencing data. *Proceedings of the National Academy of Sciences* 114, 38 (2017), 10166–10171.

[31] Elisabet Lobo-Vesga, Alejandro Russo, and Marco Gaboardi. 2020. A programming framework for differential privacy with accuracy concentration bounds. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 411–428.

[32] Chen Luo, Fei He, Fei Peng, Dong Yan, Dan Zhang, and Xin Zhou. 2019. PSpec-SQL: Enabling Fine-Grained Control for Distributed Data Analytics. *IEEE Transactions on Dependable and Secure Computing* (2019).

[33] Ashwin Machanavajjhala, Daniel Kifer, Johannes Gehrke, and Muthuramakrishnan Venkitasubramaniam. 2007. l-diversity: Privacy beyond k-anonymity. *ACM Transactions on Knowledge Discovery from Data (TKDD)* 1, 1 (2007), 3–es.

[34] Frank D McSherry. 2009. Privacy integrated queries: an extensible platform for privacy-preserving data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. ACM, 19–30.

[35] Dmitry Medvedev, Gerard Lemson, and Mike Rippin. 2016. Sciserver compute: Bringing analysis close to the data. In *Proceedings of the 28th international conference on scientific and statistical database management*. 1–4.

[36] Mark S Miller, Ka-Ping Yee, Jonathan Shapiro, et al. 2003. *Capability myths demolished*. Technical Report. Technical Report SRL2003-02, Johns Hopkins University Systems Research Laboratory.

[37] Prashanth Mohan, Abhradeep Thakurta, Elaine Shi, Dawn Song, and David Culler. 2012. GUPT: privacy preserving data analysis made easy. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM, 349–360.

[38] Payman Mohassel and Yupeng Zhang. 2017. Secureml: A system for scalable privacy-preserving machine learning. In *2017 IEEE symposium on security and privacy (SP)*. IEEE, 19–38.

[39] Arjun Narayan and Andreas Haeberlen. 2012. DJoin: Differentially private join queries over distributed databases. In *10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*. 149–162.

[40] Milad Nasr, Shuang Song, Abhradeep Thakurta, Nicolas Papernot, and Nicholas Carlini. 2021. Adversary Instantiation: Lower Bounds for Differentially Private Machine Learning. In *2021 IEEE Symposium on Security and Privacy (SP)*. 866–882. https://doi.org/10.1109/SP40001.2021.00069

[41] Pallets Organization. 2023 (accessed April 25, 2023). Flask Web Development Framework. https://flask.palletsprojects.com.

[42] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. 2011. Scikit-learn: Machine learning in Python. *the Journal of machine Learning research* 12 (2011), 2825–2830.

[43] Niels Provos and David Mazieres. 1999. Bcrypt algorithm. In *USENIX*.

[44] Joseph Reagle and Lorrie Faith Cranor. 1999. The platform for privacy preferences. *Commun. ACM* 42, 2 (1999), 48–49.

[45] Maria Rigaki and Sebastian Garcia. 2021. A Survey of Privacy Attacks in Machine Learning. arXiv:2007.07646 [cs.CR]

[46] Indrajit Roy, Srinath TV Setty, Ann Kilzer, Vitaly Shmatikov, and Emmett Witchel. 2010. Airavat: Security and privacy for MapReduce.. In *NSDI*, Vol. 10. 297–312.

[47] Andrei Sabelfeld and David Sands. 2005. Dimensions and principles of declassification. In *18th IEEE Computer Security Foundations Workshop (CSFW'05)*. IEEE, 255–269.

[48] Seref Sagiroglu and Duygu Sinanc. 2013. Big data: A review. In *2013 international conference on collaboration technologies and systems (CTS)*. IEEE, 42–47.

[49] Philipp Dominik Schubert, Ben Hermann, and Eric Bodden. 2019. PhASAR: An Inter-procedural Static Analysis Framework for C/C++. In *Tools and Algorithms for the Construction and Analysis of Systems*, Tomáš Vojnar and Lijun Zhang (Eds.). Springer International Publishing, Cham, 393–410.

[50] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *2010 IEEE symposium on Security and privacy*. IEEE, 317–331.

[51] Dzemila Sero, Arslan Zaidi, Jiarui Li, Julie D White, Tomás B González Zarzar, Mary L Marazita, Seth M Weinberg, Paul Suetens, Dirk Vandermeulen, Jennifer K Wagner, et al. 2019. Facial recognition from DNA using face-to-DNA classifiers. *Nature communications* 10, 1 (2019), 1–12.

[52] Paritosh Shroff, Scott Smith, and Mark Thober. 2007. Dynamic dependency monitoring to secure information flow. In *20th IEEE Computer Security Foundations Symposium (CSF'07)*. IEEE, 203–217.

[53] G Edward Suh, Jae W Lee, David Zhang, and Srinivas Devadas. 2004. Secure program execution via dynamic information flow tracking. *ACM Sigplan Notices* 39, 11 (2004), 85–96.

[54] Latanya Sweeney. 2002. k-anonymity: A model for protecting privacy. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* 10, 05 (2002), 557–570.

[55] Bernhard Tellenbach, Marc Rennhard, and Remo Schweizer. 2019. *Security of Data Science and Data Science for Security*. Springer International Publishing, Cham, 265–288. https://doi.org/10.1007/978-3-030-11821-1_15

[56] Stefan Van Der Walt, S Chris Colbert, and Gael Varoquaux. 2011. The NumPy array: a structure for efficient numerical computation. *Computing in science & engineering* 13, 2 (2011), 22–30.

[57] Georg Henrik Von Wright. 1951. Deontic logic. *Mind* 60, 237 (1951), 1–15.

[58] Robert NM Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. 2010. Capsicum: Practical Capabilities for UNIX.. In *USENIX Security Symposium*, Vol. 46. 2.

[59] Zettaset. 2014. The Big Data Security Gap: Protecting the Hadoop Cluster. http://fs.fish.govt.nz/Page.aspx?pk=7&sc=SUR. White Paper; accessed 2 March 2023.

[60] Jun Zhang, Graham Cormode, Cecilia M Procopiuc, Divesh Srivastava, and Xiaokui Xiao. 2017. Privbayes: Private data release via bayesian networks. *ACM Transactions on Database Systems (TODS)* 42, 4 (2017), 1–41.