PICACHV: Formally Verified Data Use Policy Enforcement for Secure Data Analytics

Haobin Hiroki Chen[1], Hongbo Chen[1], Mingshen Sun[2], Chenghong Wang[1], and XiaoFeng Wang[3*]

[1]*Indiana University Bloomington*
[2]*Independent Researcher*
[3]*Nanyang Technological University*

Abstract

Ensuring the proper use of sensitive data in analytics under complex privacy policies is an increasingly critical challenge. Many existing approaches lack portability, verifiability, and scalability across diverse data processing frameworks. We introduce PICACHV, a novel security monitor that automatically enforces data use policies. It works on relational algebra as an abstraction for program semantics, enabling policy enforcement on query plans generated by programs during execution. This approach simplifies analysis across diverse analytical operations and supports various front-end query languages. By formalizing both data use policies and relational algebra semantics in Coq, we prove that PICACHV correctly enforces policies. PICACHV also leverages Trusted Execution Environments (TEEs) to enhance trust in runtime, providing provable policy compliance to stakeholders that the analytical tasks comply with their data use policies. We integrated PICACHV into `Polars`, a state-of-the-art data analytics framework, and evaluate its performance using the TPC-H benchmark. We also apply our approach to real-world use cases. Our work demonstrates the practical application of formal methods in securing data analytics, addressing key challenges.

1 Introduction

The rapid advancement of computational devices and the rise of big data present unparalleled opportunities to accelerate scientific progress and drive innovation through data-driven decision-making. While these opportunities are ground-breaking, they are accompanied by a critical challenge: ensuring proper data *use* in compliance with complex privacy policies. Real-world data analytics often involves complex privacy policies. For instance, researchers using NIH's *All of Us* [44] platform must comply with multiple privacy regulations, such as the HIPAA Safe Harbor Rule [43] and the platform's own policies mandating patient data aggregation into groups of 20 or more.

Unfortunately, enforcing these policies poses a great challenge. Manual checks are impractical due to the complexity of analytical tasks and the high cost of human effort. This leads to reliance on machine-based solutions, which often fail to address these challenges comprehensively. For example, database access control mechanisms [41, 42, 48, 57], focus solely on restricting access to data but do not ensure that *already authorized personnel* comply with proper data use policies. Moreover, they lack verifiable guarantees to data holders as they frequently rely on very *ad hoc* heuristics. Program analysis [13, 17, 27, 34, 47, 51–53] mostly still focus on access control policies. The few that can be or have been extended to data use policy enforcement often restrict programmers to a specific language (e.g., PRIVGUARD [51] only supports Python), which is less desirable.

Addressing this problem remains non-trivial. One major challenge is the diversity of front-end programming languages. In data analytics, researchers and data scientists use a wide range of languages and rely heavily on third-party libraries. Some frameworks like Apache Spark [55] supports both Dataframe-like APIs and SQL. This diversity in frontend programming languages and tools creates significant challenges for implementing consistent and universal policy enforcement mechanisms. Each language has its own syntax, semantics, and idiosyncrasies. This variability complicates the process of policy enforcement, as a mechanism tailored for one language may not be applicable to others. Moreover, complex third-party libraries often obscure underlying data operations, making policy enforcement more challenging. Traditional approaches that focus on language-specific static or dynamic analysis techniques [17, 27, 34, 52, 53] struggle to provide a comprehensive solution across this diverse landscape.

Enforcing data use policies essentially relies on *information flow control* (IFC) [32, 48], with the underexplored aspect of *declassification* playing a key role. Existing works on declassification [19, 35, 46] primarily focus on language-level approaches and provide only general methodologies for specifying policies, but they do not address the specific requirements of our use case. Here, several key challenges emerge: 1)

[*] Previously with Indiana University Bloomington.

accurately describing data sensitivity levels and downgrading operations for data analytics, 2) proposing new semantics for the relational algebra that interplays with the declassification requirements, and then 3) defining and proving a rigorous *semantics-based security model* [45] for policy enforcement. Addressing these design complexities requires formal verification to provide mathematically rigorous security guarantees, which are critical for highly sensitive workloads.

Beyond security guarantees, effective policy enforcement must also account for performance trade-offs and system scalability. For analytical queries, performance (or total cost of ownership (TCO)) is always a critical concern in policy enforcement, with both static and dynamic approaches facing unique challenges. While static analysis minimizes runtime overhead, it introduces significant development costs [12] and lacks the precision (i.e., high false positive rate) achievable with dynamic analysis. Dynamic analysis, on the other hand, grapples with the significant runtime complexity of tracking tags across data, and the tags quickly become cumbersome and unmanageable as the system scales up. These performance considerations necessitate careful balancing of enforcement strategies.

This work. We tackle these challenges from a different perspective: relational algebra. Data analysis programs, regardless of their implementation language or framework, can be logically described using relational algebra or similar intermediate representations. Such translations have already been well-studied [24, 28, 50], there are even extensions for other applications like ML inferences [30]. Hence, we assume the existence of such techniques and focus our research on designing enforcement mechanisms over relational algebra (often in the form of query plans). We therefore describe all data manipulations in a unified query plan, overcoming the limitations imposed by specific languages or frameworks as we obtain the semantics of the program via relational algebra. Furthermore, this insight aligns seamlessly with current policy requirements, which specify necessary operations before release, as these policies inherently describe *data manipulation requirements* that can be integrated into relational algebra.

While policy enforcement at this level seems straightforward, there is a question left unsolved: How should we structure the security lattice and declassification rules to express diverse data use policies? Existing works on declassification policies [12, 19, 35] provide only high-level methodologies and are not directly applicable to our scenarios. Fortunately, data use policies inherently imply sensitivity ordering through manipulation requirements. For example, privacy policies like redaction or aggregation naturally reduce data sensitivity to some extent, giving us an intuition for structuring the security lattice in alignment with these data operations. This work presents a feasible way for specifying data use policies based on this key observation. On top of that, we propose in this paper a new operational semantics for relational algebra to support policy enforcement. To ensure correctness, we rigorously verify these semantics using Coq, providing mathematically sound guarantees for policy compliance.

We implement a prototype runtime security monitor, called PICACHV. Informally, PICACHV functions as a verified middleware that intercepts query plans and monitors query execution in parallel to detect policy violations. Since PICACHV focuses on the relational data, it allows for sophisticated optimization techniques such as parallelization. Furthermore, PICACHV integrates with TEEs to provide strong *proofs* to data owners that data is used appropriately during analytical tasks in untrusted cloud environments. TEEs provide cryptographic reports via remote attestation, ensuring the authenticity of the monitor, the integrity of operations, and proving policy compliance of queries to the data owner. Nevertheless, our technique can be applied to trusted environments like on-premise databases.

We summarize our contributions as follows.

- We formalize relational operators with declassification semantics to avoid the complexity of language-based semantics. We formalize the semantics and also provide the mechanized proof of soundness in Coq.
- We design PICACHV, a dynamic security monitor that transparently enforces these data use policies when executing analytical tasks. Our framework also incorporates multiple optimizations, and experiments show that only small runtime overhead was incurred.
- We ported PICACHV to a well-established data analysis framework called `Polars` and performed evaluation to showcase the adaptability of PICACHV in diverse data processing environments.

Roadmap. This paper is structured as follows: In Section 2, we review related work. Section 3 introduces essential background and preliminary knowledge to provide readers with the necessary foundation. In Section 4, we formalize the data use policies central to our approach. Section 5 details the operational semantics of our policy-integrated relational algebra. Section 6 outlines the implementation of PICACHV. Section 7 evaluates PICACHV, demonstrating its effectiveness and efficiency across various scenarios. Finally, Section 8 discusses the scalability, performance trade-offs, and potential limitations of our approach.

2 Related Work

Policy enforcement in data analytics. Several frameworks have been developed to analyze application code that interacts with sensitive data. Language-based information flow control mechanisms, such as STORM [34], RULEKEEPER [23], UrFLOW [17], SELINQ [47], SWIFT [18], PRIVGUARD [51] and JEEVES [53] renforce policies by rejecting non-compliant

code at compile time. Some solutions, however, require manual code annotations, which can be labor-intensive and error-prone. Moreover, these frameworks struggle to enforce fine-grained policies effectively compared to PICACHV. Query rewriting approaches, including QAPLA [42], JACQUELINE [52], and ESTRELA [13], aim to ensure policy compliance by modifying query syntax. However, these methods operate at the syntax level, often sacrificing the precision required for execution-time analysis. The most similar work to PICACHV is LAPUTA [31], which enforces table-level policies for Apache SparkQL using regular expression matching on physical query plans. However, like many earlier efforts, LAPUTA relies heavily on heuristics and lacks verifiable guarantees, limiting its applicability for sensitive workloads. It also lacks fine-grained policy enforcement.

Formal methods. Formal methods are essential for ensuring the correctness of critical systems by providing mathematically rigorous guarantees and eliminating reliance on *ad hoc* testing. Large-scale systems such as operating system kernels and virtual machine managers have successfully adopted formal verification techniques [29,36,37], demonstrating their industrial feasibility. Applying formal methods in securing data analytics remains a rare practice. Existing research has formalized and verified the correctness of database systems [39] and SQL semantics [26], but no work has yet addressed the integration of privacy policies into relational algebra.

TEEs for secure data computation. Confidential Computing enables secure data processing, unlocking new computing scenarios and fostering innovation across industries where privacy concerns previously posed barriers. Under the hood, TEEs protect unauthorized parties from seeing and modifying the data and code inside TEEs based on sophisticated hardware protection mechanisms. Many TEEs have been made available to the public, such as enclave-based Intel SGX [20], and VM-based Intel TDX [16], AMD SEV SNP [49]. There also have been many works that leverage TEEs to build secure data sharing frameworks (e.g., [38]). Industries are actively exploring utilizing TEE-based solutions for secure data computation. For example, Google Research has proposed the *Confidential Federated Computations* using TEE-based ledger systems [22].

3 Preliminary

Threat Model. PICACHV's primary goal is to offer *verifiable proofs* to data owners whose data is analyzed by any individual or group on the cloud inside TEEs that their data can be used only in accordance with the data use policies they specified. We thus assume that the TEE environment is completely safe. Attacks such as hardware side-channels, software vulnerabilities, or other exploits targeting TEEs are outside the scope of this paper and assumed to be mitigated by existing approaches (e.g., [58]). The primary concerns are unintentional or intentional policy violations during data analytics. These may include improper data aggregation, inadequate anonymization, etc.

Relational Algebra. Relational algebra, a cornerstone of database theory, provides a set of operators for manipulating relational data. Relational data is organized into tables, called relations, where each row represents a tuple (a single data entry) and each column represents an attribute (a property of the data). Relations are defined with a schema that specifies the attributes and their domains, ensuring consistency and structure in the data. It forms the theoretical basis for relational database management systems and SQL. The fundamental operators include selection (σ) for filtering rows based on a condition, projection (π) for selecting specific columns, union ($\cup$) for combining tuples from two relations, set difference ($-$) for returning tuples in one relation but not another, Cartesian product ($\times$) for creating all possible tuple combinations from two relations, and the join ($\bowtie$) for concatenating tuples. By combining these operators, complex queries can be constructed to extract and transform data efficiently. Relational algebra's mathematical foundation ensures query precision and optimizability, making it crucial for database design and management, and beyond.

4 Formalizing Data Use Policies

In this section, we present the formalization of privacy policies so that they can accommodate policies written in human natural languages. We first define a security lattice that provides a formal framework for expressing these policies. Then we introduce the integration of data use policies with declassification requirements.

4.1 Sensitivity Labels

In reality, protecting data privacy often involves hierarchical sensitivity levels. For example, a user's full address is more sensitive than their zip code, and their zip code is more sensitive than their county. Similarly, individual purchase records are more sensitive than aggregated sales data. As data undergoes transformations like aggregation or anonymization, its sensitivity typically decreases. We first define sensitivity levels as labels in Definition 4.1.

Definition 4.1 (Security lattice)**.** The security labels for PICACHV are defined as $\mathcal{L} = \{\mathbf{L}, \mathbf{N}, \mathbf{A}, \mathbf{T}, \mathbf{H}\}$ such that $\mathbf{L} \sqsubseteq \mathbf{N} \sqsubseteq \mathbf{A} \sqsubseteq \mathbf{T} \sqsubseteq \mathbf{H}$. The security lattice is defined as $\langle\, \mathcal{L}, \sqsubseteq, \sqcup, \sqcap \,\rangle$, satisfying:

- $\mathbf{L} \sqsubseteq \ell, \forall \ell \in \mathcal{L}$.
- $\ell \sqsubseteq \mathbf{H}, \forall \ell \in \mathcal{L}$.
- $\ell_1 \sqsubseteq \ell_2 \iff \ell_1 \sqcup \ell_2 = \ell_2$.
- $\ell_1 \sqsubseteq \ell_2 \iff \ell_1 \sqcap \ell_2 = \ell_1$.

Note that $\sqsubseteq$ is antisymmetric, meaning that $\ell_1 = \ell_2$ if and only if $\ell_1 \sqsubseteq \ell_2 \wedge \ell_2 \sqsubseteq \ell_1$.

The lattice in Definition 4.1 includes **T** for data transformation requirements, **A** for data aggregation requirements, and **N** for noise addition requirements. This security lattice is motivated by the varying sensitivity levels in data governed by privacy policies. Different data elements and their derivatives possess distinct levels of sensitivity. Unlike other works with only two sensitivity levels (high **H** and low **L**) [15,46,56], our lattice captures nuanced requirements for data use policies. The real-world examples will be detailed later to strengthen our design. We first present the security lattice as follows. These labels themselves solely, however, cannot represent the policy requirements by data owners. We introduce *policies* in the following section.

4.2 Policies

In our framework, each data element (specifically, individual cells within each relation) is associated with a distinct *declassification* policy. This policy specifies sanitization procedures that ensure compliance with the required privacy policies. Upon satisfying all stipulated conditions, the data becomes eligible for disclosure. These declassification policies are fundamentally built upon the previously established base security labels, which collectively form a lattice $\mathcal{L}$.

We have adapted and refined the definition presented in [19] to formalize this concept to align with our specific scenario. The formal definition is as follows:

$$p ::= \mathbf{L} \mid \ell^{O} \rightsquigarrow p, \tag{1}$$

where the lowest label **L** in the lattice indicates that all policies are enforced, allowing the data to be freely disclosed. The expression $\ell^{O} \rightsquigarrow p$ signifies that for a given security label $\ell \in \mathcal{L}$, the data must be used following the security label ℓ, and only operations defined in O can downgrade the data to the next policy p. A subtle consequence here is that operations will now have their own sensitivity levels. For example, a summation operation should have **A** because it is doing an aggregation.

We assume that we can always define a subset relation $\subseteq$ over O, which can be either a vanilla set, or some complex structures like aggregation with group size requirements. For example, for aggregates, we can define $O \mid n$ where n denotes the minimum required group size, and O denotes the set of allowed aggregate operations. Additionally, given that the security lattice is now refined with operations, we present the refined lattice "flows to" relation in Figure 1. This relation is useful for composing policies, which will be detailed later in this section.

We present the declassification rules in Figure 2, modeled as a relation $p \xrightarrow{op,\ell} p'$, where op represents the operation applied to the policy, and ℓ denotes its sensitivity level. When

$$\textsc{Lt}\ \frac{\ell_1 \sqsubseteq \ell_2 \quad \ell_2 \not\sqsubseteq \ell_1}{\ell_1^{O_1} \sqsubseteq^* \ell_2^{O_2}} \qquad \textsc{Lt2}\ \frac{\ell_1 = \ell_2 \quad O_1 \subseteq O_2}{\ell_1^{O_1} \sqsubseteq^* \ell_2^{O_2}}$$

Figure 1: The refined lattice flows-to rules $\boxed{\ell_1^{O_1} \sqsubseteq^* \ell_2^{O_2}}$.

an operation is applied to a policy, two cases arise: a) Operation sensitivity level is *lower* than the policy's sensitivity level: The absence of a corresponding reduction rule in Figure 2 indicates an ill-formed case, resulting in a runtime error. b) Operation sensitivity level is *greater* than or *equal* to the policy's sensitivity level: If the operation can successfully downgrade the policy, the policy is downgraded, as defined by the DLOWER rule. Otherwise, the policy remains unchanged, as the required operation has not yet been applied, following the DPRESERVE rule.

$$\textsc{DEmpty}\ \frac{}{\mathbf{L} \xrightarrow{op,\ell} \mathbf{L}} \qquad \textsc{DLower}\ \frac{p = \ell^{O} \rightsquigarrow p' \quad op \in O}{p \xrightarrow{op,\ell} p'}$$

$$\textsc{DPreserve}\ \frac{p = \ell'^{O} \rightsquigarrow p' \quad \ell' \sqsubseteq \ell \quad op \notin O}{p \xrightarrow{op,\ell} p}$$

Figure 2: The declassification rules $\boxed{p \xrightarrow{op,\ell} p'}$ for policy p.

Well-formed policy. Specifically, because our policy focuses on declassification, policies must be ordered (in the sense of $\rightsquigarrow$) in the descending way. This reflects a practical basis: policies often specify the operations required to make data disclosable rather than to restrict it further. As shown in Figure 3, a policy p is well-formed if it consists of a chain where labels flow from higher to lower sensitivity levels, which can be verified through pattern matching on the refined "flows to" relation (Figure 1).

$$\frac{}{\mathsf{wf}(\mathbf{L})} \qquad \frac{}{\mathsf{wf}(\ell^{O} \rightsquigarrow \mathbf{L})} \qquad \frac{\mathsf{wf}(\ell_2^{O_2} \rightsquigarrow p) \quad \ell_2^{O_2} \sqsubseteq^* \ell_1^{O_1}}{\mathsf{wf}(\ell_1^{O_1} \rightsquigarrow \ell_2^{O_2} \rightsquigarrow p)}$$

Figure 3: The well-formedness property of policy p.

Policy Composition. Policy composition is essential for scenarios where data from multiple sources converge. For instance, consider a medical dataset combining patient data from hospitals located in different states, such as California and Texas, each governed by its distinct data use policies. In cases where researchers aim to analyze the socioeconomic

factors affecting patient health across states, they must adhere to the policies of *both* datasets. This requires a mechanism for policy composition. Figure 4 illustrates rules for a commutative composition process, ensuring policies are consistently merged. These rules ensure that joint analysis adheres to all policy requirements from both parties. Informally, policy join involves constructing a chain by inserting the appropriate policy label into the correct position based on the refined "flows to" relation. For policies sharing the same sensitivity levels in the MERGE case, we apply the intersection operation $\cap$ to obtain the joined label ℓ^{O_3}.

$$\textsc{JBotL}\ \frac{}{\mathbf{L} \uplus p = p} \qquad \textsc{JBotR}\ \frac{}{p \uplus \mathbf{L} = p}$$

$$\textsc{Left}\ \frac{\ell_2^{O_2} \sqsubseteq^* \ell_1^{O_1} \quad p_1 \uplus (\ell_2^{O_2} \rightsquigarrow p_2) = p_3}{\ell_1^{O_1} \rightsquigarrow p_1 \uplus \ell_2^{O_2} \rightsquigarrow p_2 = \ell_1^{O_1} \rightsquigarrow p_3}$$

$$\textsc{Right}\ \frac{\ell_1^{O_1} \sqsubseteq^* \ell_2^{O_2} \quad (\ell_1^{O_1} \rightsquigarrow p_1) \uplus p_2 = p_3}{\ell_1^{O_1} \rightsquigarrow p_1 \uplus \ell_2 \rightsquigarrow p_2 = \ell_2^{O_2} \rightsquigarrow p_3}$$

$$\textsc{Merge}\ \frac{O_1 \cap O_2 = O_3 \quad p_1 \uplus p_2 = p_3}{\ell^{O_1} \rightsquigarrow p_1 \uplus \ell^{O_2} \rightsquigarrow p_2 = \ell^{O_3} \rightsquigarrow p_3}$$

Figure 4: The composition rules for policy p.

4.3 Policy Expressiveness

We argue that the declassification policies presented in this paper provide a practical abstraction, grounded in real-world scenarios. We provide three different privacy policy use cases to strengthen our argument.

Scenario #1: Medical Data Analysis. Medical data analysis serves as a concrete application of our policy design. Data collected by hospitals and medical research institutions is inherently sensitive, necessitating strict privacy regulations. To illustrate the applicability of our policies, we examined two examples: *HIPAA Safe Harbor* [43], a foundational standard for medical privacy, and the *All-of-Us* [44] project initiated by the U.S. National Institutes of Health (NIH) that extends the HIPPA boundary. For instance, under the HIPAA Safe Harbor framework, researchers must redact the last three digits of a zip code. This policy can be expressed as $\mathbf{T}^{\{\text{redact}(3)\}} \rightsquigarrow \mathbf{L}$. The *All-of-Us* project mandates that aggregation must be performed on groups of at least 20 individuals, represented as $\mathbf{A}^{\{\text{sum,avg,max,min...}\}|20} \rightsquigarrow \mathbf{L}$.

Scenario #2: The U.S. Census Bureau. The U.S. Census Bureau's 2020 disclosure avoidance guidance [14] employs differential privacy [21] to add statistical noise to census data, preventing the identification of individuals in small demographic groups. The Census Bureau assigns varying privacy budgets ε to different individuals and census blocks. PICACHV's policy framework captures these requirements by associating $\mathbf{N}^{\Delta} \rightsquigarrow \mathbf{L}$ with the corresponding data objects. In this case, please note that PICACHV leaves the task of specifying the noise to policymakers and does not dictate *what* noise should be added to the corresponding data. Instead, it only ensures that a specific noise Δ is applied to the data.

Scenario #3: Policy for businesses. An additional example illustrates how fine-grained policies can be applied in business contexts. Google provides users with extensive privacy rights, allowing them to control what information is collected and for what purposes[1]. For example, consider a table where Google stores user demographic information. If users opt out of ad recommendations based on their age, a cell-level policy $\mathbf{H}^{\emptyset} \rightsquigarrow \mathbf{L}$ can be specific on the corresponding individuals of the attribute `age`, indicating that their ages can never be used. Users can also specify other requirements like redaction, generalization, etc. that can be further applied on the base policy. If Google's basic policy requires age data to be aggregated for groups of at least 100 people, policies can be "overlaid" to construct $\mathbf{T}^{\{redact\}} \rightsquigarrow \mathbf{A}^{\{\text{sum,avg,}\cdots\}|100} \rightsquigarrow \mathbf{L}$.

5 Formalizing PICACHV

This section formalizes the operational semantics of relational algebra (RA^P), which supports PICACHV's function as a runtime security monitor. The full syntax of relational algebra is defined in Figure 5. We first define an extended relational data model to support policy-associated data.

Data Model. We begin by formalizing the underlying data model. The schema, s is a list of primitive types, $\overline{\tau}$. A tuple t is a dependent type based on the schema, with each element t referred to as a *cell* [39]. A relation R is a collection of tuples. We denote by $R[i]$ the i-th tuple in R, and slicing R along indices $\overline{n}$ as $R \restriction_{\overline{n}} T$. Each cell within a tuple is assigned a unique identifier (id) to enable policy lookup in the given environment detailed later. We visualize this in Figure 6.

Expressions. In our model, expressions take five forms. Two atomic expressions include the primitive value pv, used as constants, and the column identifier $\texttt{col}(id)$, for selecting values from a given column. Complex expressions include unary expressions $\otimes e$, binary expressions $e_1 \oplus e_2$, and aggregates $\mathcal{A}(e)$. Expressions are used as: 1) predicates in `select` for tuple filtering (φ) and `aggregate` (g) for group filtering, or 2) actual values evaluated as results.

Relational algebra. As illustrated in Figure 5, a query q includes a special operator for data retrieval and five fundamental relational operators. A relation R indexed by its identifier $id \in \mathbb{N}$ is denoted as $R(id)$. The union operator $q_1 \cup q_2$ merges two subqueries q_1 and q_2 that share the same schema. The join operator $q_1 \bowtie_{\overline{n_1},\overline{n_2}} q_2$ combines results from two subqueries q_1 and q_2 based on columns specified by two lists of

[1] https://safety.google/intl/en_us/privacy/ads-and-data/#controls

Data model

Type:	τ	$::= \texttt{int} \mid \texttt{str} \mid \texttt{bool} \mid$	Primitive Value:	pv	$\in \texttt{str} \mid \texttt{bool} \mid \texttt{nums}$
Schema:	s	$::= \overline{\tau}$	Unary:	$\otimes$	$::= \neg \mid - \mid + \mid ...$
Tuple:	t	$::= \bot \mid \langle\, c, t \,\rangle$	Binary:	$\oplus$	$::= \geq \mid \leq \mid = \mid \neq ...$
Cell:	c	$::= \langle\, pv, id \,\rangle$	Aggregate:	$\mathcal{A}$	$::= \texttt{sum} \mid \texttt{max} \mid \texttt{min} \mid ...$
Cell Identifier:	id	$\in \mathbb{N}$	Trace:	tr	$::= \overline{\langle\, id, tt \,\rangle}$
Tagged Value:	v	$::= c \mid \overline{c}$	Trace type:	tt	$::= \texttt{TrNone}\ p$
					$\mid \texttt{TrSingle}\ tt, op, p$
					$\mid \texttt{TrMulti}\ \overline{tt}, op, p$

Relational algebra

Query:	q	$::= R(id) \mid q_1 \cup q_2 \mid \pi_{\overline{e}}(q)$	Policy:	p	
		$\mid\ q_1 \bowtie_{\overline{n_1},\overline{n_2}} q_2 \mid \sigma_{\varphi}(q)$	Operator:	op	$::= \oplus \mid \otimes \mid \mathcal{A}$
		$\mid\ \gamma_{\overline{e}\mid\overline{g}\mid\varphi}(q)$	Group:	G	$::= \langle\, t, \overline{n} \,\rangle, n \in \mathbb{N}$
Expression:	e, g, φ	$::= pv \mid \texttt{col}(id) \mid \otimes e$	Data store:	Σ	$::= \overline{\langle\, id, \langle\, R, \Gamma \,\rangle \,\rangle}$
		$\mid\ e_1 \oplus e_2 \mid \mathcal{A}(e)$	Policy store:	Γ	$::= \overline{\langle\, id, p \,\rangle}$

Figure 5: The syntax for RA^{P}.

indices $\overline{n_1}$ and $\overline{n_2}$. The projection operator $\pi_{\overline{e}}(q)$ evaluates expressions $e \in \overline{e}$ on every row in the result of the query q. The selection operator $\sigma_{\varphi}(q)$ filters rows of q that satisfies the predicate φ. The aggregate operator $\gamma_{\overline{e}\mid\overline{g}\mid\varphi}(q)$ is slightly more complex. It computes aggregates $\overline{e}$, groups rows by $\overline{g}$, and filters groups satisfying φ (i.e., the `having` clause). Additionally, it specifies lists of aggregate expressions $\overline{e}$. Here, the group information G includes the active tuple t, which represents the grouped keys (a list of column indices used for grouping), and the group-by indices $\overline{n}$ corresponding to the tuples in R within the current group. For example, consider again the relation shown in Figure 6, where the grouping is performed on the τ_1 column. Hence, the grouping key is τ_1, and there will be two groups whose representative values are v_1 and v_2, respectively. In this case, the first group G_1 would be $\langle\, \langle\, v_1, \bot \,\rangle, \{0,1\} \,\rangle$, and the second one G_2 would be $\langle\, \langle\, v_2, \bot \,\rangle, \{2\} \,\rangle$, assuming that the index of each row starts at 0.

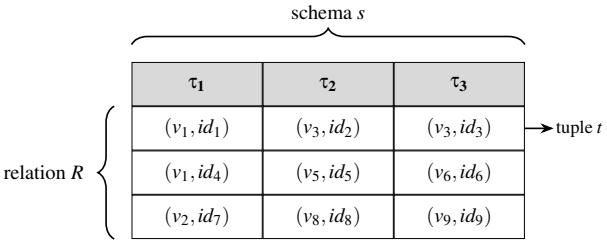


Figure 6: The relational model.

Program trace. The program trace tr serves two purposes when performing expression evaluations which be introduced next. First, it ensures that all data undergoes the appropriate operations before being released by recording the history of operations (op) performed on each cell. Second, it maintains the current policy p associated with the cell being evaluated. Accordingly, tr is represented as a list of tuples containing cell identifiers and their corresponding trace types t. The intuition behind the definition of t is straightforward: within the five relational operators, a cell may undergo one of two types of transformations. These transformations involve either a single data source, as in the case of `TrSingle` caused by `projects`, or multiple data sources, as in `TrMulti`, typically resulting from operations like `aggregates` or `joins`. For cells that have not undergone any operations, their trace type is represented as `TrNone` p, with p as policies associated with the cell.

5.1 Formal Semantics

This section introduces the core reduction rules for RA^{P}, starting with the rules for evaluating expressions in relational algebra, followed by the rules for relational operators.

5.1.1 Expressions

Expressions form the foundation of relational algebra. Expression evaluation is defined as the relation[2]:

$$\langle\, tr, T \,\rangle \xrightarrow{e} \langle\, tr', v \,\rangle,$$

where a trace tr and a tuple context $T ::= t \mid \overline{t}$ (represents either a single tuple t or a list of tuples in an aggregate context)

[2]In Coq, an additional argument for the maximum allowed steps must be introduced to satisfy the termination checker.

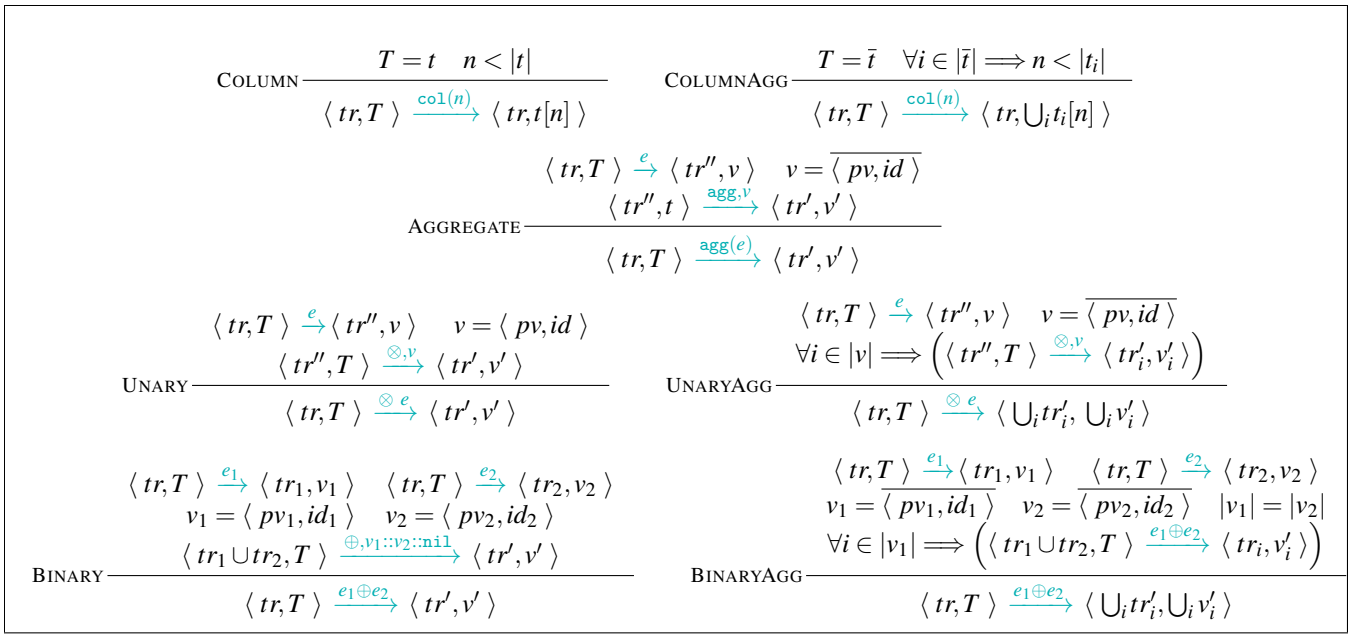


Figure 7: Main expression evaluation rules for $\langle tr, T \rangle \xrightarrow{e} \langle tr', v \rangle$.

are inputs, and the evaluation of the expression e produces a value v along with an updated trace tr'. To highlight expression evaluation that involves policy transformation, we define an auxiliary expression evaluation relation as follows.

$$\langle tr, T \rangle \xrightarrow{f,v} \langle tr', v' \rangle,$$

where f is the function being applied to tagged value v. We present the main expression evaluation rules in Figure 7, where we categorize expression evaluation based on the tuple context T. The COLUMN rule is straightforward, as it simply retrieves the value at the specified index n from the tuple t. For UNARY and BINARY expressions, the rules invoke the corresponding auxiliary evaluation functions. In the aggregate context, the evaluation works similarly but applies the rules iteratively to each element in v.

The auxiliary rules in Figure 8 are more interesting. The FUNARY rule governs unary function application, where the program trace tr must contain the policy p associated with the tagged value v (which includes a unique identifier id). The intended operation f is applied to p, transitioning the label (highlighted) and producing an updated policy p' (if possible as defined in the Figure 2). This change is reflected in the updated program trace as $tr[\langle id, p \rangle \mapsto \langle id, p' \rangle]$. The function f is then interpreted as $[\![f]\!]$ and applied to the primitive value pv carried by v, yielding $[\![f]\!](pv)$. The FBINARY rule follows a similar process but includes an additional constraint: the second argument v_2 must be clean, meaning its policy must be $\mathbf{L}$. The AGGREGATE rule is slightly more complex. Before applying the operation, it ensures that all policies in the list of primitive values are compatible with the aggregate function. It then transitions the labels to obtain each updated policy p'_i. The final result is computed by folding over the value list pvs, with the resulting policy label being the composition of all p'_i. We use `map` to split v into two lists using the pair projection functions `fst` and `snd`: one list for values, pvs, and another for identifiers, ids. It has type:

$$\texttt{map} : (A \to B) \to \overline{A} \to \overline{B},$$

which applies a function $f : A \to B$ on each element of the list $\overline{A}$, producing a list of results $\overline{B}$. Next, the list of primitive values and their identifiers is processed to compute a result using the `fold` operation, which has the type:

$$\texttt{fold} : (B \to A \to B) \to B \to \overline{A} \to B,$$

This operation takes a higher-order function f (which accumulates elements of the list), an identity element B, a list of elements $\overline{A}$, and produces a final result B. Afterward, a new identifier for the result is generated by the `new_id` function[3].

Trusted Blackbox Functions. In the expression evaluation rules defined in Figure 9, we treat functions as *blackboxes* [51] for two primary reasons. First, no limitations are imposed on function implementations, allowing library developers to create functions or user-defined functions (UDFs) in any programming language and style. This flexibility, however, complicates policy enforcement, as it requires not only understanding the semantics of these functions but also supporting

[3]We assume that this function is like a UUID generator so we do not need to worry about id conflicts.

$$\frac{v = \langle\, pv, id \,\rangle \quad \langle\, id, p \,\rangle \in tr \quad p \xrightarrow{f,\mathbf{T}} p' \quad tr' = tr[\langle\, id, p \,\rangle \mapsto \langle\, id, p' \,\rangle]}{\langle\, tr, T \,\rangle \xrightarrow{f,v} \langle\, tr', \langle\, [\![f]\!]\,(pv), id \,\rangle \,\rangle} \text{(FUNARY)}$$

$$\frac{\langle\, id_1, p \,\rangle \in tr \quad \langle\, id_2, \mathbf{L} \,\rangle \in tr \quad id = \texttt{new\_id}(tr) \quad v = \langle\, pv_1, id_1 \,\rangle :: \langle\, pv_2, id_2 \,\rangle :: \texttt{nil} \quad p \xrightarrow{f(\cdot, pv_2),\mathbf{T}} p' \quad tr' = tr[\langle\, id, p \,\rangle \mapsto \langle\, id, p' \,\rangle]}{\langle\, tr, T \,\rangle \xrightarrow{f,v} \langle\, tr', \langle\, [\![f]\!]\,(pv_1, pv_2), id \,\rangle \,\rangle} \text{(FBINARY)}$$

$$\frac{v = \overline{\langle\, pv, id \,\rangle} \quad pvs = \texttt{map}(\texttt{fst}, v) \quad ids = \texttt{map}(\texttt{snd}, v) \quad \forall i \in |P| \Longrightarrow (\langle\, ids_i, p_i \,\rangle \in tr) \wedge \left(p_i \xrightarrow{\texttt{agg},\mathbf{A}} p_i'\right) \quad tr' = \langle\, id', \biguplus_i p_i' \,\rangle :: tr \quad id' = \texttt{new\_id}(tr) \quad P = \{p_1, \cdots\}}{\langle\, tr, T \,\rangle \xrightarrow{\texttt{agg},v} \langle\, tr', \langle\, \texttt{fold}([\![\texttt{agg}]\!], pvs), id' \,\rangle \,\rangle} \text{(FAGG)}$$

Figure 8: Auxiliary rules for $\langle\, tr, T \,\rangle \xrightarrow{f,v} \langle\, tr', v' \,\rangle$ that directly manipulates policy checks and transitions that are highlighted.

a wide array of programming languages. Second, it is more practical and secure to establish a barrier between third-party code and PICACHV. This separation allows for better control and monitoring of interactions between external functions and the core system. We also believe that programmers can provide vetted code for these functions.

5.1.2 Relational Operators

We now turn our attention to the reduction rules for the operators of our core calculus RA^{P}, as illustrated in Figure 9. This behavior is formalized using big-step operational semantics, represented by the following judgment form:

$$\Sigma \vdash q \Downarrow \langle\, R, tr \,\rangle$$

This establishes a relation between the initial evaluation context Σ, q, which consists of the data store and the query q, and the resulting relation R along with the updated trace tr.

We present reduction rules for relational operators in Figure 7. The JOIN rule first evaluates its sub-queries, yielding their results R_1 and R_2. It then iterates over the tuples in the left relation, such that for each tuple $t_1 \in R_1$, it attempts to concatenate t_1 with all tuples $t_2 \in R_2$ from the right relation. The JOINT rule, which is used by the JOIN rule, is particularly noteworthy. This rule attempts to join a tuple t with a relation R. Since a join operation requires specifying which columns are used as keys, we use $t \upharpoonright_{\overline{n}} \langle\, t_1, t_1^\star \,\rangle$ to denote splitting the tuple t into two parts: t_1, which contains the selected columns (keys), and $t_1^\star$, which contains the remaining columns. The JOINT rule then iterates over the tuples in the right relation R. It ensures that all identifiers and policies of the tuple being joined are present in the initial trace tr. During each iteration, if the tuples t_1 and t_2 agree on the joined part (denoted as $t_1 \doteq t_2$), their policies p_1 and p_2 are composed as $p_1 \uplus p_2$, and the new policy is inserted into the updated trace tr'. The UNION rule simply unions the result of the two sub-queries.

The base rule is presented in RELATION where we fetch the policy from the policy store and transform it into a trace tr consisting of TrNone. For SELECT, we iterate over each tuple $R[i] \in R$ obtained from the evaluated result of the subquery, and we evaluate the predicate φ thereon. Since evaluating the predicate does not involve any policy-related operations, we disregard the program trace, using $\star$ as a special placeholder to indicate that it is irrelevant in this context and can be any well-typed traces. The evaluation of φ on each tuple produces a boolean value, resulting in a vector $\{b_i, \ldots\}$. This vector is then used to filter the relation R through the operation $\times$, retaining only the tuples that satisfy the predicate. PROJECT evaluates each expression $e_i \in \overline{e}$ on every tuple $R'[j] \in R'$. The results are concatenated into a single tuple $R[j]$, and all such tuples are combined through a union operation to produce the final result. The AGGREGATE rule begins by deriving the grouping information $R' \searrow_{\overline{g}} \overline{G}$ from R', which is obtained from the evaluation result of the sub-query. It then iterates over the list of aggregate expressions, evaluating each expression e_i using the tuple context T, which is created by slicing R' based on $\overline{n}_j$. Eventually, we obtain the result by applying the grouping predicate on the intermediate result.

5.2 Security Conditions

The key security property of PICACHV's semantics is ensuring that the enforcement is *sound*. Following this work [35], the enforcement mechanism ensures that data tagged as low has been downgraded via specified functions in accordance with its policy.

Definition 5.1 (Relaxed non-interference)**.** For any data store Σ, query q, either the evaluation of Σ, q results in an error, i.e., $\Sigma \vdash q \Downarrow \text{✗}$, or the following holds:

$$\Sigma \vdash q \Downarrow \langle\, R, tr \,\rangle \Longrightarrow (\forall c \in [\![R]\!], \langle\, c, \mathbf{L} \,\rangle \in tr \Longrightarrow \mathcal{E}(c) \approx \Sigma(c)),$$

where $[\![R]\!]$ means to extract identifiers of all data in the relation, $\mathcal{E}$ is a trace extraction function defined in Figure 10, $\approx$ is a compatible relation between policies, and we use $\Sigma(c)$ to denote the initial policy for c. Informally, $\mathcal{E}$ identifies every transformation path of the data involved in computing $c \in R$. We define the compatible relation in Figure 11.

This security definition encapsulates two key scenarios: (1) evaluation results in an error due to a policy breach, re-

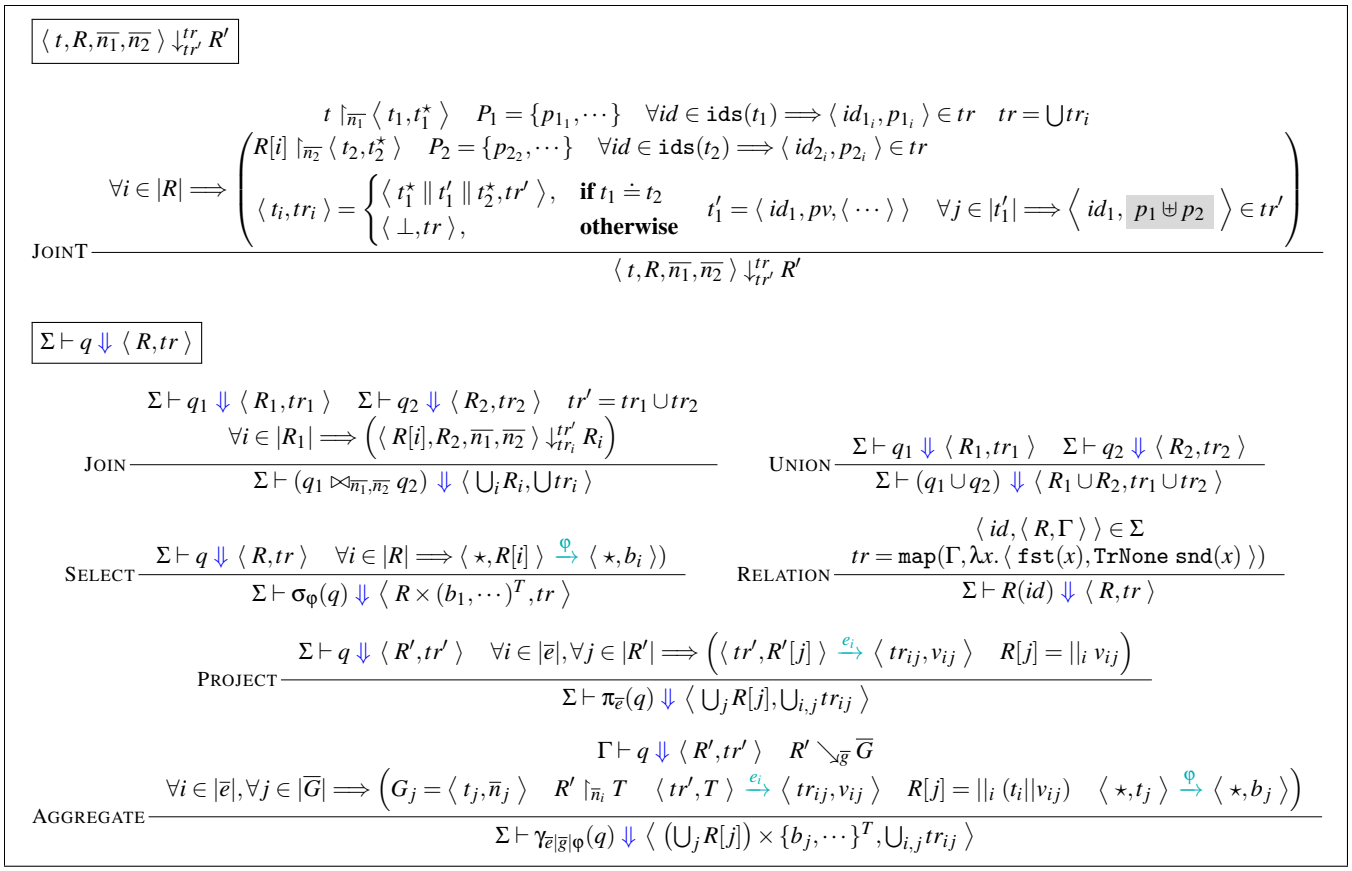


Figure 9: Selected reduction rules for RA^{P}.

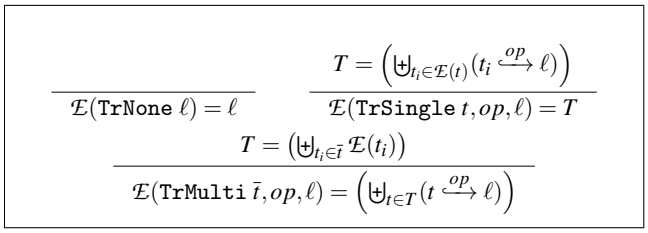


Figure 10: Trace extraction rules.

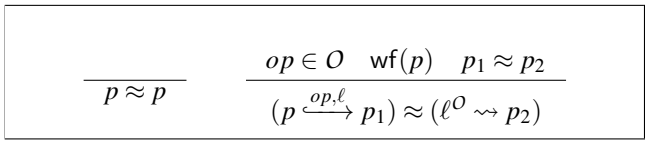


Figure 11: Rules for the compatible relation $\approx$.

turning no output. (2) Relaxed non-interference ensures that all data tagged as low **L** has undergone proper declassification procedures as specified by the data owner. Notably, when no declassification is allowed (i.e., all data is set to **H**), this definition reduces to standard non-interference. For instance, consider a simple program that performs an aggregation over the age attribute with a policy $\mathbf{A}^{\mathbf{avg}} \rightsquigarrow \mathbf{L}$, oututting a final relation R containing a single value, $\langle\, pv, id \,\rangle$. Then $\mathcal{E}(id) \equiv (\mathcal{E}(c_0) \xrightarrow{\mathbf{avg},\mathbf{A}} \mathbf{L}) :: (\mathcal{E}(c_1) \xrightarrow{\mathbf{avg},\mathbf{A}} \mathbf{L}) :: \cdots$.

Theorem 5.1 (Soundness)**.** The semantics enforces relaxed non-interference.

Theorem 5.2 (Strict non-interference)**.** If no declassification is permitted, then our semantics enforces strict non-interference.

6 Implementation

In this section, we explain how the PICACHV runtime monitor operates and how policies are encoded.

6.1 Implementing Policies

Encoding policies using shadow tables. As described in Section 4, each relation is modeled as a list of tuples, with each cell assigned a unique identifier *id* linking it to its correspond-

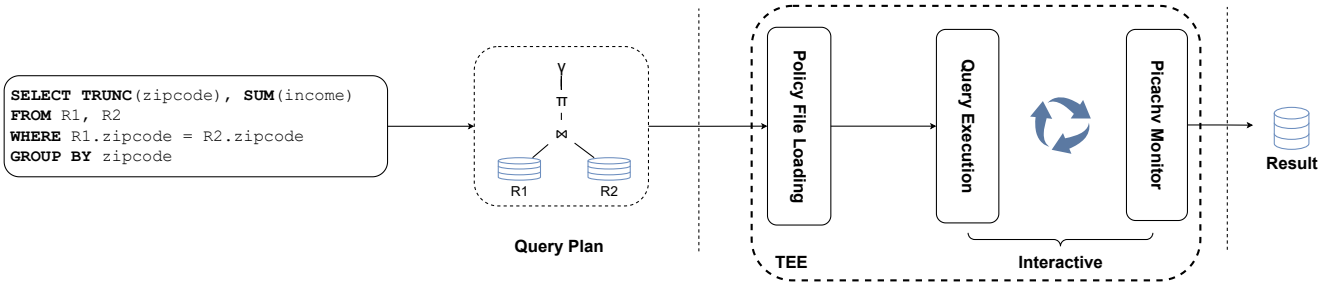


Figure 12: The high-level query execution workflow where we put the query execution engine and PICACHV runtime monitor inside a TEE.

ing policies. To address the inefficiency of frequent policy lookups during execution, particularly in parallel processing environments (that require locks), we introduce the concept of *shadow tables*, which maintains only the policy tags for the given relation in a one-to-one correspondence. Shadow tables mirror the structure of the original tables but store policy tags instead of the actual data. This design separates data from its policies, enabling efficient runtime policy enforcement. This way, the policies for a given relation are stored separately with the data and will be loaded at runtime when query is being executed. In our implementation, shadow tables that store policies are encoded in Apache Parquet format where we serialize policies into byte arrays and compress them.

Support for flexible policies. Shadow tables also support flexibility by allowing policymakers to load different policies for the same relation from separately stored policy files. For instance, researchers from different institutions working on the same dataset may require distinct data use policies tailored to their roles. The policymaker who controls the data can design different policies for the same patient data depending on the roles of the researcher. At the same time, this way allows policies to be arbitrarily composed when multiple relations with different policy requirements are being joined during analysis. Even better is that this design allows for policy overlay. Imagine if we have a "base policy" for a given table and there would be several users having different privacy preferences. As such, these requirements can be overlayed on the base policy by applying the policy composition rules shown in Figure 4.

6.2 Implementing PICACHV

At a high level, the implementation of PICACHV consists of two key components: a formal proof written in Coq and a runtime monitor developed in Rust, designed as a standalone dynamic library. This library can be integrated into existing data analytics frameworks through foreign function interfaces (FFIs), requiring minimal modifications to the codebase. Frameworks like `Pandas` and `SparkQL` can invoke PICACHV 's monitoring APIs to enforce data use policies dynamically whenever an executor is executed.

The high-level workflow of PICACHV is illustrated in Figure 12. The process begins with the parsing and transformation of a query into a query plan, which is subsequently forwarded to the execution engine. PICACHV identifies and retrieves the relevant policy files from disk, based on the tables referenced in the query—such as R1 and R2 in this example. Query execution proceeds interactively, with PICACHV actively monitoring each node's execution in real time. If the query adheres to the specified policies, the computed results are allowed to exit PICACHV. Conversely, if the query violates any policies, an error is raised to block non-compliant results from being returned. Note that we place PICACHV and data inside TEEs for confidentiality, integrity, and verifiability.

Query execution. Figure 13 details the query execution process in PICACHV. Data and policy shadow tables are first fetched by `TableScan` operator, then processed by `Project` and `Aggregate`, and finally `sink` is applied. At a high level, since shadow tables are maintained for the original tables, the execution process is divided into two parallel phases. Recall that the way shadow tables are implemented can support flexible policies for the *same table* under different circumstances. The effects of relational operators on the shadow tables will being actively captured by the semantics described in Figure 9, while the actual data is processed using the native query executor (in blue). In other words, the effects of the relational operators (join $\bowtie$, projection π, and aggregate γ) on the data will also be *shadowed* by the FFI calls. For instance, when PICACHV is integrated with MySQL, its native query executors are invoked to process the corresponding node in the query plan. However, the native executor can only proceed to the next node if PICACHV *confirms* that its checks are passed, and no policy breaches are detected

Sink. After execution, a sink function (see Figure 13) is applied before results are returned. This step prevents data with remaining tags from inadvertently leaving the protected environment. This extra step is required to ensure operations must be performed on the data. Consider a scenario where an attacker attempts to access raw personal identifiers from a dataset that should first be aggregated. Permitting such data

to pass through would breach the policy. Therefore, we implement a sink function that checks for the presence of any tags on the data. Thus, this final safeguard ensures that only properly processed and policy-compliant data is released from the system, with the protection of the query execution phase where *disallowed operations* should never be performed.

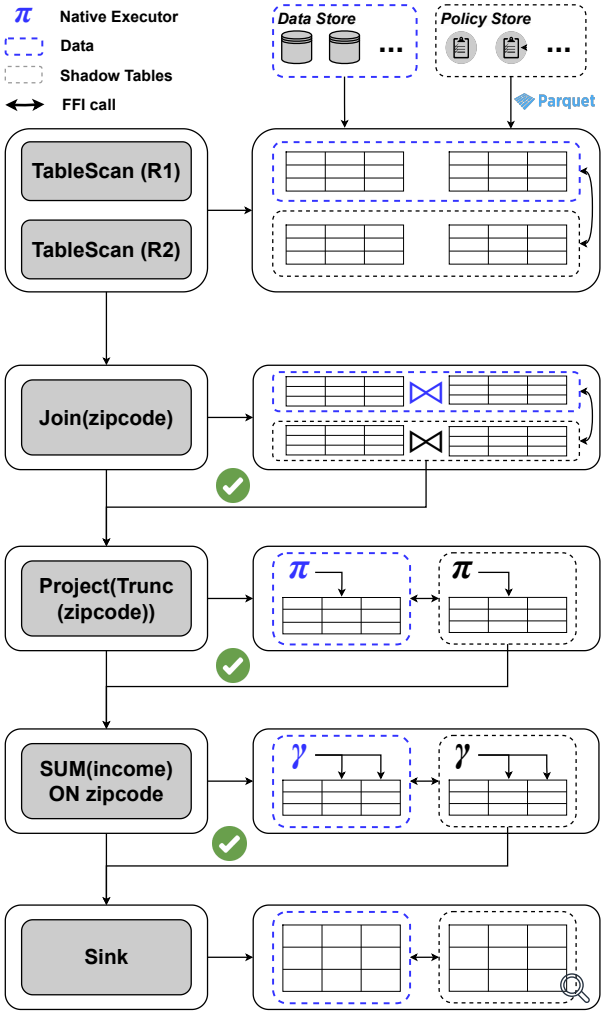


Figure 13: The query execution phase.

Verifiability. The primary motivation for leveraging TEEs to host PICACHV is to provide verifiable guarantees for private data processing on the server side to external parties (data owners) who lack direct control over their data in the cloud. TEEs offer data owners the ability to verify:

- *What* is being executed within the TEE by utilizing *remote attestation*, which produces a non-forgeable cryptographic report detailing the TEE's properties.
- *How* the binary running inside the TEE was built, ensuring it originates from verified source code through a trusted build system.

By initiating a remote attestation session, data owners can validate the source code and build process referenced in the attestation report, providing reassurance that their data is being processed securely and as intended.

6.3 Optimization Techniques

The nature of PICACHV as a dynamic security monitor naturally raises performance concerns due to the overhead of managing security labels at the cell level. We detail in this section potential optimization techniques.

Materialization and caching. Inspired by materialization techniques commonly used in databases [25, 33], we adapt this concept for policy checking in PICACHV. Our approach caches the verifications of frequently queried plans or subplans that have already been confirmed as compliant with data use policies. When a new query is issued, query rewriting techniques are applied to determine if the query can be transformed to leverage these cached materialized views. This strategy reduces redundant checks and significantly reuses prior verification computations, leading to improved efficiency. However, implementing this approach poses challenges for traditional program analysis and verification methods, as they typically lack the higher-level semantic abstractions necessary to capture the intent of programs.

Hybrid scheme. One of the key advantages of leveraging query plans is the inherent ability to gain deep insights into data flow. Query plans are embedded with strong semantics that detail not only the flow of data but also the types of operations performed, dependencies among data elements, and potential interferences. This rich semantic information allows us to conduct preliminary static analysis before engaging in dynamic verification processes. By performing static analysis on the query plan, we can potentially verify the entire query or its subcomponents before execution. If the static analysis successfully validates the entire query, then one can proceed to the next query without further checks. Moreover, in cases where the static analysis is insufficient or incomplete, dynamic verification can be seamlessly initiated from the point where the static analysis has left.

7 Evaluation

In this section, we present the experimental results of PICACHV to demonstrate that 1) The additional runtime overhead of PICACHV is small, 2) PICACHV can support many real-world analytical tasks and policies, and 3) PICACHV can enforce these privacy policies.

7.1 Experiment Setup

Test environment. Our evaluation was conducted inside a VM TEE that is based on the Intel Trusted Domain eXtension (TDX) on a server with two 2.3 GHz Intel Xeon Platinum 8568Y+ CPUs (a total of 96 cores and 192 threads) and 512 GB of memory, running Ubuntu 22.04. We have integrated PICACHV into a state-of-the-art data analytical engine called `Polars` [9] (31K stars on GitHub): A powerful library for high-performance data manipulation and analysis in Python and Rust. We build all the components in release mode with optimization level at `O3`.

Dataset and test suite. In our benchmark, we employ the latest TPC-H specification (v3.0.1) [10], using `tpch-dbgen` [6] to generate a dataset of different sizes by changing the scale factors. The TPC-H is a decision support benchmark. It consists of a suite of 22 business-oriented *ad hoc* queries on data split across 8 tables. The queries and the data populating the database have been chosen to have broad industry-wide relevance. We implement TPC-H queries in `Polars` (using its dataframe APIs). Currently, there are no official data use policies for the TPC-H testbed. To address this gap, we manually crafted policies to simulate real-world scenarios. We then manually verified output correctness.

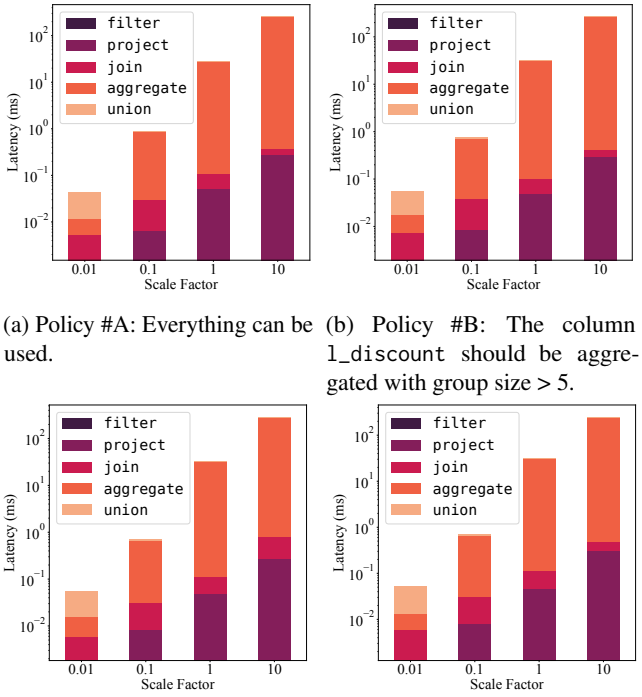
7.2 Performance Overhead

End-to-end latency. We begin by presenting an overview of PICACHV's end-to-end performance overhead, comparing it against the unmodified query engines from `Polars` as the insecure baseline. We evaluate performance using selected queries from the TPC-H benchmark suite. In this experiment, we set the scale factor to 1. Queries not included in the benchmark contain some features currently not yet supported by PICACHV. We exclude the policy file reading time from our measurements to provide a more accurate representation of runtime performance. We believe this overhead can be mitigated through strategies such as preloading policies during startup, and this often occurs infrequently. To measure the end-to-end performance of query execution with policy checking enabled, we assign dummy labels (**L**) to each cell in this benchmark. Figure 15 shows the complete experimental results. PICACHV generally shows higher execution times than the baseline (from $\sim 1.2\times$ to $\sim 15\times$), indicating some overhead from policy checking. Some queries, like Q8, Q12, and Q13, show minimal differences between Picachv and the baseline, while others, such as Q9, Q15, exhibit more noticeable performance gaps. Such large overheads, as indicated in the microbenchmark (see Figure 14), can be attributed to both the `projection` and `aggregate` operators.

Microbenchmark. To understand what components contribute to the major runtime overhead, we choose query Q3 because the minimal query incorporates all the necessary operations we want to evaluate from the TPC-H benchmark, and

Table size	Policy file loading time (s)
10 MB	1.67
100 MB	12.16
1 GB	91.50
10 GB	703.64

Table 1: Time used to load policy files.



(a) Policy #A: Everything can be used.

(b) Policy #B: The column `l_discount` should be aggregated with group size > 5.

(c) Policy #C: The column `l_discount` should be added by 1.0.

(d) Policy #D: Composition of Policy #B and Policy #C.

Figure 14: The result of the microbenchmark on each relational operator's runtime overhead.

we slightly modified it to accommodate the policy. We run it atop `Polars` to analyze the breakup of runtime overhead. The benchmark results are reported in Figure 14. To give a clearer understanding of which component significantly contributes to the overhead, we split the overhead into the following parts: 1) the cost of policy file loading, 2) `project`, 3) `join`, 4) `aggregate`, 5) `union`, and 6) `filter`. We also designed this query for different privacy policies to see which kinds of policies will cause significant policy checking overhead. Also note that in the microbenchmark, we do not consider the time taken for `polars` to fulfill the query but solely consider the time taken by PICACHV. In this experiment, we set the scale factor of the database generation from 0.01 to 10 (sizes from 10 MB to 10 GB) to test the performance under different sizes. In our experiment, we observed considerable overhead when loading the policy files from the disk, and we report the

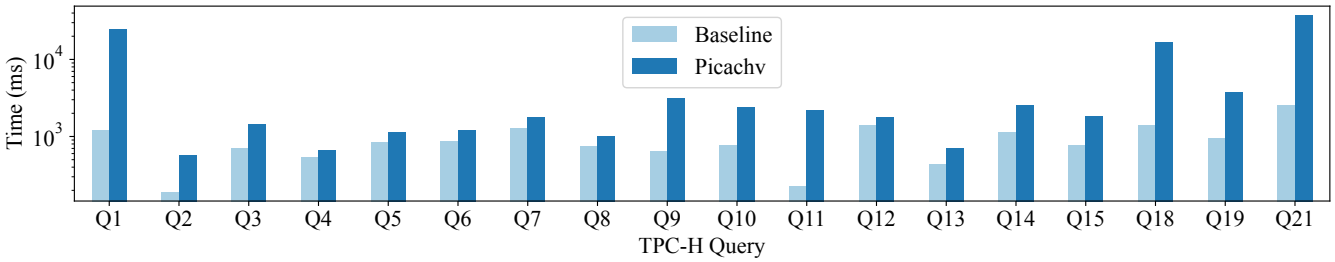


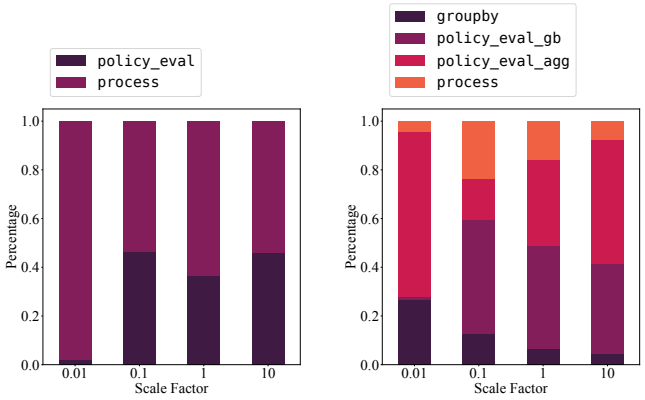
Figure 15: The runtime overhead of TPC-H testbed of PICACHV.

Dataset used in the task	Task Name	Execution Time (ms)	Checking Time (ms)	Result
Chronic illness [3]	1: autoimmune [2]	32.69	469.394 (14.36×)	✗; `user_id`
	2: EDA [5]	23.93	1416.04(59.17×)	✗; group size too small
	3: Symptoms [4]	170.25	588.089 (3.46×)	✓
Healthcare Dataset [8]	4: Healthcare [1]	22.12	299.95 (13.56×)	✓
	5: Trends [11]	49.58	132.55 (2.67×)	✓
	6: Analysis [7]	57.02	598.94 (10.50×)	✓

Table 2: Results of analytical tasks in the case study are presented. We use ✓ to indicate programs that comply with the privacy policy and ✗ to denote instances where a policy breach was detected. For programs that fail to meet policy requirements, we provide explanations. The original code was written in Python and subsequently adapted to Rust. We recorded execution times without policy enforcement to establish a baseline for comparison.

latency in Table 1 to give a clearer understanding of the overhead incurred by policy checking at runtime. Despite such a large overhead, we believe it can be reduced by utilizing more advanced optimization techniques. We instead focus on the runtime overhead of policy checking of PICACHV. The experimental results in Figure 14 show that the major performance overhead comes from the `project` and `aggregate` operator that involves data transformation, and as data grows, the percentage of the `aggregate` operator soon dominates (from 1% to nearly 99%). Interestingly, the impact of privacy policy types is minimal during our evaluation.

Analysis on `project` and `aggregate` operators. To understand *why* `project` and `aggregate` contribute significantly to runtime overhead, we conducted a detailed benchmark for these operators. In this experiment, we fixed the policy type of the composition of policy #B and policy #C to isolate their impact. In PICACHV 's implementation, the `project` operator consists of two sub-routines: a) `policy_eval`, which applies declassification rules to policy tags, and b) `process`, which handles intermediate in-memory representations for further processing. The `aggregate` operator includes: a) `group by`, grouping tuples as per the query plan; b) `policy_eval_gb`, applying declassification rules during grouping; c) `policy_eval_agg`, declassifying grouped policies; and d) `process`. Shadow tables are already in-memory, so disk I/O overhead is excluded. We decompose these operators into their primitives and provide a cost breakdown in Figure 16. For the `project` operator (Figure 16a), `process` consistently dominates runtime overhead across all



(a) The cost breakdown of the `project` operator.

(b) The cost breakdown of the `aggregate` operator.

Figure 16: Detailed analysis on the `project` and `aggregate` operators in terms of the percentage of their sub-routines.

scale factors, while `policy_eval` remains a smaller but noticeable contributor, especially at smaller scales. This shows the primary overhead in `project` comes from in-memory processing logic rather than policy evaluation. For the `aggregate` operator (Figure 16b), the experimental result highlights the increased complexity of its sub-routines. As the scale factor grows, the `process` 's contribution remains prominent, but the costs of `policy_eval_gb` and `policy_eval_agg` also grow proportionally. Notably, `group by` contributes a smaller share to the overall latency because this operation can be efficiently

parallelized. Since `aggregate` requires folding on the groups, This indicates that the policy evaluation stages in `aggregate` are computationally intensive and scale-sensitive. This explains why aggregate-intensive queries in the TPC-H testbed might have more runtime overhead. Future works can be focused on designing more efficient aggregate algorithms.

7.3 Case Studies

To demonstrate PICACHV's practical applications and versatility, we conducted case studies using two datasets: a chronic illness dataset and a healthcare dataset. We adapted three analytical tasks for each dataset, implementing them using `Polars`. For these tasks, we manually crafted privacy policies simulating real-world scenarios such as those found in the All-of-Us project [44] and HIPAA regulations [43]. The results, presented in Table 2, show that PICACHV successfully detected policy violations in two out of six tasks, while allowing the compliant tasks to proceed. Notably, the policy checking time often exceeded the task execution time, and the in the worst case this would introduce nearly 59.17× overhead, suggesting potential for future optimization. We found this is mainly due to the `group by` operation found commonly in data analytics, as suggested by Figure 14 and Figure 16b.

These case studies validate PICACHV's capability to enforce complex privacy policies particularly in sensitive domains like healthcare. The monitor's ability to detect and prevent policy violations, even for quickly executed tasks, underscores its value in ensuring responsible data use. Furthermore, the application of PICACHV across diverse analytical tasks demonstrates its potential for adaptation to various data-intensive fields requiring stringent privacy protection.

8 Discussion

Semi-structured and unstructured data. PICACHV's operation at the relational algebra level inherently limits its support for semi-structured and unstructured data. This limitation is significant, as many real-world workloads involve interactions with such data types. For instance, electronic health record (EHR) analysis often requires processing clinical notes, which are unstructured free text. Similarly, human genome analysis deals with semi-structured Single Nucleotide Polymorphism (SNP) data. Enforcing privacy policies on these diverse data formats presents a substantial challenge that extends beyond PICACHV's current capabilities. Addressing this limitation would greatly enhance the system's applicability across a broader range of data-intensive domains.

Expressiveness of relational algebra. While PICACHV's use of relational algebra as an abstraction enables policy enforcement for a wide range of data operations, it also presents limitations in expressiveness for certain advanced analytics tasks. Many modern data analytics workflows, particularly in machine learning and artificial intelligence, involve operations that extend beyond the capabilities of traditional relational algebra. For instance, complex matrix operations, iterative algorithms, and non-linear transformations common in ML models cannot be directly expressed using standard relational operators. Recent work has proposed primitive operators for a tensor relational algebra [54] tailored to these specific use cases, analogous to SPJUA in traditional relational algebra. Future work could explore extending PICACHV's current solution to incorporate such algebras, potentially broadening its applicability to more advanced analytics scenarios.

Complete verification. We currently trust the query planner to produce correct and policy-compliant plans. However, this trust assumption introduces a potential vulnerability in the overall system. In reality, a comprehensive security guarantee would require verification of the entire pipeline, including the query planner. Verifying the planner would ensure that the generated query plans themselves adhere to the specified policies and do not introduce unexpected data flows or operations that could violate privacy constraints. This extension of our verification scope represents an important avenue for future work, as it would close a significant gap in the end-to-end formal guarantees of our system.

Automated policy interpretation. Translating privacy regulations like GDPR into computer-interpretable policies typically requires significant human effort, and this work implicitly assumes the existence of such policies. Recent advancements in Natural Language Processing (NLP), such as ARC [40], offer promising solutions for automating this process. These technologies could bridge the gap between human-readable regulations and machine-executable policies, enhancing efficiency and accuracy in applying privacy standards. While crucial for privacy compliance, we consider this challenge orthogonal yet complementary to PICACHV. The synergy between automated policy interpretation and PICACHV could significantly advance privacy-preserving data analytics.

9 Conclusion

In this paper, we present PICACHV, a significant advancement in enforcing data use policies for analytics. By abstracting program semantics via relational algebra and employing formal verification, we have created a system that effectively balances policy compliance with analytical flexibility. Our evaluations demonstrate PICACHV's efficiency, accuracy, and real-world applicability across various regulatory frameworks. While limitations exist, particularly for non-relational data, PICACHV provides a robust foundation for responsible data usage in an increasingly data-driven world. This work paves the way for future developments in secure data analytics.

Acknowledgements

We extend our sincere gratitude to our shepherd and the anonymous reviewers for their invaluable feedback and constructive suggestions. We also wish to thank Prof. Danfeng Zhang for the insightful discussions, the members of CDCC, Mona Vij, and Marcela Melara from Intel for their thoughtful comments on an early draft of this paper, and Haosen Guan for his kind support. This work was supported by NSF under awards CNS-2207231 and OAC-2419821. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Ethics Considerations

The authors of this paper carefully reviewed related documents and PICACHV in its design and implementation. As a policy enforcement tool, the design and implementation do not involve any ethical considerations.

Open Science

All data and programs used in the evaluation section are publicly available. Furthermore, the authors will fully disclose the source code, formal proofs, and other benchmark tools as a standalone artifact to the public to support future research. Code can be found at https://github.com/picachv and at https://zenodo.org/records/14639575.

References

[1] analysis healthcare dataset — kaggle.com. https://www.kaggle.com/code/manarmohamed24/analysis-healthcare-dataset. [Accessed 05-09-2024].

[2] autoimmune-symptom — kaggle.com. https://www.kaggle.com/code/donottalk/autoimmune-symptom. [Accessed 03-09-2024].

[3] Chronic illness: symptoms, treatments and triggers — kaggle.com. https://www.kaggle.com/datasets/flaredown/flaredown-autoimmune-symptom-tracker/data. [Accessed 02-09-2024].

[4] Flaredown Autoimmune Symptoms Prediction — kaggle.com. https://www.kaggle.com/code/dzmitryashkinadze/flaredown-autoimmune-symptoms-prediction. [Accessed 05-09-2024].

[5] Flaredown Data Exploratory Analysis — kaggle.com. https://www.kaggle.com/code/ultron2412/flaredown-data-exploratory-analysis#Symptoms. [Accessed 05-09-2024].

[6] GitHub - electrum/tpch-dbgen: TPC-H dbgen — github.com. https://github.com/electrum/tpch-dbgen. [Accessed 12-08-2024].

[7] Health Care Data Analysis — kaggle.com. https://www.kaggle.com/code/vinod123kumar/health-care-data-analysis. [Accessed 05-09-2024].

[8] Healthcare Dataset — kaggle.com. https://www.kaggle.com/datasets/prasad22/healthcare-dataset/data. [Accessed 02-09-2024].

[9] Polars. https://pola.rs. Accessed: 2023-07-03.

[10] The tpc-h benchmark. https://www.tpc.org/tpch. Accessed: 2023-07-16.

[11] Unlocking Healthcare Trends: Data Analysis — kaggle.com. https://www.kaggle.com/code/muhammadfurqan0/unlocking-healthcare-trends-data-analysis. [Accessed 05-09-2024].

[12] Anindya Banerjee, David A Naumann, and Stan Rosenberg. Expressive declassification policies and modular static enforcement. In *2008 IEEE Symposium on Security and Privacy (SP'08)*, pages 339–353. IEEE, 2008.

[13] Abhishek Bichhawat, Matt Fredrikson, Jean Yang, and Akash Trehan. Contextual and granular policy enforcement in database-backed applications. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security (AsiaCCS'20)*, pages 432–444, 2020.

[14] US Census Bureau. Disclosure avoidance for the 2020 census: An introduction, 2021.

[15] Ethan Cecchetti, Andrew C Myers, and Owen Arden. Nonmalleable information flow control. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security(CCS'17)*, pages 1875–1891, 2017.

[16] Pau-Chen Cheng, Wojciech Ozga, Enriquillo Valdez, Salman Ahmed, Zhongshu Gu, Hani Jamjoom, Hubertus Franke, and James Bottomley. Intel tdx demystified: A top-down approach. *ACM Computing Surveys*, 56(9):1–33, 2024.

[17] Adam Chlipala. Static checking of *dynamically – varying* security policies in *database – backed* applications. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI' 10)*, 2010.

[18] Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. Secure web applications via automatic partitioning. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, page 31–44, New York, NY, USA, 2007. Association for Computing Machinery.

[19] Stephen Chong and Andrew C Myers. Security policies for downgrading. In *Proceedings of the 11th ACM conference on Computer and communications security (CCS'04)*, pages 198–209, 2004.

[20] Victor Costan and Srinivas Devadas. Intel sgx explained. *Cryptology ePrint Archive*, 2016.

[21] Cynthia Dwork. Differential privacy. In *International colloquium on automata, languages, and programming*, pages 1–12. Springer, 2006.

[22] Hubert Eichner, Daniel Ramage, Kallista Bonawitz, Dzmitry Huba, Tiziano Santoro, Brett McLarnon, Timon Van Overveldt, Nova Fallen, Peter Kairouz, Albert Cheu, et al. Confidential federated computations. *arXiv preprint arXiv:2404.10764*, 2024.

[23] Mafalda Ferreira, Tiago Brito, José Fragoso Santos, and Nuno Santos. Rulekeeper: Gdpr-aware personal data compliance for web frameworks. In *2023 IEEE Symposium on Security and Privacy (SP'23)*, pages 1014–1031. IEEE Computer Society, 2022.

[24] Tim Fischer, Denis Hirn, and Torsten Grust. Snakes on a plan: Compiling python functions into plain sql queries. In *Proceedings of the 2022 International Conference on Management of Data*, pages 2389–2392, 2022.

[25] Robert C. Goldstein and Veda C. Storey. Materialization [database design]. *IEEE Transactions on Knowledge and Data Engineering*, 6(5):835–842, 1994.

[26] Paolo Guagliardo and Leonid Libkin. A formal semantics of sql queries, its validation, and applications. *Proceedings of the VLDB Endowment*, 11(1):27–39, 2017.

[27] Marco Guarnieri, Musard Balliu, Daniel Schoepe, David Basin, and Andrei Sabelfeld. Information-flow control for database-backed applications. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 79–94. IEEE, 2019.

[28] Stefan Hagedorn, Steffen Kläbe, and Kai-Uwe Sattler. Putting pandas in a box. In *Conference on Innovative Data Systems Research (CIDR);(Online)*, page 15, 2021.

[29] Dongseok Jang, Zachary Tatlock, and Sorin Lerner. Establishing browser security guarantees through formal shim verification. In *Proceedings of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 113–128, Bellevue, WA, August 2012. USENIX Association.

[30] Konstantinos Karanasos, Matteo Interlandi, Doris Xin, Fotis Psallidas, Rathijit Sen, Kwanghyun Park, Ivan Popivanov, Supun Nakandal, Subru Krishnan, Markus Weimer, et al. Extending relational query processing with ml inference. *arXiv preprint arXiv:1911.00231*, 2019.

[31] Byeongwook Kim, Jaewon Hur, Adil Ahmad, and Byoungyoung Lee. Laputa: Secure data analytics in apache spark with fine-grained policy enforcement and isolated execution. In *Network and Distributed Systems Security*, 2025.

[32] Elisavet Kozyri, Stephen Chong, Andrew C Myers, et al. Expressing information flow properties. *Foundations and Trends® in Privacy and Security*, 3(1):1–102, 2022.

[33] P-A Larson, Jonathan Goldstein, and Jingren Zhou. Mtcache: Transparent mid-tier database caching in sql server. In *Proceedings. 20th International Conference on Data Engineering*, pages 177–188. IEEE, 2004.

[34] Nico Lehmann, Rose Kunkel, Jordan Brown, Jean Yang, Niki Vazou, Nadia Polikarpova, Deian Stefan, and Ranjit Jhala. Storm: refinement types for secure web applications. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI' 21)*, 2021.

[35] Peng Li and Steve Zdancewic. Downgrading policies and relaxed noninterference. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'05)*, pages 158–170, 2005.

[36] Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui. Formally verified memory protection for a commodity multiprocessor hypervisor. In *Proceedings of the 30th USENIX Security Symposium (USENIX Security 21)*, pages 3953–3970. USENIX Association, August 2021.

[37] Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui. A secure and formally verified linux kvm hypervisor. In *2021 IEEE Symposium on Security and Privacy (SP'21)*, pages 1782–1799, 2021.

[38] Wen-jie Lu, Zhicong Huang, Qizhi Zhang, Yuchen Wang, and Cheng Hong. Squirrel: A scalable secure {Two-Party} computation framework for training gradient boosting decision tree. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 6435–6451, 2023.

[39] Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Toward a verified relational database management system. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'10)*, pages 237–248, 2010.

[40] Sunil Manandhar, Kapil Singh, and Adwait Nadkarni. Towards automated regulation analysis for effective privacy compliance. In *Network and Distributed System Security Symposium (NDSS'24)*, pages 631–647, 2024.

[41] Alana Marzoev, Lara Timbó Araújo, Malte Schwarzkopf, Samyukta Yagati, Eddie Kohler, Robert Morris, M Frans Kaashoek, and Sam Madden. Towards multiverse databases. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS'19)*, pages 88–95, 2019.

[42] Aastha Mehta, Eslam Elnikety, Katura Harvey, Deepak Garg, and Peter Druschel. Qapla: Policy compliance for database-backed systems. In *26th USENIX Security Symposium (Sec' 17)*, pages 1463–1479, 2017.

[43] Rachel Nosowsky and Thomas J Giordano. The health insurance portability and accountability act of 1996 (hipaa) privacy rule: implications for clinical research. *Annu. Rev. Med.*, 57:575–590, 2006.

[44] All of Us Research Program Investigators. The "all of us" research program. *New England Journal of Medicine*, 381(7):668–676, 2019.

[45] Andrei Sabelfeld and Andrew C Myers. Language-based information-flow security. *IEEE Journal on selected areas in communications*, 21(1):5–19, 2003.

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

[47] Daniel Schoepe, Daniel Hedin, and Andrei Sabelfeld. Selinq: tracking information across application-database boundaries. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming (ICFP'14)*, pages 25–38, 2014.

[48] David Schultz and Barbara Liskov. Ifdb: decentralized information flow control for databases. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys'13)*, pages 43–56, 2013.

[49] AMD Sev-Snp. Strengthening vm isolation with integrity protection and more. *White Paper, January*, 53:1450–1465, 2020.

[50] Hesam Shahrokhi, Amirali Kaboli, Mahdi Ghorbani, and Amir Shaikhha. Pytond: Efficient python data science on the shoulders of databases. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*, pages 423–435. IEEE, 2024.

[51] Lun Wang, Usmann Khan, Joseph Near, Qi Pang, Jithendaraa Subramanian, Neel Somani, Peng Gao, Andrew Low, and Dawn Song. *PrivGuard*: Privacy regulation compliance made easier. In *31st USENIX Security Symposium (Sec' 22)*, pages 3753–3770, 2022.

[52] Jean Yang, Travis Hance, Thomas H Austin, Armando Solar-Lezama, Cormac Flanagan, and Stephen Chong. Precise, dynamic information flow for database-backed applications. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'16)*, pages 631–647, 2016.

[53] Jean Yang, Kuat Yessenov, and Armando Solar-Lezama. A language for automatically enforcing privacy policies. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'12)*, pages 85–96, 2012.

[54] Binhang Yuan, Dimitrije Jankov, Jia Zou, Yuxin Tang, Daniel Bourgeois, and Chris Jermaine. Tensor relational algebra for distributed machine learning system design. *Proceedings of the VLDB Endowment*, 14(8), 2021.

[55] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 10)*, 2010.

[56] Wen Zhang, Aurojit Panda, and Scott Shenker. Access control for database applications: Beyond policy enforcement. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, HOTOS '23, page 223–230, New York, NY, USA, 2023. Association for Computing Machinery.

[57] Wen Zhang, Eric Sheng, Michael Chang, Aurojit Panda, Mooly Sagiv, and Scott Shenker. Blockaid: Data access policy enforcement for web applications. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI' 22)*, pages 701–718, 2022.

[58] Shijun Zhao, Qianying Zhang, Yu Qin, Wei Feng, and Dengguo Feng. Sectee: A software-based approach to secure enclave architecture using tee. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1723–1740, 2019.

A Proofs

A.1 Proofs for Theorem 5.1

Proof Sketch. Let Σ be any data store, q be any query, and st be any program state. By definition, either $\Sigma \vdash q \Downarrow \langle R, tr \rangle$

holds or it does not. For the former case, we apply mathematical induction over $\Sigma \vdash q \Downarrow \langle R, tr \rangle$. In the latter case, an error is thrown and nothing is returned. □

A.2 Proofs for Theorem 5.2

Proof. By the definition of the finalization function at the end of the execution, we immediately filter out data with the remaining tags. Thus, if the program returns valid data, then following the soundness theorem, due to invocation of the sink function, we know that $\forall c \in R, \mathcal{E}(c) \equiv \mathbf{L}$, meaning that the query trace is equivalent to a computation that involves no secret. This completes the proof. □

B Policies Used in the Case Studies

The policies used in the case study of Section 7 are presented in Table 3.

C Query 3 from the TPC-H Benchmark

We present the code of Query 3 from the TPH-H benchmark used in our microbenchmark in Listing 1.

Listing 1: Code of Query 3 in the TPC-H Benchmark

```
SELECT
    l_orderkey,
    sum(l_extendedprice * (1 - l_discount)) as
     revenue,
    o_orderdate,
    o_shippriority
FROM
    customer,
    orders,
    -- We added a self-union for `lineitem`.
    (SELECT * FROM lineitem)
        UNION ALL
    (SELECT * FROM lineitem)
WHERE
    c_mktsegment = 'BUILDING'
    AND c_custkey = o_custkey
    AND l_orderkey = o_orderkey
    AND o_orderdate < date '1995-03-15'
    AND l_shipdate > date '1995-03-15'
GROUP BY
    l_orderkey,
    o_orderdate,
    o_shippriority
ORDER BY
    revenue desc,
    o_orderdate
LIMIT 20;
```

Dataset	Privacy Policy in Natural Language
Chronic illness [3]	1. People whose `age` > 89 must be generalized (Safe Harbor). 2. `user_id` should be removed (Safe Harbor). 3. `trackable_*` should only be aggregated with one of `MAX, MIN, SUM, COUNT`, and size should be greater than 20 (NIH-like policy).
Healthcare dataset [8]	1. `name` should be removed (Safe Harbor). 2. `medical_condition` must be aggregated with one of `MAX, MIN, SUM, COUNT`. (Common aggregate requirements)

Table 3: The dataset and its corresponding privacy policies used in case studies.