PrivGuard: Privacy Regulation Compliance Made Easier

Lun Wang, *UC Berkeley;* Usmann Khan, *Georgia Tech;* Joseph Near, *University of Vermont;* Qi Pang, *Zhejiang University;* Jithendaraa Subramanian, *NIT Tiruchirappalli;* Neel Somani, *UC Berkeley;* Peng Gao, *Virginia Tech;* Andrew Low and Dawn Song, *UC Berkeley*

https://www.usenix.org/conference/usenixsecurity22/presentation/wang-lun

This paper is included in the Proceedings of the 31st USENIX Security Symposium.

August 10–12, 2022 • Boston, MA, USA

978-1-939133-31-1

Open access to the Proceedings of the 31st USENIX Security Symposium is sponsored by USENIX.

PRIVGUARD: Privacy Regulation Compliance Made Easier

Lun Wang
UC Berkeley

Usmann Khan
Georgia Tech

Joseph Near
University of Vermont

Qi Pang
Zhejiang University

Jithendaraa Subramanian
NIT Tiruchirappalli

Neel Somani
UC Berkeley

Peng Gao
Virginia Tech

Andrew Low
UC Berkeley

Dawn Song
UC Berkeley

Abstract

Continuous compliance with privacy regulations, such as GDPR and CCPA, has become a costly burden for companies from small-sized start-ups to business giants. The culprit is the heavy reliance on human auditing in today's compliance process, which is expensive, slow, and error-prone. To address the issue, we propose PRIVGUARD, a novel system design that reduces human participation required and improves the productivity of the compliance process. PRIVGUARD is mainly comprised of two components: (1) PRIVANALYZER, a static analyzer based on abstract interpretation for partly enforcing privacy regulations, and (2) a set of components providing strong security protection on the data throughout its life cycle. To validate the effectiveness of this approach, we prototype PRIVGUARD and integrate it into an industrial-level data governance platform. Our case studies and evaluation show that PRIVGUARD can correctly enforce the encoded privacy policies on real-world programs with reasonable performance overhead.

1 Introduction

With the advent of privacy regulations such as the EU's General Data Protection Regulation (GDPR) and California Consumer Privacy Act (CCPA), unprecedented emphasis is put on the protection of user data. This is a positive development for data subjects, but presents major challenges for compliance. Today's compliance paradigm relies heavily on human auditing, and is problematic in two aspects. First, it is an expensive process to hire and train data protection personnel and rely on manual effort to monitor compliance. According to a report from Forbes [12], GDPR cost US Fortune 500 companies $7.8 Billion as of May 25th, 2018. Another report from DataGrail [4] shows that 74% of small- or mid-sized organizations spent more than $100,000 to prepare for continuous compliance with GDPR and CCPA. Second, human auditing is slow and error-prone. The inefficiency of compliance impedes the effective use of data and hinders productivity. Errors made by compliance officers can harm data subjects and result in legal liability.

An ideal solution would enable data curators to easily ensure fine-grained compliance with minimal human participation and quickly adapt to new changes in privacy regulations. A significant amount of academic work seeks to address this challenge [2, 30, 32, 42, 43, 51, 53, 55, 57]. The European ICT PrimeLife project proposes to encode regulations using Primelife Policy Language (PPL) [55] and enforce them by matching the policies with user-specified privacy preferences and triggering obligatory actions when detecting specific behaviors. A-PPL [30] extends the PPL language by adding accountability rules. These two pioneering works play important roles in the exploration of efficient policy compliance. However, as they focus on Web 2.0 applications, they provide limited support for fine-grained privacy requirement compliance in complex data analysis tasks. The SPECIAL project [2] partly inherits the design of the PPL project and, as a result, suffers from similar limitations. The closest to our work is by Sen et al. [53], which proposed a formal language (LEGALEASE) for privacy policies and a system (GROK) to enforce them. However, GROK uses heuristics to help decide whether the analysis process is compliant with a policy and human auditing is required to catch false-negatives. Thus, effective compliance with privacy regulations at scale remains an important challenge.

PRIVGUARD: Facilitating Compliance. This paper describes a principled data analysis framework called PRIVGUARD to reduce human participation in the compliance process. PRIVGUARD works in a five-step pipeline under the protection of cryptographic tools and trusted execu-

tion environments (TEEs).

First, data protection officers (DPOs), legal experts, and domain experts collaboratively translate privacy regulations into a machine-readable policy language. The translation process is application-specific and requires domain-specific knowledge in both the application and the privacy regulation (*e.g.* mapping legal concepts to concrete fields.). The encoded policy is referred to as the *base policy*. Encoding the base policy is the step with the most human effort in PRIVGUARD's workflow.

Second, before the data is collected, the data subjects are aided by a client-side API to specify their *privacy preferences*. They can either directly accept the base policy or add their own privacy requirements. The privacy preferences are collected together with the data.

Third, data analysts submit programs to analyze the collected data. Analysts are required to submit a corresponding *guard policy* no weaker than the base policy along with their program. Only data with policy no stronger than the guard policy can be used.

Fourth, our proposed static analyzer, PRIVANALYZER, examines the analysis program to confirm its compliance with the guard policy. At the same time, the subset of the data whose privacy preferences are no stronger than the guard policy will be loaded to conduct the real analysis.

Finally, depending on the output of PRIVANALYZER, the result will be either declassified to the analyst or guarded by the remaining unsatisfied privacy requirements (called a *residual policy*).

Extension of LEGALEASE: Encoding Policies. PRIVGUARD is designed to be compatible with many machine-readable policy languages such as [30, 55]. In this work, we instantiate our implementation with LEGALEASE [53] due to its readability and extensibility. We extend LEGALEASE [53] by providing new attribute types, including attributes requiring the use of privacy-enhancing technologies like differential privacy.

PRIVANALYZER: Enforcing Policies. The core component of PRIVGUARD is PRIVANALYZER, a static analyzer checking the compliance of an analysis program with a privacy policy. PRIVANALYZER performs static analysis of the programs submitted by analysts to check their compliance with the corresponding guard policies.

In contrast to previous approaches relying on access control [28] or manual verification [2, 36, 53], PRIVANALYZER is a novel policy enforcement mechanism based on abstract interpretation [49]. PRIVANALYZER does not rely on heuristics to infer policy or program semantics, and provides provable soundness for some properties. PRIVANALYZER examines only the program and the policy (not the data), so the use of PRIVANALYZER does not reveal the content of the data it protects. Our approach works for general-purpose programming languages, including those with complex control flow, loops, and imperative features. Thus, PRIVANALYZER is able to analyze programs as written by analysts—so analysts do not need to learn a new programming language or change their workflows. We instantiate our implementation with Python, one of the most widely used programming languages for data analysis.

We implemented PRIVANALYZER in about 1400 lines of Python and integrated it in an industrial-level data governance platform to prototype PRIVGUARD. We evaluated the prototype experimentally on 23 open-source Python programs that perform data analytics and machine learning tasks. The selected programs leverage popular libraries like Pandas, PySpark, TensorFlow, PyTorch, Scikit-learn, and more. Our results demonstrate that PRIVGUARD is scalable and capable of analyzing unmodified Python programs, including programs that make extensive use of external libraries.

Contributions. In brief, this paper makes the following contributions.

- We propose PRIVGUARD, a novel framework for privacy regulation compliance minimizing human effort.
- We propose PRIVANALYZER, a static analyzer based on abstract interpretation for enforcing privacy policies on unmodified analysis programs.
- We implemented PRIVANALYZER for LEGALEASE and Python in about 1400 lines of Python. Our implementation supports commonly-used analysis libraries such as Pandas, Scikit-learn, *etc*.
- We prototyped PRIVGUARD by integrating PRIVANALYZER in PARCEL, an industrial-level data governance platform. We simulated the execution of PRIVGUARD with up to one million clients and the results show that PRIVGUARD incurs about two-minute overhead when dealing with one million clients.

2 PRIVGUARD Overview

In this section, we outline the design and implementation of PRIVGUARD. We first walk through PRIVGUARD using a toy example and then introduce the system design and implementation. As last, the threat model and the security of PRIVGUARD are discussed.

2.1 A Toy Example

We use a toy example to demonstrate the workflow of PRIVGUARD, which also allows us to present the main components used. A company launches a mobile application and collects user data to help make informed

business decisions. To facilitate compliance with privacy regulations, the company deploys PRIVGUARD to protect the collected data.

First, the DPOs, legal experts, and domain experts encode two requirements in the base policy: (1) minors' data shall not be used in any analysis; (2) any statistics on the data shall be protected using differential privacy.

Second, the privacy preferences are collected from the data subjects together with the data. Some data subjects (Group 1) trust the company and directly accept the base policy. Some (Group 2) are more cautious and want their zip codes to be redacted before analysis. The others (Group 3) do not trust the company and do not want their data to be used for purposes except for legitimate interest.

Third, a data analyst wants to survey the user age distribution. It specifies a guard policy, that besides the base policy, zip codes shall not be used in the analysis either. The analyst submits a program calculating the user age histogram to PRIVGUARD. She remembers to filter out all the minor information and redact the zip code field but forgets to protect the program with differential privacy.

Fourth, PRIVGUARD uses PRIVANALYZER to examine the privacy preferences and loads data of Group 1 and 2 into the TEE as their privacy preferences are no stricter than the guard policy. PRIVGUARD runs the program and saves the resulting histogram. However, after examining the program and the guard policy, PRIVGUARD finds that the program fails to protect the histogram with differential privacy. Thus, the histogram is encrypted, dumped to the storage layer and guarded by a *residual policy* indicating that differential privacy should be applied before the result can be declassified.

Lastly, PRIVGUARD outputs the *residual policy* to the analyst. The analyst, after checking the residual policy, submits a program which adds noise to the histogram to satisfy differential privacy. PRIVGUARD then decrypts the histogram, loads it into TEE, and executes the program to add noise to it. This time, PRIVGUARD finds that all the requirements in the guard policy are satisfied, so it declassifies the histogram to the analyst.

2.2 System Design

Base Policy Encoding. Encoding the base policy is the step with the most human participation in PRIVGUARD's workflow. The base policy should be designed collaboratively by DPOs, legal experts, and domain experts to accurately reflect the minimum requirements of the privacy regulation. Note that only one base policy is needed for each data collection and can be reused throughout all the analyses on the data. The purpose of the base policy is

```
ALLOW ROLE
    Physician
ALLOW SCHEMA
    HealthInformation
  AND FILTER age < 90
  AND REDACT zip
```

(a) General encoding.

```
ALLOW ROLE
    Physician
ALLOW SCHEMA
    SerumCholestoral
  AND FILTER age < 90
```

(b) Concrete encoding.

Figure 1: Encoding of several HIPAA requirements.

two-fold. First, the text version of the base policy is to be presented to data subjects as the minimum privacy standard before they opt in the data collection. Second, the data analysts need to understand the base policy before conducting analysis. If their analysis satisfies a stricter privacy standard, they can specify their own guard policy to take advantage of more user data.

We demonstrate the encoding process using a subset of the HIPAA safe harbor method[1] (Figure 1). The DPOs and legal experts first encode the regulation in a general way without considering concrete data format. As shown in Figure 1a, the first clause (line 1-2) allows the patient's physician to check his or her data. The second clause (lines 3-7) represents some safe harbor requirements: health information may be released if the subject is under 90 years old and the zip codes are removed. Then the DPOs and domain experts map the encoding to a concrete data collection by introducing real schemas and removing unnecessary requirements. For example, in Figure 1b, HealthInformation is replaced with a concrete column name in the dataset, SerumCholestoral, and the last requirement is removed as the dataset does not contain zip codes.

Data & Privacy Preference Collection. Besides the base policy, the data subjects can also specify additional privacy preferences to exercise their rights to restrict processing. These privacy preferences are sent to the data curator along with the data, where they will be kept together in the storage layer. To defend against attacks during transmission and storage, the data is encrypted before sent to the data curator. The decryption key is delegated to a key manager for future decryption (Section 2.3).

A natural question is "how much expertise is needed to specify privacy preferences in LEGALEASE?" Sen et al. [53] conducted a survey targeting DPOs and found that the majority were able to correctly code policies after training. To complement their survey and better understand how much expertise is needed, we conducted an online survey targeting general users without train-

[1] Recent research has shown that the approach prescribed in HIPAA does not really protect the privacy of individuals. In the future, we expect that many data subjects will add a PRIVACY attribute requiring the use of a provable privacy technology like differential privacy.

ing. The survey reveals two interesting facts: (1) there is a significant positive correlation between the difficulty of understanding and encoding LEGALEASE policies and the user's familiarity with other programming languages; (2) most users cannot correctly understand privacy techniques such as differential privacy without training. According to these observations, we strongly recommend the users without programming experience directly accept the base policy instead of encoding their own. Although out of scope, we deem it important future direction to simplify privacy preference specification by developing more user-friendly UI and translation tools. The details of the survey are deferred to Appendix B.

Analysis Initialization. To initialize an analysis task, the analyst needs to submit (1) the analysis program, and (2) a guard policy, to PRIVANALYZER. A guard policy should be no weaker than the base policy to satisfy the minimum privacy requirements. The stricter the guard policy is, the more data can be used for analysis.

PRIVANALYZER Analysis. After receiving the submission, PRIVANALYZER will load the privacy preferences from the storage layer and compare them with the guard policy. Only the data with preferences no stricter than the guard policy will be loaded into the TEE, decrypted using keys from the key manager, and merged to prepare for the real analysis. Meanwhile, PRIVANALYZER will (1) check that the guard policy is no weaker than the base policy and (2) then examine the guard policy and the program to generate the residual policy. To make sure the static analysis runs correctly, PRIVANALYZER is protected inside a trusted execution environment (TEE).

The compliance enforcement actually hinges on three checks: (1) the guard policy is no weaker than the base policy; (2) only data with privacy preferences no stronger than the guard policy is used; (3) the guard policy should be satisfied before declassification. For (3), the guard policy can be satisfied either by a single program or by multiple programs applied sequentially on the data. This design endows PRIVGUARD with the ability to enforce privacy policies in multi-step analyses.

Execution & Declassification. After PRIVANALYZER finishes its analysis, the submitted program will be executed with the decrypted data inside the TEE. If the residual policy generated in the previous step is empty, then the result can be declassified to the analyst. Otherwise, the output will be encrypted again and stored in the storage layer protected by the residual policy.

Attentive readers might ask "why does PRIVGUARD not directly reject programs that fail to comply with the guard policy?" The design choice is motivated by two considerations. First, it is not always possible to get an empty residual policy when the guard policy contains ROLE or PURPOSE attributes. These attributes will be satisfied by human auditing after the real data analysis. Second, PRIVGUARD is designed to be compatible with multi-step analysis, a common case in real-world product pipelines. In multi-step analysis, it is likely that privacy requirements are satisfied in different steps.

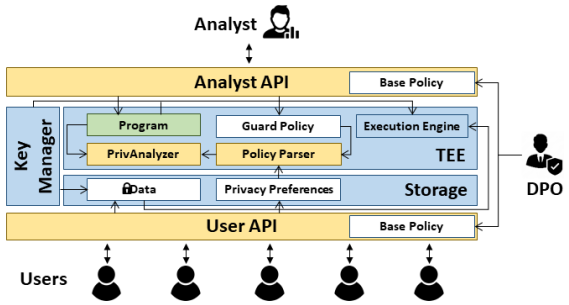


Figure 2: PRIVGUARD prototype infrastructure. White: data/policies; green: analysis programs; blue: off-the-shelf components; yellow: newly-designed components.

2.3 System Security

In this section, we present the threat model and demonstrate how to secure PRIVGUARD under the threat model.

Threat Model. Our setting involves four parties - (1) data subjects (*e.g.* users), (2) a data curator (*e.g.* web service providers, banks, or hospitals), (3) data analysts (*e.g.* employees of the data curator), and (4) untrusted third parties (*e.g.* external attackers). Data is collected from data subjects, managed by the data curator, and analyzed by data analysts. Both the data subjects and the data curator would like to comply with privacy regulations to either protect their own data or avoid legal or reputational risk. The data analysts, however, are honest but reckless, and might unintentionally write programs that violate privacy regulations. The only way that a data analyst can interact with the data is to submit analysis programs and check the output. The untrusted third parties might actively launch attacks to steal the data or interfere with the compliance process. A concrete example is the cloud provider which hosts a small company's service or data. We protect data confidentiality and execution integrity from third parties under the following two assumptions. First, we assume that the untrusted third parties cannot submit analysis programs to PRIVANALYZER or compromise insiders to do so. Second, we assume that the untrusted third parties fit in the threat model of the chosen TEE so that they cannot break the security guarantee of the TEE.

Security Measure. PRIVGUARD takes the following measures to defend against untrusted third parties and establish a secure workflow under the above threat model. First, data is encrypted locally by the data subjects before transmitted to the data curator. The decryption key is delegated to the key manager so no one can touch the data intentionally or carelessly without asking the key manager or the data subject for the decryption key. To bind data and policy in an immutable way, the encrypted data contains a hash value of the corresponding policy. Second, all transmission channels satisfy transport layer security standards (TLS 1.3). Third, PRIVANALYZER is run inside a TEE to guarantee the integrity of the static analysis. The key manager can attest remotely to confirm that PRIVANALYZER correctly examines the program and the policies before issuing the decryption key. Fourth, data decryption and analysis program execution are protected inside the TEE as well.

Security of PRIVGUARD against untrusted third-parties is based on the following sources of trust. First, confidentiality and integrity of data are preserved inside TEE and TLS channels. Second, the integrity of code execution is preserved inside the TEE. Remote attestation can correctly and securely report the execution output. Third, the key manager is completely trusted such that the confidentiality of decryption keys is preserved. The design of trusted key managers is orthogonal to the focus of the paper. Potential solutions include a key manager inside TEE or a decentralized key manager [44].

3 PRIVANALYZER: Static Analysis for Enforcing Privacy Policies

This section describes PRIVANALYZER, a static analyzer for enforcing the privacy policies tracked by PRIVGUARD. We first review the LEGALEASE policy language [53], which we use to encode policies formally, then describe how to statically enforce them. The formal model is deferred to Appendix A.

3.1 Background & Design Challenges

LEGALEASE is one example of a growing body of work that has explored formal languages for encoding privacy policies [31, 32, 36, 39, 42, 43, 51, 53, 56, 57]. A complete discussion of related work appears in Section 5. We adopt LEGALEASE to express PRIVGUARD policies due to its expressive power, formal semantics, and extensibility.

Sen et al. [53] developed a system called GROK that combines static and dynamic analyses to enforce LEGALEASE policies. GROK constructs a data dependency graph which encodes *all* flows of sensitive data, then applies a set of inference rules to check that each node in the graph satisfies the policy. GROK combines analysis of system logs with limited static analysis to construct the graph.

$$
\begin{array}{rcllll}
\mathit{attr} & \in & \textit{ROLE}, \textit{SCHEMA}, \textit{PRIVACY}, \textit{FILTER}, & & & \\
 & & \textit{REDACT}, \textit{PURPOSE} & & & \\
A & \in & \text{attribute} & ::= & \mathit{attr}\ \mathit{attrValue} & \\
C & \in & \text{policy clause} & ::= & A \mid A\ \textbf{AND}\ C \mid A\ \textbf{OR}\ C & \\
P & \in & \text{policy} & ::= & (\textbf{ALLOW}\ C)^{+} &
\end{array}
$$

Figure 3: Policy language surface syntax

The GROK approach presents two challenges. First, the approach is a heuristic: it examines syntactic properties of the program and individual executions of the program (via system logs), and thus may miss policy violations due to implicit flows [37, 48, 50, 59]. Second, the GROK approach requires making the entire dataflow graph explicit; in large systems with many data flows, constructing this graph may be intractable.

PRIVANALYZER is designed as an alternative to address both challenges. It uses static analysis based on abstract interpretation instead of GROK's heuristic analysis and avoids making the dataflow graph explicit by constructing composable *residual policies*.

3.2 Policy Syntax & Semantics

PRIVANALYZER enforces privacy policies specified in LEGALEASE [53], a framework for expressing policies using *attributes* of various types. Attributes are organized in *concept lattices* [3], which provide a partial order on attribute values. We express policies according to the grammar in Figure 3 (a slightly different syntax from that of LEGALEASE). A policy consists of a top-level **ALLOW** keyword followed by *clauses* separated by **AND** (for conjunction) and **OR** (for disjunction). For example, the following simple policy specifies that doctors or researchers may examine analysis results, as long as the records of minors are not used in the analysis:

ALLOW (*ROLE* Doctor **OR** *ROLE* Researcher)
AND *FILTER* age >= 18

Sen et al. [53] define the formal semantics of LEGALEASE policies using a set of inference rules and the partial ordering given by each attribute's concept lattice. We take the same approach, but use a new attribute framework based on *abstract domains* [49] instead of concept lattices. Our approach enables PRIVPOLICY to encode policies with far more expressive requirements, like row-based access control and the use of privacy-enhancing technologies as described below.

Attributes. Attributes are the basic building blocks

in LEGALEASE. Sen et al. [53] describe a set of useful attributes. We extend this set with two new ones: *FILTER* encodes row-based access control requirements, and *PRIVACY* requires the use of privacy-enhancing technologies.

Role. The *ROLE* attribute controls *who* may examine the contents of the data. Roles are organized into partially ordered hierarchies. A particular individual may have many roles, and a particular role specification may represent many individuals. For example, the *doctor* role may represent doctors with many different specialties. The following policy says that only individuals with the role Oncologist may examine the data it covers:

ALLOW *ROLE* Oncologist

Schema. The *SCHEMA* attribute controls *which columns* of the data may be examined. For example, the following policy allows oncologists to examine the *age* and *condition* columns, but no others:

ALLOW *ROLE* Oncologist
AND *SCHEMA* age, condition

Privacy. The *PRIVACY* attribute controls *how* the data may be used, by requiring the use of privacy-enhancing technologies. As a representative sample of the spectrum of available mechanisms, our implementation supports the following (with easy additions): (1) De-identification (or pseudonymization); (2) Aggregation; (3) k-Anonymity [54]; (4) ℓ-diversity [41]; (5) t-closeness [40]; (6) Differential privacy [33,34]. For example, the following policy allows oncologists to examine the *age* and *condition* columns under the protection of differential privacy with certain privacy budget:

ALLOW *ROLE* Oncologist
AND *SCHEMA* age, condition
AND *PRIVACY* DP (1.0, 1e-5)

Filter. The *FILTER* attribute allows policies to specify that certain data items must be excluded from the analysis. For example, the following policy says that oncologists may examine the age and condition of individuals over the age of 18 with differential privacy:

ALLOW *ROLE* Oncologist
AND *SCHEMA* age, condition
AND *PRIVACY* DP (1.0, 1e-5)
AND *FILTER* age > 18

Redact. The *REDACT* attribute allows policies to require the partial or complete redaction of information in a column. For example, the following policy requires analysis to redact the last 3 digits of ZIP codes (e.g. by replacing them with stars). The (2 :) notation is taken from Python's slice and indicates the substring between the third character and end of the string.

ALLOW *ROLE* Oncologist
AND *SCHEMA* age, condition
AND *PRIVACY* DP (1.0, 1e-5)
AND *FILTER* age > 18
AND *REDACT* zip (2 :)

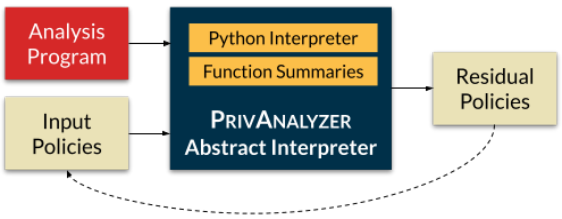


Figure 4: PRIVANALYZER Overview. PRIVANALYZER inputs an analysis program and policies, and produces residual policies; it can be applied repeatedly (dashed line) for multi-step analyses.

Purpose. The *PURPOSE* attribute allows the policy to restrict the purposes for which data may be analyzed. For example, the following policy allows the use of age and medical condition for public interest purposes with all the above requirements:

ALLOW *ROLE* Oncologist
AND *SCHEMA* age, condition
AND *PRIVACY* DP (1.0, 1e-5)
AND *FILTER* age > 18
AND *REDACT* zip (2 :)
AND *PURPOSE* PublicInterest

3.3 PRIVANALYZER Overview

PRIVANALYZER performs its static analysis via *abstract interpretation* [49], a general framework for sound analysis of programs. Abstract interpretation works by running the program on *abstract values* instead of concrete (regular) values. Abstract values are organized into *abstract domains*: partially ordered sets of abstract values which can represent all possible concrete values in the programming language. An abstract value usually represents a specific *property* shared by the concrete values it represents. In PRIVANALYZER, abstract values are based on the abstract domains described earlier.

Our approach to static analysis is based on a novel instantiation of the abstract interpretation framework, in which we encode *policies as abstract values*. The approach is summarized in Figure 4. The use of abstract interpretation allows us to construct the static analysis systematically, ensuring its correspondence with the intended semantics of attribute values.

Analyzing Python Programs. The typical approach to abstract interpretation is to build an *abstract interpreter* that computes with abstract values. For a complex, general-purpose language like Python, this approach requires significant engineering work. Rather than building an abstract interpreter from scratch, we *re-use the stan-*

dard Python interpreter to perform abstract interpretation. We embed abstract values with attached privacy policies as Python objects and define operations over abstract values as methods on these objects.

For example, the Pandas library defines operations on concrete dataframes; PRIVANALYZER defines the AbsDataFrame class for *abstract* dataframes. The AbsDataFrame class has the same interface as the Pandas DataFrame class, but its methods are redefined to compute on abstract values with attached policies. We call the redefined method a *function summary*, since it summarizes the policy-relevant behavior of the original method. For example, the Pandas indexing function __getitem__ is used for filtering, so PRIVANALYZER's function summary for this function removes satisfied *FILTER* attributes from the policy.

```
def __getitem__(self, key):
    ......
    If isinstance(key, AbsIndicatorSeries):
        # ' runFilter ' removes satisfied FILTER attributes
        newPolicy = self.policy.runFilter (...)
        return Dataframe(..., newPolicy)
    ......
```

Multi-step Analyses & Residual Policies. As shown in Figure 4, the output of PRIVANALYZER is a *residual policy*. A residual policy is a new policy for the program's concrete output—it contains the requirements *not yet satisfied* by the analysis program. For a multi-step analysis, each step of the analysis can be fed into PRIVANALYZER as a separate analysis program, and the residual policies in the previous step become the input policies for the next step. PRIVANALYZER is *compositional*: if multiple analyses are merged together into a single analysis program, then the final residual policy PRIVANALYZER returns for the multi-step analysis will be at least as restrictive as the one for the single-step version. The use of residual policies in PRIVGUARD enables compositional analyses without requiring explicit construction of a global dataflow graph, addressing the challenge of GROK [53] mentioned earlier.

Handling libraries. Scaling to large programs is a major challenge for many static analyses, including abstract interpreters. Libraries often present the biggest challenge, since they tend to be large and complex; it may be impossible to analyze even a fairly small target program if the program depends on a large library. This is certainly true in our setting (Python programs for data processing), where programs typically leverage large libraries like pandas (327,007 lines of code), scikit-learn (184,144 lines of code), PyTorch (981,753 lines of code) and Tensorflow (2,490,119 lines of code). Worse, many libraries are written in a mix of languages (e.g. Python and C/C++) for performance reasons, so analysis for each one of these languages would be needed.

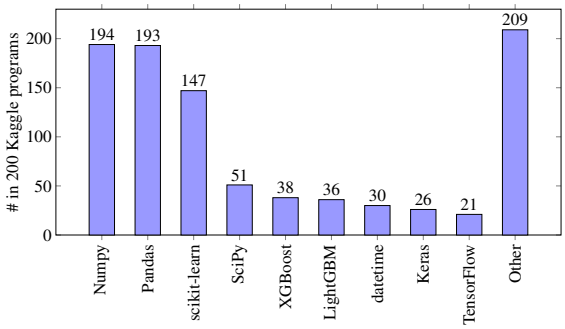


Figure 5: Python library frequency statistics. We summarized the top frequently used libraries.

Our solution is to develop *specifications* of the abstract functionality of these libraries, like the AbsDataFrame example shown earlier, in the form of function summaries. We use the function summaries during analysis instead of the concrete implementation of the library itself. This approach allows PRIVGUARD to enforce policies even for analysis programs that leverage extremely large libraries written in multiple languages.

Our approach for handling libraries requires a domain expert with knowledge of the library to implement its specification. In our experience, the data science community has largely agreed upon a core set of important libraries which are commonly used (e.g. NumPy, pandas, scikit-learn, etc.), so providing specifications for a small number of libraries is sufficient to handle most programs. To validate the conjecture empirically, we randomly selected 200 programs from the Kaggle platform and counted the libraries they use (Figure 5). The results confirmed that most data analysis programs use similar libraries. We have already implemented specifications for the most frequently used libraries (Section 4). Fortunately, the abstract behavior for a library function tends to be simpler than its concrete behavior. We have implemented 380 function summaries mainly for Numpy, Pandas, and scikit-learn and are actively working on adding more function summaries for various libraries.

We require correct specifications to rigorously enforce privacy policies. An illustrative example of the importance of correctly implementing specifications is the renaming function. Cunning inside attackers may want to bypass the static analysis by renaming sensitive columns. A correct specification which renames the columns in both the schema and the privacy clauses should mitigate such attacks. To mitigate risks due to such errors, function summaries should be open-sourced so the com-

munity can help check their correctness.

Comparison with dynamic approaches. Our choice of a static analysis for PRIVANALYZER is motivated by two major advantages over dynamic approaches: (1) the ability to handle implicit data flows, and (2) the goal of adding minimal run-time overhead. The ability to detect implicit flows is a major advantage of static analysis systems [37, 48, 50, 59], including PRIVANALYZER. Unlike dynamic approaches, PRIVANALYZER cannot be defeated by complex control flow designed to obfuscate execution. For example, the data subject might specify the policy **ALLOW** *REDACT* name (1 :), which requires redacting most of the name column. An analyst might write the following program:

```
if data.name == 'Alice':
  return 1
else:
  return 2
```

This program clearly violates the policy, even though it does not return the value of data.name directly. This kind of violation is due to an *implicit flow* of the name column to the return value. A return value of 1 allows the analyst to confirm with certainty that the data subject's name is Alice. This kind of implicit flow presents a significant challenge for dynamic analyses, because dynamic analyses *only execute one branch* of each conditional, and can make no conclusions about the branch *not* taken. A dynamic analysis must either place significant restrictions on the use of sensitive values in conditionals, or allow for unsoundness due to implicit flows.

Static analyzers like PRIVANALYZER, on the other hand, can perform a worst-case analysis that inspects *both* branches. PRIVANALYZER's analysis executes both branches with the abstract interpreter and returns the *worst-case* outcome of both branches. For loops with no bound on the number of iterations, the analysis results represent the worst-case outcome, no matter how many iterations execute at runtime. This power comes at the expense of a potential lack of precision—the analysis may reject programs that are actually safe to run. Our evaluation suggests, however, that PRIVANALYZER analysis is sufficiently precise for programs that perform data analyses. Static analysis tools like PRIVANALYZER do not require the policy specification to be aware of implicit flows as it combines both types of flows in its results.

3.4 PRIVANALYZER by Example

The input to PRIVANALYZER is a single analysis program, plus all of the policies of the data files it processes. For each of the program's outputs, PRIVANALYZER produces a residual policy. After running the analysis, PRIVGUARD will attach each of these residual policies to the appropriate concrete output.

PRIVANALYZER works by performing abstract interpretation, where the inputs to the program are abstract values containing representations of the associated policies. The output of this process is a set of abstract values containing representations of the residual policies.

A complete example of this process is summarized in Figure 6. The analysis is a Python program adapted from an open-source analysis submitted to Kaggle [9]. Important locations in the program are labeled with numbers (e.g. ①) and the associated residual policy PRIVANALYZER computes *at that program location*.

Reading Data into Abstract Values. The program begins by reading in a CSV containing the sensitive data (Fig. 6, ①). PRIVANALYZER's abstract interpretation library redefines read_csv to return an *abstract dataframe* containing the policy associated with the data being read. At location ①, the variable df thus contains the full policy for the program's input. In this example, the policy allows the program to use the "age," "credit," and "duration" columns, requires filtering out the data of minors, and

Mixing Concrete and Abstract Values. The next part of the program defines some constants, which PRIVANALYZER represents with concrete values lacking attached policies. Then, the program drops one of the input columns (Fig. 6, ②); this action does not change the policy, because the columns being dropped are not sufficient to satisfy any of the policy's requirements, so the df variable is unchanged.

Satisfying *FILTER* Requirements. The next statement (Fig. 6, ③) performs a filtering operation. The PRIVANALYZER library redefines this operation to eliminate the appropriate *FILTER* requirements from the policy; since this filtering operation removes individuals below the age of 25, it satisfies the *FILTER* requirement in the policy, and the new value of df is an abstract dataframe whose policy does not have this requirement.

Handling Loops. The next part of the program contains a **for** loop (Fig. 6, ④). Loops are traditionally a big challenge for static analyzers. PRIVANALYZER is designed to take advantage of *loops over concrete values*, like this one, by simply executing them concretely. PRIVANALYZER can also handle loops over abstract values (described later in this section), but these were relatively rare in our case studies.

Libraries and Black-box Operations. The next pieces of code (Fig. 6, ⑤ and ⑥) first take the log of each feature, then scale the features. Both of these operations impact the *scale* of feature values. After these operations,

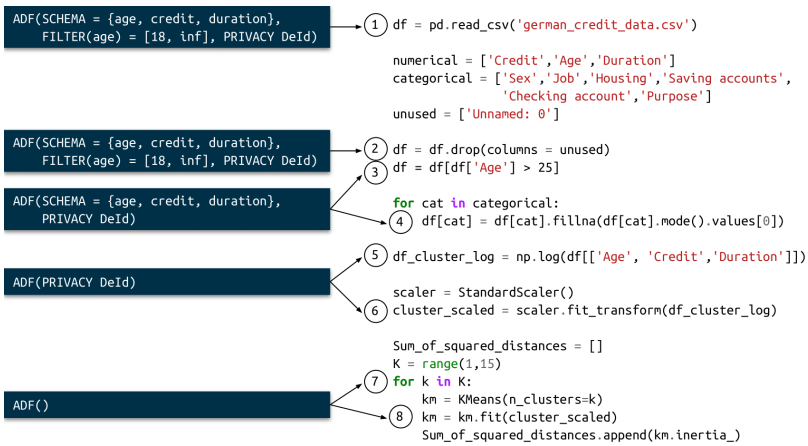


Figure 6: Example Abstract Interpretation with PRIVANALYZER.

it becomes impossible to satisfy policy requirements like *FILTER*, because the original data values have been lost. For lossy operations like these, which we call *black-box* operations (detailed below), PRIVANALYZER is designed to raise an alarm if value-dependent requirements (like *FILTER*) remain in the policy.

Training a Model. The final piece of code (Fig. 6, ⑦ and ⑧) uses the KMeans implementation in scikit-learn to perform clustering. We summarize this method to specify that it satisfies de-identification requirements in the policy. The result in our example is an empty residual policy, which would allow the analyst to view the results.

3.5 Challenging Language Features

We now address the approach taken in PRIVANALYZER for several challenging language features.

Conditionals. Conditionals depending on abstract values require the abstract interpreter to run both branches and compute the upper bound on both results. Since Python does not allow redefining `if` statements, we add a pre-processing step to PRIVGUARD that transforms conditionals by running *both* branches.

Loops. Loops are traditionally the most challenging construct for abstract interpreters to handle. Fortunately, loops in Python programs for data analytics often fall into restricted classes, like the ones in the example of Figure 6. Both loops in this example are over constant values—so our abstract interpreter can simply run each loop body as many times as the constant requires.

Loops over abstract values are more challenging, and the simple approach may never terminate. To address this situation, we define a *widening* operator [49] for each of the abstract domains used in PRIVANALYZER. Widening operators force the loop to arrive at a fixpoint; in our example, widening corresponds to assuming the loop body will be executed over the whole dataframe.

Aliasing. Another challenge for abstract interpretation comes from the issue of *aliasing*, where two variables point to the same value. Sometimes, it is impossible for the analysis to determine *what* abstract value a variable references. In this case, it is also impossible to determine the outcome of the side effects on the variable.

Our approach of re-using the existing Python interpreter helps address this challenge: in PRIVANALYZER, all variable references are evaluated concretely. In most cases, references are to concrete objects, so the analysis corresponds exactly to concrete execution. In a few cases, however, this approach leads to less precise analysis. For example, if a variable is re-assigned in both branches of a conditional, PRIVANALYZER must assume the *worst-case* abstract value (i.e. with the most restrictive policy) is assigned to the variable in both cases. This approach works well in our setting, where conditionals and aliasing are both relatively rare.

3.6 Attribute Enforcement

We now describe some attribute-specific details of our compliance analysis.

Schema, Filter, and Redact. The *SCHEMA*, *FILTER*, and *REDACT* attributes can be defined formally and compliance can be checked by PRIVANALYZER. In our implementation, relevant function summaries remove the attribute from the privacy policy if the library's concrete implementation satisfies the corresponding requirement. Our summaries thus implement abstract interpretation for these functions. Note that PRIVANALYZER assumes that functions without summaries do not satisfy *any* policy requirements. PRIVANALYZER is therefore *incomplete*:

some programs may be rejected (due to insufficient function summaries) despite satisfying the relevant policies.

Privacy. The *PRIVACY* attribute is also checked by PRIVANALYZER. Analysis programs can satisfy de-identification requirements by calling functions that remove identifying information (e.g. aggregating records or training machine learning models). Programs can satisfy k-Anonymity[2], ℓ-diversity[2], t-closeness or differential privacy requirements by calling specific functions that provide these properties. Our function summaries include representative implementations from the current literature: IBM Differential Privacy Library [38], K-Anonymity Library [16], and Google's Tensorflow Privacy library [26].

There are two subtleties when enforcing differential privacy attributes. First, programs that satisfy differential privacy also need to track the *privacy budget* [34]. By default, PRIVGUARD tracks a single global cumulative privacy cost (values for ε and δ) for each source of submitted data, and rejects new analysis programs after the privacy cost exceeds the budget amount. PRIVANALYZER reports the privacy cost of a single analysis program, allowing PRIVGUARD to update the global privacy cost. A single global privacy budget may be quickly exhausted in a setting with many analysts performing different analyses. One solution to this problem is to generate differentially private synthetic data, which can be used in later analyses without further privacy cost. The High-Dimensional Matrix Mechanism (HDMM) [45] is one example of an algorithm for this purpose, used by the US Census Bureau to release differentially private data. In PRIVGUARD, arbitrarily many additional analyses can be performed on the output of algorithms like HDMM without using up the privacy budget. Another solution is fine-grained budgeting, at the record level (as in ProPer [35]) or a statically defined "region" level (as in UniTraX [47]). The first is more precise, but requires silently dropping records for which the privacy budget is exhausted, leading to biased results. Both approaches allow for continuous analysis of fresh data in growing databases (e.g. running a specific query workload every day on just the new data obtained that day). Second, to calculate privacy budget, PRIVGUARD initializes a variable to track the sensitivity of the pre-processing steps before the differentially private function. The pre-processing function summaries should manipulate the variable to specify their influence on the sensitivity. If such specification is absent in any function before the differentially private function, PRIVGUARD will throw a warning and recognize the differential privacy requirement as unsatisfied.

Role. *ROLE* attributes are enforced by authentication techniques such as password, 2-factor authentication , or even biometric authentication . In addition, *ROLE* attributes are also recorded in an auditable system log described in the next paragraph, and the analysts and the data curators will be held accountable for fake identities.

Purpose. The *PURPOSE* attribute is inherently informal. Thus, we take an accountability-based approach to compliance checking for purposes. Analysts can specify their purposes when submitting the analysis program, and may specify an invalid purpose unintentionally or maliciously. These purposes will be used by PRIVANALYZER to satisfy *PURPOSE* requirements. PRIVGUARD produces an audit log recording analysts, analysis programs, and claimed purposes. Thus, all the analysis happening in the system can be verified after the fact, and analysts can be held legally accountable for using invalid purposes.

4 Evaluation

The evaluation is designed to demonstrate that (1) PRIVANALYZER supports commonly-used libraries for data analytics and can analyze real-world programs, and (2) PRIVGUARD is lightweight and scalable. To demonstrate, we (1) test PRIVANALYZER using 23 real-world analysis programs drawn from the Kaggle contest platform, and (2) measure the overhead of PRIVGUARD using a subset of these programs. The results show that PRIVGUARD can correctly enforce PRIVPOLICY policies on these programs with reasonable performance overhead.

4.1 Experiment Setup

We implemented PRIVANALYZER in about 1400 lines of Python and integrated it in an industrial-level data governance platform, Parcel [1], to prototype PRIVGUARD. We instantiated our implementation with Inter Planetary File System (IPFS) for the storage layer, AES-256-GCM for the encryption algorithm, and AMD SEV for TEE.

To evaluate PRIVGUARD's static analysis on real-world programs, we collect analysis programs for 23 different tasks from Kaggle, one of the most well-known platforms for data analysis contests. These programs analyze sensitive data such as fraud detection [15] and transaction prediction [22]. We selected these programs as case studies to demonstrate PRIVGUARD's ability

[2] k-Anonymity and ℓ-diversity are vulnerable to disclosure attacks as pointed out in [40]. k-Anonymity is vulnerable to homogeneity and background knowledge attacks, and ℓ-diversity suffers from skewness and similarity attacks. We strongly encourage using t-closeness or differential privacy for stronger protection. PRIVGUARD provides weaker approaches only for compatibility purposes.

Index	Application Type	# Features	# Samples	# LoC	External libraries
1	Fraud Detection - Random [15]	435	590541	157	LightGBM, NumPy, pandas, scikit-learn
2	Fraud Detection - Top [15]	435	590541	157	LightGBM, NumPy, pandas, scikit-learn
3	Merchant Recommendation [11]	41	201917	199	LightGBM, NumPy, pandas, scikit-learn
4	Customer Satisfaction Prediction [21]	370	76020	104	NumPy, pandas, scikit-learn, XGBoost
5	Customer Transaction Prediction - Random [22]	201	200000	89	NumPy, pandas, scikit-learn
6	Customer Transaction Prediction - Top [22]	201	200000	89	NumPy, pandas, scikit-learn
7	Bank Customer Classification [6]	13	10000	276	NumPy, pandas, scikit-learn
8	Bank Customer Segmentation [7]	9	1000	75	NumPy, pandas, scikit-learn
9	Credit Risk Analysis [9]	9	1000	57	NumPy, pandas, sklearn
10	Bank Customer Churn Prediction [5]	13	10000	169	NumPy, pandas, SciPy, scikit-learn
11	Heart Disease Causal Inference [14]	14	303	83	NumPy, pandas, SHAP, scikit-learn
12	Classify Forest Categories [8]	52	1000	50	NumPy, pandas, PySpark
13	PyTorch-Simple LSTM [23]	45	1780832	178	NumPy, pandas, Keras, PyTorch
14	Tensorflow-Solve Titanic [25]	7	891	163	NumPy, pandas, scikit-learn, Tensorflow
15	Earthquake Prediction - Top [17]	2	1000	132	NumPy, pandas, tsfresh, librosa, pywt, SciPy
16	Display Advertising - Top [10]	41	1000	60	math
17	Fraud Detection - Top [24]	8	1000	106	NumPy, pandas, Keras, Tensorflow
18	Restaurant Revenue Prediction - Top [20]	42	1000	115	NumPy, pandas, FastAI, scikit-learn
19	NFL Analytics - Top [19]	18	1000	152	NumPy, pandas, SciPy, scikit-learn
20	NCAA Prediction - Top [13]	34	1000	561	NumPy, pandas, Pymc3
21	Home Value Prediction - Top [29]	58	1000	272	NumPy, pandas, sklearn, XGBoost
22	Malware Prediction - Top [18]	83	1000	194	NumPy, pandas
23	Web Traffic Forecasting - Top [27]	551	1000	346	NumPy, pandas

Table 1: Case Study Programs. # LoC = Lines of Code. Random suffix means the program is chosen randomly from the contest. Top-ranked suffix means the program is chosen from the top-ranked programs on the leaderboard.

to analyze real-world analysis programs and support commonly-used libraries. These case studies are chosen to be representative of the programs written by data scientists during day-to-day operations at many different kinds of organizations. We surveyed 100 kaggle programs randomly and found that approximately 85% programs are less than 300 lines of code (after removing blank lines). Correspondingly, our case studies range between 50 and 276 lines of code, total exactly 1600 lines of code and include randomly picked programs from Kaggle notebook and top-ranked programs on the contest leaderboard. As shown in Table 1, these programs use a variety of external libraries including widely used libraries like pandas, PySpark, Tensorflow, PyTorch, scikit-learn, and XGBoost.

As the first step of the evaluation, we use PRIVANALYZER to analyze the collected programs listed in Table 1. In the experiment, we manually designed an appropriate LEGALEASE policy for each program, and then attached them to each of the datasets. For each program, we recorded both the time running on the dataset and the time for PRIVANALYZER to analyze the program. We also manually checked that the analysis result output by PRIVANALYZER was correct. All the experiments were run on a Ubuntu 18.04 LTS server with 32 AMD Opteron(TM) Processor 6212 with 512GB RAM. The results appear in Table 2. As the second step of the evaluation, we picked 7 case studies with open-source datasets, ran them on the PRIVGUARD prototype, and measured the system overhead.

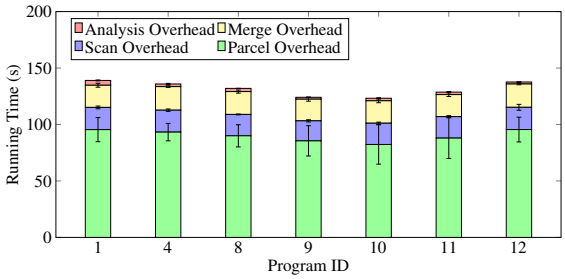


Figure 7: System overhead of PRIVGUARD prototype with one million simulated users.

4.2 Results

Support for Real-World Programs. Our experiment demonstrates PRIVGUARD's ability to analyze the kinds of analysis programs commonly written to process data in organizations. The results in Table 2 show that the static analysis took just a second or two for most programs, with three outliers taking 3.32, 4.78, and 6.84 seconds. The reason for the outliers is described in the next paragraph.

As in other abstract interpretation and symbolic execution frameworks, we expect that conditionals, loops, and other control-flow constructs will have a bigger effect on analysis time than program length. Fortunately, programs for data analytics and machine learning tend not to make

Index	Exec Time (s)	Analysis Time (s)	Overhead	Soundness
1	12571.01	1.41	$1.12e-2\%$	✓
2	19951.10	3.32	$1.66e-2\%$	✓
3	16762.65	1.18	$7.04e-3\%$	✓
4	151.72	1.22	$8.04e-1\%$	✓
5	17.14	1.08	6.30%	✓
6	33.71	0.96	2.84%	✓
7	32.66	2.03	6.22%	✓
8	86.82	2.19	2.52%	✓
9	4.65	1.01	21.72%	✓
10	295.16	1.29	$4.37e-1\%$	✓
11	3.99	1.00	25.06%	✓
12	1017.83	1.01	$1.00e-1\%$	✓
13	717.58	6.84	$9.53e-1\%$	✓
14	13.33	4.78	35.86%	✓
15	217.36	2.26	1.04%	✓
16	3.60	1.20	33.33%	✓
17	5.19	1.37	26.40%	✓
18	47.12	1.57	3.33%	✓
19	202.96	1.33	$6.55e-1\%$	✓
20	59.83	1.66	2.77%	✓
21	54.44	2.55	4.68%	✓
22	51.36	1.23	2.39%	✓
23	45.58	2.45	5.37%	✓

Table 2: Execution Time vs. Analysis Time. The index of case studies is the same as in Table 1.

extensive use of these constructs, especially compared to traditional programs. Instead, they tend to use constructs provided by libraries, like the query features defined in pandas or the model construction classes provided by scikit-learn. The outliers mentioned above (case studies 2, 13, and 14) contain relatively heavy use of conditionals, and as a result, their analysis took slightly longer than the other programs. These results suggest that PRIVGUARD will scale to even longer programs for data analytics and machine learning, especially if those programs follow the same pattern of favoring library use over traditional control-flow constructs.

Table 2 reports performance overhead for all 23 case studies. The results report *analysis performance overhead*—the ratio of the time taken for static analysis to the native execution time of the original program. The results show that this overhead is negligible. For case study programs which take a significant time to run, the performance overhead of deploying PRIVGUARD is typically *less than 1%*. For faster-running programs, the absolute overhead is similar—just a second or two, typically—but this represents a larger relative change when the program's execution time is small. The *maximum* relative performance overhead in our experiments was about 35%, for a program taking only 13.33 seconds.

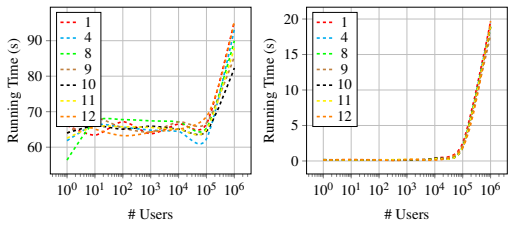
Overall Performance Overhead and Scalability. We also evaluate 7 case studies on our prototype implementation and measure the total overhead of the PRIVGUARD system. The results appear in Figure 7 and 8. For each case study, we synthesize one million random policies by combining possible attributes and changing parameters in the attributes to simulate one million data subjects' privacy preferences. The results show that the performance overhead for ingesting one million policies is under 150 seconds. Concretely, over half of the overhead is spent on Parcel's system overhead such as data uploading, data storage, data encryption, *etc*. Data ingestion takes about one-third of the overhead and the static analysis only takes less than 10 seconds.

We also benchmark the overhead with different numbers of users as shown in Figure 8. Parcel overhead refers to the overhead incurred by the Parcel platform such as data loading or transmission. Scan overhead refers to the time spent on finding the policies no stricter than the guard policy. Merge overhead refers to the time used to merge the datasets inside TEE. Analysis overhead refers to the overhead of running PRIVANALYZER. As shown in Figure 8, Parcel overhead, scan overhead and merge overhead are relatively stable when the number of users is small and then scale linearly with the number of users. Note that we use the log_{10} scale to represent the x-axis. The curves are exponential but the rate scales linearly. For all experiments except the static analysis: Overhead $= O(\#\text{Users}) + O(1)$. Not surprisingly, the analysis overhead is almost constant for a fixed program. The results show that PRIVGUARD is scalable to a large number of users and datasets.

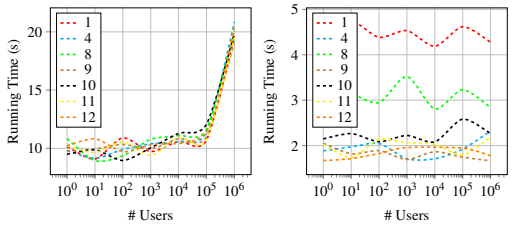
5 Related Work

Related work falls into two categories: (1) formalization and (2) enforcement of privacy policies.

Privacy Policy Formalism. Tschantz et al. [56] use modified Markov Decision Process to formalize purpose restrictions in privacy policies. Chowdhury et al. [31] present a policy language based on a subset of FOTL capturing the requirements of HIPAA. Lam et al. [39] prove that for any privacy policy that conforms to patterns evident in HIPAA, there exists a finite representative hospital database that illustrates how the law applies in all possible hospitals. Gerl et al. [36] introduce LPL, an extensible Layered Privacy Language that allows to express and enforce new privacy properties such as user consent. Trabelsi et al. [55] propose the PPL sticky policy based on XACML to express and process privacy policies in Web 2.0. Azraoui et al. [30] focus on the accountability



(a) Parcel overhead. (b) Scan overhead.



(c) Merge overhead. (d) Analysis overhead.

Figure 8: PRIVGUARD overhead details. The scalability is linear (note the logarithmic scale).

side of privacy policies and extend PPL to A-PPL.

Privacy Policy Compliance Enforcement. Going beyond the formalism of privacy regulations, recent research also explores techniques to enforce formalized privacy policies. Chowdhury et al. [32] propose to use temporal model-checking for run-time monitoring of privacy policies. Sen et al. [53] introduce GROK, a data inventory for Map-Reduce-like big data systems. PODS / SOLID [43] focuses on returning control over data to its owners. In PPL policy engine [55], the policy decision point (PDP) matches the data curator's privacy policy and data subjects' privacy preferences to decide compliance. The privacy policy is enforced by the policy enforcement point. Compared with our work, the PPL policy engine provides limited support for fine-grained privacy compliance in complex data analysis tasks as its enforcement engine relies on direct trigger-to-action translation. In addition, PPL does not provide a rigorous soundness proof. Similar differences exist in its extension A-PPL [30] and the SPECIAL project [2]. Our work provides an enforcement mechanism necessary to address these issues and can be seen as a first step towards meeting the ambitious challenge posed by Maniatis et al. [42].

6 Limitation

We would like to note several limitations of PRIVGUARD and deem mitigating them as important future directions. First, PRIVGUARD is vulnerable to insider attacks. In our threat model, we assume the data analysts are honest but reckless and might violate the privacy regulation unintentionally. Such a threat model should be enough to capture most real-world use cases. However, defending against malicious analysts is much more challenging. Because PRIVANALYZER is implemented as a Python library, it is possible to craft malicious programs that evade its analysis. For example, a malicious program might dynamically redefine PRIVANALYZER's `runFilter` function (used in our earlier example) to always report that the policy has been satisfied. A syntactic analysis that detects the use of dynamic language features before the program is loaded could address this issue. However, it is challenging to detect all such attacks due to the large number of dynamic features in Python.

Second, many *PURPOSE* attributes cannot be automatically enforced by PRIVGUARD as they are not related to program properties. For example, whether a program represents a "legitimate interest" can only be judged by a human, and thus cannot be decided by any fully automated system without manual description by a human. To address this challenge, we choose to log these attributes and make the log accessible for human auditing. We emphasize that our goal is to *minimize* rather than eliminate human efforts in the compliance process.

Third, PRIVGUARD relies on TEE such as AMD SEV to defend against untrusted third parties. However, recent studies have spotted several vulnerabilities in mainstream TEEs [46, 58] which weakens their protection against malicious third parties. Although out of scope, we would like to mention these possible exploits to potential users.

7 Conclusion & Future Work

In this paper, we propose PRIVGUARD, a framework for facilitating privacy regulation compliance. The core component is PRIVANALYZER, a static analyzer supporting compliance verification between a program and a policy. We prototype PRIVGUARD on Parcel, an industrial-level data governance platform. We believe that PRIVGUARD has the potential to significantly reduce the cost of privacy regulation compliance.

There are also several future directions we would like to pursue for future versions of PRIVGUARD. First, we would like to further improve the usability of PRIVGUARD's API in consideration of HCI requirements so that non-experts can easily specify their own privacy preferences. Second, PRIVGUARD now adopts an one-shot consent strategy, which covers most current application scenarios but has several defects as pointed out in [52]. This limitation can be addressed by allowing the data curator to ask data subjects for dynamic consent after data collection, as depicted in [52].

References

[1] Oasis labs parcel sdk. https://www.oasislabs.com/parcelsdk. Accessed: 2021-02-02.

[2] Special: Scalable policy-aware linked data architecture for privacy, transparency and compliance. https://specialprivacy.ercim.eu/. Accessed: 2021-05-04.

[3] Formal concept analysis. https://en.wikipedia.org/wiki/Formal_concept_analysis, 2019. Online; accessed 30 May 2019.

[4] The age of privacy: The cost of continuous compliance. https://datagrail.io/downloads/GDPR-CCPA-cost-report.pdf, 2020. Online; accessed 30 July 2020.

[5] Bank customer churn predition. https://www.kaggle.com/bandiang2/prediction-of-customer-churn-at-a-bank, 2020. Online; accessed 9 January 2020.

[6] Bank customer classification random forest. https://www.kaggle.com/taronzakaryan/bank-customer-classification-random-forest, 2020. Online; accessed 9 January 2020.

[7] Bank customer segmentation. https://www.kaggle.com/paulinan/bank-customer-segmentation, 2020. Online; accessed 9 January 2020.

[8] Classify forest categories. https://www.kaggle.com/suyashgulati/using-pyspark-randomforest-crossvalidatn-gridsrch, 2020. Online; accessed 9 January 2020.

[9] Credit risk analysis. https://www.kaggle.com/damaradiprabowo/clustering-german-credit-data, 2020. Online; accessed 9 January 2020.

[10] Display advertising challenge. https://www.kaggle.com/c/criteo-display-ad-challenge/discussion/10322, 2020. Online; accessed 9 January 2020.

[11] Elo merchant category recommendation. https://www.kaggle.com/c/elo-merchant-category-recommendation, 2020. Online; accessed 9 January 2020.

[12] The gdpr racket: Who's making money from this $9bn business shakedown. https://www.forbes.com/sites/oliversmith/2018/05/02/the-gdpr-racket-whos-making-money-from-this-9bn-business-shakedown/#54c0702034a2, 2020. Online; accessed 30 July 2020.

[13] Google cloud & ncaa ml competition 2019-women's. https://www.kaggle.com/c/womens-machine-learning-competition-2019/discussion/90156, 2020. Online; accessed 9 January 2020.

[14] Heart disease causal inference. https://www.kaggle.com/tentotheminus9/what-causes-heart-disease-explaining-the-model, 2020. Online; accessed 9 January 2020.

[15] Ieee cis fraud detection. https://www.kaggle.com/c/ieee-fraud-detection, 2020. Online; accessed 9 January 2020.

[16] K-anonymity library. https://github.com/KENNN/k-anonymity, 2020. Online; accessed 02 May 2020.

[17] Lanl earthquake prediction. https://www.kaggle.com/c/LANL-Earthquake-Prediction/discussion/94390, 2020. Online; accessed 9 January 2020.

[18] Microsoft malware prediction. https://www.kaggle.com/shrutimechlearn/large-data-loading-trick-with-ms-malware-data, 2020. Online; accessed 9 January 2020.

[19] Nfl punt analytics competition. https://www.kaggle.com/c/NFL-Punt-Analytics-Competition/discussion/78041#latest-663557, 2020. Online; accessed 9 January 2020.

[20] Restaurant revenue prediction. https://www.kaggle.com/jquesadar/restaurant-revenue-1st-place-solution, 2020. Online; accessed 9 January 2020.

[21] Santander customer satisfaction. https://www.kaggle.com/c/santander-customer-satisfaction, 2020. Online; accessed 9 January 2020.

[22] Santander customer transaction prediction. https://www.kaggle.com/c/santander-customer-transaction-prediction, 2020. Online; accessed 9 January 2020.

[23] Simple lstm - pytorch version. https://www.kaggle.com/bminixhofer/simple-lstm-pytorch-version, 2020. Online; accessed 9 January 2020.

[24] Talkingdata adtracking fraud detection challenge. https://www.kaggle.com/c/talkingdata-adtracking-fraud-detection/discussion/56283, 2020. Online; accessed 9 January 2020.

[25] Tensorflow-deep learning to solve titanic. https://www.kaggle.com/linxinzhe/tensorflow-deep-learning-to-solve-titanic, 2020. Online; accessed 9 January 2020.

[26] Tensorflow privacy. https://github.com/tensorflow/privacy, 2020. Online; accessed 03 May 2020.

[27] Web traffic time series forecasting. https://www.kaggle.com/muonneutrino/wikipedia-traffic-data-exploration, 2020. Online; accessed 9 January 2020.

[28] What is iam? https://docs.aws.amazon.com/IAM/latest/UserGuide/introduction.html, 2020. Online; accessed 29 April 2020.

[29] Zillow prize: Zillow's home value prediction (zestimate). https://www.kaggle.com/sudalairajkumar/simple-exploration-notebook-zillow-prize, 2020. Online; accessed 9 January 2020.

[30] Monir Azraoui, Kaoutar Elkhiyaoui, Melek Önen, Karin Bernsmed, Anderson Santana De Oliveira, and Jakub Sendor. A-ppl: an accountability policy language. In *Data privacy management, autonomous spontaneous security, and security assurance*, pages 319–326. Springer, 2014.

[31] Omar Chowdhury, Andreas Gampe, Jianwei Niu, Jeffery von Ronne, Jared Bennatt, Anupam Datta, Limin Jia, and William H Winsborough. Privacy promises that can be kept: a policy analysis method with application to the hipaa privacy rule. In *Proceedings of the 18th ACM symposium on Access control models and technologies*, pages 3–14. ACM, 2013.

[32] Omar Chowdhury, Limin Jia, Deepak Garg, and Anupam Datta. Temporal mode-checking for runtime monitoring of privacy policies. In *International Conference on Computer Aided Verification*, pages 131–149. Springer, 2014.

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

[34] Cynthia Dwork, Aaron Roth, et al. The algorithmic foundations of differential privacy. *Foundations and Trends® in Theoretical Computer Science*, 9(3–4):211–407, 2014.

[35] Hamid Ebadi, David Sands, and Gerardo Schneider. Differential privacy: Now it's getting personal. *Acm Sigplan Notices*, 50(1):69–81, 2015.

[36] Armin Gerl, Nadia Bennani, Harald Kosch, and Lionel Brunie. Lpl, towards a gdpr-compliant privacy language: Formal definition and usage. In *Transactions on Large-Scale Data-and Knowledge-Centered Systems XXXVII*, pages 41–80. Springer, 2018.

[37] Daniel B Giffin, Amit Levy, Deian Stefan, David Terei, David Mazières, John C Mitchell, and Alejandro Russo. Hails: Protecting data privacy in untrusted web applications. In *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, pages 47–60, 2012.

[38] Naoise Holohan, Stefano Braghin, Pól Mac Aonghusa, and Killian Levacher. Diffprivlib: The ibm differential privacy library. *arXiv preprint arXiv:1907.02444*, 2019.

[39] Peifung E Lam, John C Mitchell, Andre Scedrov, Sharada Sundaram, and Frank Wang. Declarative privacy policy: finite models and attribute-based encryption. In *Proceedings of the 2nd ACM SIGHIT International Health Informatics Symposium*, pages 323–332. ACM, 2012.

[40] Ninghui Li, Tiancheng Li, and Suresh Venkatasubramanian. t-closeness: Privacy beyond k-anonymity and l-diversity. In *2007 IEEE 23rd International Conference on Data Engineering*, pages 106–115. IEEE, 2007.

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

[42] Petros Maniatis, Devdatta Akhawe, Kevin R Fall, Elaine Shi, and Dawn Song. Do you know where

your data are? secure data capsules for deployable data protection. In *HotOS*, volume 7, pages 193–205, 2011.

[43] Essam Mansour, Andrei Vlad Sambra, Sandro Hawke, Maged Zereba, Sarven Capadisli, Abdurrahman Ghanem, Ashraf Aboulnaga, and Tim Berners-Lee. A demonstration of the solid platform for social web applications. In *Proceedings of the 25th International Conference Companion on World Wide Web*, pages 223–226. International World Wide Web Conferences Steering Committee, 2016.

[44] Sai Krishna Deepak Maram, Fan Zhang, Lun Wang, Andrew Low, Yupeng Zhang, Ari Juels, and Dawn Song. Churp: Dynamic-committee proactive secret sharing. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 2369–2386, 2019.

[45] Ryan McKenna, Gerome Miklau, Michael Hay, and Ashwin Machanavajjhala. Optimizing error of high-dimensional statistical queries under differential privacy. *Proceedings of the VLDB Endowment*, 11(10):1206–1219, 2018.

[46] Mathias Morbitzer, Sergej Proskurin, Martin Radev, Marko Dorfhuber, and Erick Quintanar Salas. Severity: Code injection attacks against encrypted virtual machines. *arXiv preprint arXiv:2105.13824*, 2021.

[47] Reinhard Munz, Fabienne Eigner, Matteo Maffei, Paul Francis, and Deepak Garg. Unitrax: protecting data privacy with discoverable biases. In *International Conference on Principles of Security and Trust*, pages 278–299. Springer, Cham, 2018.

[48] Andrew C Myers. Jflow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 228–241, 1999.

[49] Flemming Nielson, Hanne R Nielson, and Chris Hankin. *Principles of program analysis*. Springer, 2015.

[50] Andrei Sabelfeld and Alejandro Russo. From dynamic to static and back: Riding the roller coaster of information-flow control research. In *International Andrei Ershov Memorial Conference on Perspectives of System Informatics*, pages 352–365. Springer, 2009.

[51] Stefan Saroiu, Alec Wolman, and Sharad Agarwal. Policy-carrying data: A privacy abstraction for attaching terms of service to mobile data. In *Proceedings of the 16th International Workshop on Mobile Computing Systems and Applications*, pages 129–134, 2015.

[52] Eva Schlehahn, Patrick Murmann, Farzaneh Karegar, and Simone Fischer-Hübner. Opportunities and challenges of dynamic consent in commercial big data analytics. In *IFIP International Summer School on Privacy and Identity Management*, pages 29–44. Springer, 2019.

[53] Shayak Sen, Saikat Guha, Anupam Datta, Sriram K Rajamani, Janice Tsai, and Jeannette M Wing. Bootstrapping privacy compliance in big data systems. In *2014 IEEE Symposium on Security and Privacy*, pages 327–342. IEEE, 2014.

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

[55] Slim Trabelsi, Jakub Sendor, and Stefanie Reinicke. Ppl: Primelife privacy policy engine. In *2011 IEEE International Symposium on Policies for Distributed Systems and Networks*, pages 184–185. IEEE, 2011.

[56] Michael Carl Tschantz, Anupam Datta, and Jeannette M Wing. Formalizing and enforcing purpose restrictions in privacy policies. In *2012 IEEE Symposium on Security and Privacy*, pages 176–190. IEEE, 2012.

[57] Frank Wang, Ronny Ko, and James Mickens. Riverbed: enforcing user-defined privacy constraints in distributed web services. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 615–630, 2019.

[58] Luca Wilke, Jan Wichelmann, Florian Sieck, and Thomas Eisenbarth. undeserved trust: Exploiting permutation-agnostic remote attestation. In *2021 IEEE Security and Privacy Workshops (SPW)*, pages 456–466, 2021.

[59] Lantian Zheng and Andrew C Myers. Dynamic security labels and static information flow control. *International Journal of Information Security*, 6(2-3):67–84, 2007.

A Formal Model of PRIVANALYZER

In this section, we formally present technique for proving soundness of our static analysis. We use the filter attribute as an example to demonstrate the technique for proving soundness in the context of a simple model of a programming language. As described in Section 3, Python is a more dynamic language than our model, and these dynamic features may represent possible side channels for malicious adversaries.

A.1 Abstract Domains

We provide the formal definition on of the abstract domains used in PRIVGUARD in this section. Formally, given a concrete domain $\mathcal{D}$, we define the following components:

- an *abstract domain* $< \mathcal{D}^\sharp, \sqsubseteq >$ containing abstract values ($\mathcal{D}^\sharp$) which represent classes of abstract values, and a partial ordering ($\sqsubseteq$) on those values.
- an *abstraction function* $\alpha : \mathcal{D} \to \mathcal{D}^\sharp$ mapping concrete values to abstract values.
- a *concretization function* $\gamma : \mathcal{D}^\sharp \to \mathcal{P}(\mathcal{D})$ mapping abstract values to sets of concrete values.

For each attribute type, we define $\mathcal{D}^\sharp$, $\sqsubseteq$, α, and γ. We define each abstract domain in terms of a tabular data format approximating a Pandas dataframe (which we denote df).

As described earlier, some kinds of loops may cause the abstract interpreter to loop forever. To address this challenge, we adopt the standard approach of using a *widening operator* [49], denoted ∇, in place of the standard partial ordering operator $\sqsubseteq$. Unlike the partial ordering, the widening operator is guaranteed to stabilize when applied repeatedly in a loop. Finite abstract domains do not require a widening operator; for infinite domains (like the interval domain used in *FILTER* attributes), we adopt the standard widening operator used for the underlying domain (e.g. widening for intervals [49]).

Filter Attributes. We track filtering done by analysis programs using an *interval domain* [49], which is commonly used in the abstract interpretation literature. Abstract dataframes in this domain (denoted $\mathbb{D}^\sharp$) associate an interval (denoted $\mathbb{I}$) with each column c_i, and analysis results are guaranteed to lie within the interval. In addition to known intervals (i.e. (n_1, n_2)), our set of intervals includes $\top$ (i.e. the interval $(-\infty, \infty)$) and $\bot$ (i.e. the interval containing no numbers). Our interval domain works for dataframe columns containing integers or real numbers; our formalization below uses $\mathbb{R}^\infty$ to denote the set of real numbers, extended with infinity.

$$
\begin{array}{llll}
i & \in & \mathbb{I}_\mathbb{R} & = & (\mathbb{R}^\infty \times \mathbb{R}^\infty) \cup \{\top, \bot\} \\
df^\sharp & \in & \mathbb{D}^\sharp & = & (c_1 : \mathbb{I}_\mathbb{R}) \times ... \times (c_n : \mathbb{I}_\mathbb{R})
\end{array}
$$

$$
\begin{array}{llll}
f \in \mathsf{field} & m \in \mathsf{int} & s \in \mathsf{schema} & x \in \mathsf{dataframes} \\
\varphi & \in & \mathsf{filter} & ::= & c < m \mid c > m \\
e & \in & \mathsf{expr} & ::= & x \mid \mathrm{filter}(\varphi, e) \mid \mathrm{project}(s, e) \\
 & & & \mid & \mathrm{redact}(c, n, n, e) \mid \mathrm{join}(e, e) \\
 & & & \mid & \mathrm{union}(e, e) \mid \mathrm{dpCount}(\varepsilon, \delta, e)
\end{array}
$$

Figure 9: Program surface syntax

$$
\begin{array}{lll}
\texttt{eval}(\rho, x) & = & \rho(x) \\
\texttt{eval}(\rho, \mathrm{filter}(\varphi, e)) & = & \sigma_\varphi \texttt{eval}(\rho, e) \\
\texttt{eval}(\rho, \mathrm{project}(s, e)) & = & \Pi_s \texttt{eval}(\rho, e) \\
\texttt{eval}(\rho, \mathrm{redact}(c, n_1, n_2, e)) & = & \{c : \mathsf{stars}(s, n_1, n_2) \mid \\
 & & \quad c : s \in \texttt{eval}(\rho, e)\} \\
\texttt{eval}(\rho, \mathrm{join}(e_1, e_2)) & = & \texttt{eval}(\rho, e_1) \bowtie \texttt{eval}(\rho, e_2) \\
\texttt{eval}(\rho, \mathrm{union}(e_1, e_2)) & = & \texttt{eval}(\rho, e_1) \cup \texttt{eval}(\rho, e_2)
\end{array}
$$

Figure 10: Concrete interpreter for language in Figure 9.

For ease of presentation, and without loss of generality, we define α and γ in terms of dataframes with a single column c. We denote values in the column c by $df.c$.

$$
\begin{array}{lll}
c : \bot & \sqsubseteq & \mathbb{D}^\sharp \\
\mathbb{D}^\sharp & \sqsubseteq & c : \top \\
c : (n_1, n_2) & \sqsubseteq & c : (n_3, n_4) \text{ if } n_1 \geq n_3 \wedge n_2 \leq n_4 \\
\alpha(df) & = & c : (\min(df.c), \max(df.c)) \\
\gamma(c : \top) & = & \mathbb{D} \\
\gamma(c : \bot) & = & \{\} \\
\gamma(c : (n_1, n_2)) & = & \{df \mid \forall v \in df.c.\ n_1 \leq v \leq n_2\}
\end{array}
$$

A.2 Soundness

A sound analysis by abstract interpretation requires defining the following:

- A *programming language* of expressions $e \in \mathsf{Expr}$. We define a simple language for dataframes, inspired by Pandas, in Figure 9.
- A *concrete interpreter* $\texttt{eval} : \mathsf{Env} \times \mathsf{Expr} \to \mathcal{D}$ specifying the semantics of the programming language on concrete values. We define the concrete interpreter for our simple language in Figure 10.
- An *abstract interpreter* $\texttt{eval}^\sharp : \mathsf{Env}^\sharp \times \mathsf{Expr} \to \mathcal{D}^\sharp$ specifying the semantics of the programming language on abstract values. An example for *FILTER* attributes appears in Figure 11.

The concrete interpreter `eval` computes the concrete result of a program e in the context of an environment mapping variables to concrete values. The abstract interpreter $\texttt{eval}^\sharp$ computes an *output policy* of a program e in the context of an abstract environment mapping variables to their policies. The program satisfies its input policies if it has at least one empty clause (*i.e.* a satisfied clause) in its output policy.

$$
\begin{array}{lcl}
\texttt{eval}^\sharp(\rho, x) & = & \rho(x) \\
\texttt{eval}^\sharp(\rho, \text{filter}(\varphi, e)) & = & \texttt{eval}^\sharp(\rho, e) - \text{interval}(\varphi) \\
\texttt{eval}^\sharp(\rho, \text{project}(s, e)) & = & \texttt{eval}^\sharp(\rho, e) \\
\texttt{eval}^\sharp(\rho, \text{redact}(c, n_1, n_2, e)) & = & \texttt{eval}^\sharp(\rho, e) \\
\texttt{eval}^\sharp(\rho, \text{join}(e_1, e_2)) & = & \texttt{eval}^\sharp(\rho, e_1) \sqcup \texttt{eval}^\sharp(\rho, e_2) \\
\texttt{eval}^\sharp(\rho, \text{union}(e_1, e_2)) & = & \texttt{eval}^\sharp(\rho, e_1) \sqcup \texttt{eval}^\sharp(\rho, e_2)
\end{array}
$$

$$
\begin{array}{l}
\text{interval}(c < m) = c : (-\infty, m) \\
\text{interval}(c > m) = c : (m, \infty)
\end{array}
$$

$$
\begin{array}{l}
c : (l_1, u_1) - c : (l_2, u_2) = c : (l_3, u_3) \\
\text{where } l_3 = -\infty \text{ when } l_1 \leq l_2, \text{ and } l_1 \text{ otherwise} \\
\qquad u_3 = \infty \text{ when } u_1 \geq u_2, \text{ and } u_1 \text{ otherwise}
\end{array}
$$

Figure 11: Abstract interpreter for *FILTER* attributes

The *soundness* property for the abstract interpreter says that the concrete result of evaluating a program e is contained in the class of values represented by the result of evaluating the same program using the abstract interpreter.

Theorem 1 (Soundness)**.** *For all environments* ρ *and expressions e, the abstract interpreter* $\texttt{eval}^\sharp$ *is a sound approximation of the concrete interpreter* $\texttt{eval}$*:*

$$\texttt{eval}(\rho, e) \in \gamma(\texttt{eval}^\sharp(\alpha(\rho), e))$$

where the abstract environment $\alpha(\rho)$ *is obtained by abstracting each value in the concrete environment* ρ *(i.e.* $\alpha(\rho)(x) = \alpha(\rho(x))$*).*

Soundness can be proven for each abstract domain separately. In each case, the proof of soundness proceeds by induction on e, with case analysis on the kind of abstract value returned by uses of $\texttt{eval}^\sharp$ on subterms.

Soundness for Filter Attributes. We present the abstract interpreter for the filter abstract domain in Figure 11. The interesting case of this interpreter is the one for filter expressions, which converts the filter predicate φ to an interval and returns an abstract value derived from the meet of this interval and the recursive call. We prove the soundness of the interpreter as following.

Proof of soundness. By induction on e. We consider the (representative) case where $e = \text{filter}(\varphi, e')$. By the inductive hypothesis, we have that $\texttt{eval}(\rho, e') \in \gamma(\texttt{eval}^\sharp(\alpha(\rho), e'))$. Let $\text{interval}(\varphi) = (n_1, n_2)$. We want to show (by definition of $\texttt{eval}$ and $\texttt{eval}^\sharp$:

$$
\begin{array}{l}
\texttt{eval}(\rho, \text{filter}(\varphi, e')) \in \gamma(\texttt{eval}^\sharp(\alpha(\rho), \text{filter}(\varphi, e'))) \\
\iff \sigma_\varphi \texttt{eval}(\rho, e') \in \gamma(\texttt{eval}^\sharp(\alpha(\rho), e') - \text{interval}(\varphi)) \\
\iff \sigma_{n_1 \leq c \leq n_2} \texttt{eval}(\rho, e') \in \gamma(c : (n_1, n_2) \sqcap \texttt{eval}^\sharp(\alpha(\rho), e')) \\
\iff \sigma_{n_1 \leq c \leq n_2} \texttt{eval}(\rho, e') \in \gamma(c : (\max(n_1, n_3), \min(n_2, n_4))) \\
\iff \sigma_{n_1 \leq c \leq n_2} \texttt{eval}(\rho, e') \in \\
\quad \{df \mid \forall v \in df.c.\ \max(n_1, n_3) \leq v \leq \min(n_2, n_4)\}
\end{array}
$$

which holds by the inductive hypothesis and semantics of selection in relational algebra. □

B Usability Survey

To complement the survey in [53] targeting privacy champions, we conducted an online survey targeting general users to obtain a preliminary understanding of how far expertise is needed to understand or encode privacy preferences. The survey is granted IRB exemption by Office for Protection of Human Subjects under category 2 of the Federal and/or UC Berkeley requirements.

We recruit 30 participants in total among which 7 has no background in programming (Group A), 2 has programmed in one language (Group B), 15 has programmed in two languages or more (Group C), and 6 self-identify as experts in programming language (Group D). The survey is comprised of 8 questions in total. The first three are about understanding privacy policies and the latter five are to choose the correct option from 4 possible choices. Group A makes 38.1% (8/21) mistakes when understanding and 31.4% (11/35) mistakes when selecting. Group B makes 16.7% (1/6) mistakes when understanding and 20.0% (2/10) mistakes when selecting. Group C makes 15.6% (7/45) mistakes when understanding and 17.3% (13/75) mistakes when selecting. Group D makes 11.1% (2/18) mistakes when understanding and 13.3% (4/30) mistakes when selecting. Besides, each question has a different focus. For example, Question 2 focuses on understanding *ROLE* and *PURPOSE* attributes.

We observe several interesting facts in the survey results. First, there is a big gap in accuracy between Group A and B. This indicates that it might not be trivial for users without programming experience to specify their privacy preferences in LEGALEASE directly. Although out of scope of this paper, we deem it an important future direction to simplify this process for Group A users using more user-friendly API or machine-learning-based translation tools. Besides, this also shows that any programming experience is helpful in understanding and encoding in LEGALEASE. Second, there is no obvious accuracy gap between Group B and Group C, and Group D has better accuracy than them. Third, it is hard for all groups to answer questions related to *PRIVACY* attributes. The hardness stems from the hardness in understanding privacy techniques such as differential privacy.

Ethical Considerations. The survey was posted as a public questionnaire on Twitter and Wechat with informed consent. The participants opted in the survey voluntarily. In order to fully respect the participants' privacy, we do not collect any personal identifiable information from them. Only the answers to the questionnaire are collected.