Static Analysis by Abstract Interpretation Against Data Leakage in Machine Learning

Caterina Urban[b], Pavle Subotić[a], Filip Drobnjaković[a]

[a] *Sonic Research | Formal Labs, Serbia*
[b] *Inria & ENS | PSL, France*

Abstract

Data leakage is a well-known problem in machine learning which occurs when the training and testing datasets are not independent. This phenomenon leads to unreliably overly optimistic accuracy estimates at training time, followed by a significant drop in performance when models are deployed in the real world. This can be dangerous, notably when models are used for risk prediction in high-stakes applications. In this paper, we propose an abstract interpretation-based static analysis to prove the absence of data leakage at development time, long before model deployment and even before model training. We implemented it in the NBLyzer framework and we demonstrate its performance and precision on 2111 Jupyter notebooks from the Kaggle competition platform.

Keywords: Abstract Interpretation, Static Analysis, Data Leakage, Data Science

1. Introduction

As artificial intelligence (AI) continues its unprecedented impact on society, ensuring that machine learning (ML) models are reliable and accurate is more critical than ever. Proper training, validation, and testing of these model is crucial to ensure they generalize well to real-world scenarios. This iterative task is typically performed within data science notebook environments, which have become the go-to tools for ML practitioners due to their interactive nature and ability to facilitate rapid experimentation and model development (Perkel, 2018; jet, 2020). However, amid the flexibility that these environments provide, the subtle yet insidious issue of *data leakage* (Papadimitriou and Garcia-Molina, 2009) can easily be introduced and silently compromise the model training process. Data leakage has been identified as a pervasive problem by the data science community (Kapoor and Narayanan, 2023; Kaufman et al., 2012; Nis, 2018). In a number of recent cases, data leakage crippled the performance of real-world risk prediction systems leading to flawed decision-making with dangerous consequences in high-stakes applications such as child welfare (Chouldechova et al., 2018) and healthcare (Wong et al., 2021). As such, preventing and mitigating data leakage has become a priority for the ML community, particularly when safety and fairness are paramount.

Data leakage occurs when information from outside the training dataset is inadvertently used to train a model. It can significantly impact the performance and generalization capabilities of the model, leading to unreliable overly optimistic performance estimates during model evaluation. The primary forms of data leakage are due to *train-test contamination* – when information from the test or validation dataset leaks into the training set – or *target leakage* – when features directly related to the predicted variable are included in the training data (Wong et al., 2021). Data leakage can occur for various reasons, typically it is due to improper data pre-processing or feature engineering.

Example 1 (Feature Normalization Leakage)**.** *Consider the excerpt of a Python data science notebook (inspired by* 569.ipynb *from our benchmarks) shown in Figure 1. It comprises six cells, numbered from* 1 *to* 6 *in the order in which they are defined and, we assume, executed. Cell* 1 *simply imports the necessary libraries from Pandas and Sklearn. Then, Cell* 2 *reads data from a CSV file and Cell* 3 *rescales it to be in the* $[0, 1]$ *range. Cell* 4 *splits the data into train and test sets. Finally, Cell* 5 *and Cell* 6 *respectively trains and test a logistic regression model.*

In this case, data leakage is introduced because Cell 3 *performs a normalization of the data,* before Cell 4 *splits it into train and test sets. In this way, the normalization had knowledge of the full distribution of data when calculating*

In [1]:
```
import pandas as pd
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
```

In [2]:
```
data = pd.read_csv("credit.csv")
X = data[["Age", "Amount"]]
y = data[["Approved"]]
```

In [3]:
```
min_max_scaler = MinMaxScaler()
X = min_max_scaler.fit_transform(X)
```

In [4]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.025)
```

In [5]:
```
lr = LogisticRegression()
lr.fit(X_train, y_train)
```

In [6]:
```
y_pred = lr.predict(X_test)
accuracy_score(y_test, y_pred)
```

Figure 1: Example of data science notebook exhibiting data leakage caused by improper data pre-processing.

the scaling factors and, as a result, the train and test data become indirectly dependent on each other. In our experimental evaluation (cf. Section 6) we found that this is a common pattern for introducing a data leakage in several real-world notebooks. In the following, we say that the data resulting from the normalization done in Cell 3 *is* tainted*.*

Example 2 (Feature Engineering Leakage)**.** *Consider the Python data science notebook shown in Figure 2. Again, Cell* 1 *simply imports Pandas and Sklearn libraries and Cell* 2 *reads data from a CSV file. Cell* 3 *creates a new data column indicating, for each row, whether the corresponding value in the "Amount" column is above the mean of the values in the "Amount" column. Finally, Cell* 4 *splits train and test data, and Cell* 5 *and Cell* 6 *train and test a model.*

In this case, data leakage is introduced by the feature engineering done in Cell 3 *because it uses the mean value of "Amount" from the entire dataset. As a result, the model trained in Cell* 5 *has access to information (the mean value) that it should not have during deployment, leading to biased predictions when faced with new, previously unseen data.*

Mainstream approaches against data leakage primarily focus on retroactive detection (Kaufman et al., 2011; Papadimitriou and Garcia-Molina, 2009). Upon encountering a suspicious results, such as an unexpectedly high model accuracy, these approaches typically leverage data analysis techniques to uncover hidden data dependencies. However, a significant limitation of retroactive detection is its reliance on noticeable anomalies. Reasonable yet flawed results may evade scrutiny, allowing data leakage to persist until the model is already deployed. This is a natural use case for *static analysis* to detect data leakages already at development time, even before the model is trained or tested.

In this paper, we propose a static analysis for proving the absence of data leakage in data-manipulating programs: it tracks the origin of data used for training and testing and verifies that they originate from *disjoint* and *untainted* data sources. In Example 1 our analysis identifies a data leakage since `X_train` and `X_test`, despite being disjoint, originate from previously normalized (and, thus, tainted) data. Similar, in Example 2, data leakage is detected since values in `X_train` and `X_test` are previously calculated over the same data source (they are again disjoint but tainted). With this work, we mainly address data leakage due train-test contamination, including contamination caused by multi-step feature engineering. However, other forms of data leakage such as target leakage (or also *group leakage* – when data associated with the same group appears in both training and testing sets (Chouldechova et al., 2018)) can also be addressed with the synergic use of a data correlation analysis. We consider such an extension out of the scope of this work, and leave it for future development.

Our static analysis (cf. Section 4) is designed within the abstract interpretation framework (Cousot and Cousot, 1977): it is derived through successive abstractions from the (sound and complete, but not computable) collecting program semantics (cf. Section 3). This formal development allows us to formally justify the soundness of the analysis (cf. Theorem 3), and to exactly pinpoint where it can lose precision (e.g., modeling data joins, cf. Section 4.2) to guide the design of more precise abstractions, if necessary in the future (in our evaluation we found the current analysis to

```
In [1]:  import pandas as pd
         from sklearn.model_selection import train_test_split
         from sklearn.linear_model import LogisticRegression

In [2]:  df = pd.read_csv("credit.csv")

In [3]:  m = df.mean()
         a = df['Amount']
         df['High'] = a > m['Amount']

In [4]:  X = df[['Age', 'High']]
         y = df[['Approved']]
         X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.025)

In [5]:  lr = LogisticRegression()
         lr.fit(X_train, y_train)

In [6]:  y_pred = lr.predict(X_test)
```

Figure 2: Exemple of data science notebook exhibiting data leakage caused by improper feature engineering.

be sufficiently precise, cf. Section 6). Moreover, it allows a clear comparisons with other related static analyses, e.g., information flow and taint analyses (cf. Section 7). Finally, this design principle allowed us to identify and overcome issues and shortcomings of previous data leakage analysis attempts (Subotić et al., 2022; Subotic et al., 2022).

We implemented our analysis in the NBLyzer (Subotić et al., 2022) static analysis framework for data science notebooks. We evaluate its performance on 2111 Jupyter notebooks from the Kaggle competition platform, and demonstrate that our approach scales to the performance constraints of interactive data science notebook environments while detecting 25 real data leakages with a precision of 93%. Notably, we are able to detect 60% more data leakages compared to the ad-hoc analysis previously implemented in NBLyzer.

Note. The results described in this paper have been published in (Drobnjakovic et al., 2024) and are presented here with several extensions. More specifically, with respect to (Drobnjakovic et al., 2024), we better characterize the forms of data leakage that we address with our static analysis (and discuss future extensions to tackle other forms). Moreover, Section 3 and Section 4 feature extended explanations and additional examples to better illustrate our method, as well as complete proofs. Finally, implementation details that were omitted in (Drobnjakovic et al., 2024) are presented in Section 5, and Section 6 has been extended to fully present our experimental evaluation.

2. Background

2.1. Data Frame-Manipulating Programs

We consider programs manipulating data frames, that is, tabular data structures with columns labeled by non-empty unique names. Let $\mathbb{V}$ be a set of (heterogeneous atomic) values (i.e., such as numerical or string values). We can formalize a data frame as a possibly empty $(r \times c)$-matrix of values, where $r \in \mathbb{N}$ and $c \in \mathbb{N}$ denote the number of matrix rows and columns, respectively. Let

$$\mathbb{D} \stackrel{\text{def}}{=} \bigcup_{r \in \mathbb{N}} \bigcup_{c \in \mathbb{N}} \mathbb{V}^{r \times c} \tag{1}$$

be the set of all possible data frame. Given a non-empty data frame $D \in \mathbb{D}$, we use R_D and C_D to denote the number of its rows and columns, respectively, and write $\mathsf{hdr}(D)$ for the set of labels of its columns. We also write $D[r]$ for the specific row indexed with $r \in R_D$ in D.

2.2. Trace Semantics

The *semantics* of a data frame-manipulating program is a mathematical characterization of its behavior when executed for all possible input data. We model the operational semantics of a program in a language-independent way

as a *transition system* $\langle \Sigma, \tau \rangle$, where Σ is a (potentially infinite) set of program states and the transition relation $\tau \subseteq \Sigma \times \Sigma$ describes the possible transitions between states. The set of program *final states* is $\Omega \stackrel{\text{def}}{=} \{s \in \Sigma \mid \forall s' \in \Sigma : \langle s, s' \rangle \notin \tau\}$.

In the following, let $\Sigma^{+\infty} \stackrel{\text{def}}{=} \Sigma^+ \cup \Sigma^\omega$ be the set of all non-empty finite or infinite sequences of program states. A *trace* is a non-empty sequence of program states that respects the transition relation τ, that is, $\langle s, s' \rangle \in \tau$ for each pair of consecutive states $s, s' \in \Sigma$ in the sequence. The *trace semantics* $\Upsilon \in \mathcal{P}(\Sigma^{+\infty})$ generated by a transition system $\langle \Sigma, \tau \rangle$ is the union of all finite traces that are terminating with a final state in Ω, and all infinite traces (Cousot, 2002):

$$\begin{aligned} \Upsilon \stackrel{\text{def}}{=} & \bigcup_{n \in \mathbb{N}^+} \{s_0 \dots s_{n-1} \in \Sigma^n \mid \forall i < n-1 : \langle s_i, s_{i+1} \rangle \in \tau, s_{n-1} \in \Omega\} \\ & \cup \ \{s_0 \cdots \in \Sigma^\omega \mid \forall i \in \mathbb{N} : \langle s_i, s_{i+1} \rangle \in \tau\} \end{aligned} \tag{2}$$

In the rest of the paper, we write $\Upsilon[\![P]\!]$ for the trace semantics of a program P.

3. Concrete Data Leakage Semantics

The trace semantics fully describes the behavior of a program. However, reasoning about a particular property of a program is facilitated by the design of a semantics that abstracts away from irrelevant details about program executions. In this section, we define our property of interest — absence of data leakage — and use abstract interpretation to systematically derive, by abstraction of the trace semantics, a semantics that precisely captures this property.

3.1. (Absence of) Data Leakage

We use an extensional definition of a *property* as the set of elements having such a property (Cousot and Cousot, 1977, 1979), e.g., we represent the property "being an even natural number" as the set of numbers $\{0, 2, 4, 6, 8, \dots\}$. This allows checking property satisfaction by set inclusion (see below) also across abstractions (cf. Theorems 1 and 2) in the next subsections. Semantic properties of programs are properties of their semantics. Thus, properties of programs with trace semantics in $\mathcal{P}(\Sigma^{+\infty})$ are sets of sets of traces in $\mathcal{P}(\mathcal{P}(\Sigma^{+\infty}))$. The set of program properties forms a complete boolean lattice $\langle \mathcal{P}(\mathcal{P}(\Sigma^{+\infty})), \subseteq, \cup, \cap, \emptyset, \mathcal{P}(\Sigma^{+\infty}) \rangle$ for subset inclusion (i.e., logical implication). The strongest property is the standard *collecting semantics* $\Lambda \in \mathcal{P}(\mathcal{P}(\Sigma^{+\infty}))$:

$$\Lambda \stackrel{\text{def}}{=} \{\Upsilon\} \tag{3}$$

i.e., the property of "being the program with trace semantics Υ". Let $\Lambda(\!|P|\!)$ denote the collecting semantics of a program P. Then, P satisfies a property $\mathcal{H} \in \mathcal{P}(\mathcal{P}(\Sigma^{+\infty}))$ if and only if its collecting semantics is a subset of $\mathcal{H}$:

$$P \models \mathcal{H} \Leftrightarrow \Lambda(\!|P|\!) \subseteq \mathcal{H}. \tag{4}$$

In this paper, we consider the property of *absence of data leakage*, which requires data used for training and data used for testing a machine learning model to be *independent*.

Example 3 ((In)dependent Data Frame Variables)**.** *Let us consider programs with a single input data frame variable reading data frames in* $\bigcup_{r \in \{1,2,3,4\}} \{3, 9\}^r$ *(cf. Equation 1), with four rows and one single column with values in* $\{3, 9\}$.

One such program P (akin to the data science notebook in Example 1) first performs min-max normalization (i.e., rescaling all data frame values to be in the $[0, 1]$ *range) and then splits the data frame in half to use the first two rows for training and the last two rows for testing. The table in Figure 3a shows all possible train and test data resulting from all possible values of the input data of this program P. In this case, train and test data are* not *independent: if we consider, for instance, the execution* σ *with input data frame value "3|9|9|9" we can change the value of its first row (i.e.,* $r = 1$ *in Figure 4) from* $\bar{v} = 3$ *to* $\bar{v} = 9$ *(while leaving all other rows unchanged) to obtain an execution* σ' *resulting in a difference in* both *train and test data (i.e.,* $\sigma(\mathrm{U}_P^{\text{train}}) \neq \sigma'(\mathrm{U}_P^{\text{train}})$ *and* $\sigma(\mathrm{U}_P^{\text{test}}) \neq \sigma'(\mathrm{U}_P^{\text{test}})$ *in Figure 4), with train data differing at line* 2 *and test data differing at both lines* 1 *and* 2*).*

Instead, the table in Figure 3b shows all possible train and test data resulting from all possible input data of another program Q in which the min-max normalization is performed after *the split into train and test data. Here train and test data remain independent as modifying any input data row r in any execution yields another execution that may result in a difference in either train and test data* but never both*. Equivalently, all possible values of* either *train and test data are possible independently of the choice of the value of the row r.*

input data	3	3	3	3	9	9	9	9	3	3	3	3	9	9	9	9	1
	3	3	9	9	3	3	9	9	3	3	9	9	3	3	9	9	2
	3	9	3	9	3	9	3	9	3	9	3	9	3	9	3	9	3
	3	3	3	3	3	3	3	3	9	9	9	9	9	9	9	9	4
	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	
train data	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	0	1
	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	0	2
test data	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	0	1
	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	0	2
												σ				σ'	

(a)

input data	3	3	3	3	9	9	9	9	3	3	3	3	9	9	9	9	1
	3	3	9	9	3	3	9	9	3	3	9	9	3	3	9	9	2
	3	9	3	9	3	9	3	9	3	9	3	9	3	9	3	9	3
	3	3	3	3	3	3	3	3	9	9	9	9	9	9	9	9	4
	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	
train data	0	0	0	0	1	1	0	0	0	0	0	0	1	1	0	0	1
	0	0	1	1	0	0	0	0	0	0	1	1	0	0	0	0	2
test data	0	1	0	1	0	1	0	1	0	0	0	0	0	0	0	0	1
	0	0	0	0	0	0	0	0	1	0	1	0	1	0	1	0	2

(b)

Figure 3: Train and test data resulting from all possible input data of program P (*a*) and program Q (*b*) in Example 3.

$$\textsc{independent}(\Upsilon[\![P]\!]) \stackrel{\text{def}}{=} \forall \sigma \in \Upsilon[\![P]\!], i \in \mathrm{I}_P, r \in \{0, \ldots, R_{i-1}\} \colon \textsc{unchanged}(\sigma, i, r, \mathrm{U}_P^{\text{test}}) \vee \textsc{unchanged}(\sigma, i, r, \mathrm{U}_P^{\text{train}})$$

$$\textsc{unchanged}(\sigma, i, r, U) \stackrel{\text{def}}{=} \forall \bar{v} \in \mathbb{V}^{C_i} \colon \sigma(i)[r] \neq \bar{v} \Rightarrow (\exists \sigma' \in \Upsilon[\![P]\!] \colon \sigma'(i)[r] = \bar{v} \wedge \eta(\sigma) = \eta(\sigma') \wedge \sigma(U) = \sigma'(U))$$

$$\eta(\sigma) \stackrel{\text{def}}{=} \lambda j \colon \lambda r' \colon \begin{cases} \sigma(j)[r'] & j \in \mathrm{I}_P \setminus \{i\} \vee r' \in \{0, \ldots, R_{i-1}\} \colon r' \neq r \\ \top & \text{otherwise} \end{cases}$$

$$\sigma(X) = \sigma'(X) \stackrel{\text{def}}{=} \forall x \in X \colon \sigma(x) = \sigma'(x)$$

Figure 4: Formal definition of the INDEPENDENT predicate, which states when a program (semantics) uses independent data for training and testing.

More formally, let $\mathbb{X}$ be the set of all the (data frame) variables of a (data frame-manipulating) program P. We denote with $\mathrm{I}_P \subseteq \mathbb{X}$ the set of its *input* or source data frame variables, i.e., data frame variables whose value is directly read from the input, and use $\mathrm{U}_P \subseteq \mathbb{X}$ to denote the set of its *used* data frame variables, i.e., data frame variables used for training or testing a ML model. We write $\mathrm{U}_P^{\text{train}} \subseteq \mathrm{U}_P$ and $\mathrm{U}_P^{\text{test}} \subseteq \mathrm{U}_P$ for the variables used for training and testing, respectively. For simplicity, we can assume that programs are in static single-assignment form so that data frame variables are assigned exactly once: data is read from the input, transformed and normalized, and ultimately used for training and testing. Given a trace $\sigma \in \Upsilon[\![P]\!]$, we write $\sigma(i)$ and $\sigma(o)$ to denote the value of the data frame variables $i \in \mathrm{I}_P$ and $o \in \mathrm{U}_P$ in σ. We define when used data frame variables are independent in a program with trace semantics $\Upsilon[\![P]\!]$ in Figure 4, where R_i and C_i stand for $R_{\sigma(i)}$ (i.e., number of rows of the data frame value of $i \in \mathrm{I}_P$) and $C_{\sigma(i)}$ (i.e., number of columns of the data frame value of $i \in \mathrm{I}_P$), respectively. The definition requires that changing the value of a data source $i \in \mathrm{I}_P$ can modify data frame variables used for training ($\mathrm{U}_P^{\text{train}}$) or testing ($\mathrm{U}_P^{\text{test}}$), *but not both*: the value of data frame variables used for either training or testing in a trace σ remains the same independently of all possible values $\bar{v} \in \mathbb{V}^{C_i}$ of any portion (e.g., any row $r \in \{0, \ldots, R_{i-1}\}$) of any input data frame variable $i \in \mathrm{I}_P$ in σ. Note that this definition quantifies over changes in data frame rows since the split into train and test data happens across rows (e.g., using `train_test_split` in Pandas), but takes into account all possible column values in each row ($\bar{v} \in \mathbb{V}^{C_i}$). It also implicitly takes into account implicit flows of information by considering traces in $\Upsilon[\![P]\!]$. In particular, in terms of secure information flow, notably non-interference, this definition says that *we cannot simultaneously observe different values* in $\mathrm{U}_P^{\text{train}}$ and $\mathrm{U}_P^{\text{test}}$, regardless of the values of the input data frame variables. Here we weaken non-interference to consider either $\mathrm{U}_P^{\text{train}}$ or $\mathrm{U}_P^{\text{test}}$ as low outputs (depending on which row of the input data frame variables is modified),

$$\alpha_{I\rightsquigarrow U}(S) \stackrel{\text{def}}{=} \left\{ i[r] \rightsquigarrow o[r'] \;\middle|\; \begin{array}{l} i \in I, r \in \mathbb{N}, o \in U, r' \in \mathbb{N}, (\forall T \in S : \exists \sigma \in T, \bar{v} \in \mathbb{V}^{C_i} : \sigma(i)[r] \neq \bar{v} \wedge \\ \quad \forall \sigma' \in T : \sigma'(i)[r] = \bar{v} \wedge \eta(\sigma) = \eta(\sigma') \Rightarrow \sigma(o)[r'] \neq \sigma'(o)[r']) \end{array} \right\}$$

Figure 5: Formal definition of the dependency abstraction $\alpha_{I\rightsquigarrow U} : \mathcal{P}(\mathcal{P}(\Sigma^{+\infty})) \to \mathcal{P}((\mathbb{X} \times \mathbb{N}) \times (\mathbb{X} \times \mathbb{N}))$.

instead of fixing the choice beforehand. Note also that UNCHANGED quantifies over all possible values $\bar{v} \in \mathbb{V}^{C_i}$ of the changed row i rather than quantifying over traces to allow non-determinism, i.e., not all traces that only differ at row $r \in \{0, \ldots, R_{i-1}\}$ of data frame variable $i \in \mathrm{I}_P$ need to agree on the values of the used variables $U \subseteq U_P$ but all values of U that are feasible from a value of r of i need to be feasible for *all* possible values of r of i. Only if we exclude non-determinism, the definition of UNCHANGED can be simplified to a perhaps more familiar non-interference formulation: $\text{UNCHANGED} \stackrel{\text{def}}{=} \forall \sigma, \sigma' \in \Upsilon[\![P]\!] : \sigma(i)[r] \neq \sigma'(i)[r] \wedge \eta(\sigma) = \eta(\sigma') \Rightarrow \sigma(U) = \sigma'(U)$, where the values of the used variables U must be the same in any two traces differing only in the value at row $r \in \{0, \ldots, R_{i-1}\}$ of the input data frame variable $i \in \mathrm{I}_P$. In terms of input data (non-)usage (Urban and Müller, 2018), our definition says that training and testing *do not use* the same (portions of the) input data sources. Here we generalize the notion of data usage proposed by Urban and Müller (2018) to multi-dimensional variables and allow multiple values for all outcomes but one (variables used for either training or testing) for each variation in the values of the input variables.

The absence of data leakage property $\mathcal{I}$ can now be formally defined as the set

$$\mathcal{I} \stackrel{\text{def}}{=} \{\Upsilon[\![P]\!] \in \mathcal{P}(\Sigma^{+\infty}) \mid \text{INDEPENDENT}(\Upsilon[\![P]\!])\} \tag{5}$$

of program semantics that use independent data for training and testing ML models. Thus, from Equation 4, we have

$$P \models \mathcal{I} \Leftrightarrow \Lambda(\!|P|\!) \subseteq \mathcal{I}. \tag{6}$$

Example 4. *The program Q (resp. P) from Example 3 satisfies (resp. does not satisfy) the absence of data leakage property $\mathcal{I}$, i.e., $Q \models \mathcal{I}$ (resp. $P \not\models \mathcal{I}$).*

In the rest of this section, we derive, by abstraction of the collecting semantics Λ, a *sound* and *complete* semantics $\dot{\Lambda}$ that contains only and exactly the information needed to reason about (the absence of) data leakage. In the next section, a further abstraction loses completeness but yields a *sound* and *computable* over-approximation of $\dot{\Lambda}$ that allows designing a static analysis to effectively detect data leakage in data frame-manipulating programs.

3.2. Dependency Semantics

From the definition of absence of data leakage, we observe that for reasoning about data leakage we essentially need to track the flow of information between (portions of) input data sources and data used for training or testing. Thus, we can abstract the collecting semantics into a set of dependencies between (rows of) input data frame variables and used data frame variables.

We define the following Galois connection:

$$\langle \mathcal{P}(\mathcal{P}(\Sigma^{+\infty})), \subseteq \rangle \underset{\alpha_{I\rightsquigarrow U}}{\overset{\gamma_{I\rightsquigarrow U}}{\leftrightarrows}} \langle \mathcal{P}((\mathbb{X} \times \mathbb{N}) \times (\mathbb{X} \times \mathbb{N})), \supseteq \rangle \tag{7}$$

between sets of sets of traces and sets of relations (i.e., dependencies) between data frame variables indexed at some row. The abstraction and concretization function are parameterized by a set $I \subseteq \mathbb{X}$ of input variables and a set $U \subseteq \mathbb{X}$ of used variables of interest. In particular, the dependency abstraction $\alpha_{I\rightsquigarrow U} : \mathcal{P}(\mathcal{P}(\Sigma^{+\infty})) \to \mathcal{P}((\mathbb{X} \times \mathbb{N}) \times (\mathbb{X} \times \mathbb{N}))$ is defined in Figure 5, where we write $i[r] \rightsquigarrow o[r']$ for a dependency $\langle\langle i, r\rangle, \langle o, r'\rangle\rangle$ between a data frame variable $i \in I$ at the row indexed by $r \in \mathbb{N}$ and a data frame variable $o \in U$ at the row indexed by $r' \in \mathbb{N}$. In particular, $\alpha_{\rightsquigarrow}$ extracts a dependency $i[r] \rightsquigarrow o[r']$ when (in all sets of traces T in the semantic property S) there is a value $\bar{v} \in \mathbb{V}^{C_i}$ for row r of data frame variable i that changes the value at row r' of data frame variable o, that is, there is a value for row r' of data frame variable o that cannot be reached if the value for row r of i is not changed to $\bar{v}$ (and all else remains the same, i.e., $\eta(\sigma) = \eta(\sigma')$). This is essentially the negation of the UNCHANGED predicate in Figure 4.

Note that our dependency abstraction generalizes that of Cousot (Cousot, 2019) to non-deterministic programs and multi-dimensional data frame variables, thus tracking dependencies between portions of data frames. As in

(Cousot, 2019), this is an abstraction of semantic properties thus the dependencies must hold for all semantics having the semantic property: more semantics have a semantic property, fewer dependencies will hold for all semantics. Therefore, sets of dependencies are ordered by superset inclusion $\supseteq$ (cf. Equation 7).

Example 5 (Dependencies Between Data Frame Variables)**.** *Let us consider again the program P from Example 3. Let i denote the input data frame of the program and let* o_{train} *and* o_{test} *denote the data frames used for training and testing. In this case, for instance, we have* $i[1] \rightsquigarrow o_{\text{train}}[2]$ *because, taking execution* σ*, changing only the value of* $i[1]$ *from* 3 *to* 9 *yields execution* σ' *which changes the value of* $o_{\text{train}}[2]$*, i.e., all other executions either differ at other rows of i or differ at least in the value of* $o_{\text{train}}[2]$ *(such as* σ'*). In fact, the set of dependencies for the whole set of executions of the program shows that* o_{train} *and* o_{test} *depend on* all *rows of the input data frame variable i.*

Instead, performing normalization after *splitting into train and test data as in program Q (also from Example 3) yields the set of dependencies* $\{i[1] \rightsquigarrow o_{\text{train}}[j], i[2] \rightsquigarrow o_{\text{train}}[j], i[3] \rightsquigarrow o_{\text{test}}[j], i[4] \rightsquigarrow o_{\text{test}}[j]\}$*,* $j \in \{1, 2\}$*, where* o_{train} *and* o_{test} *depend on disjoint subsets of rows of the input data frame i.*

It is easy to see that the abstraction function $\alpha_{\mathrm{I}_P \rightsquigarrow \mathrm{U}_P}$ is a complete join morphism. Thus, we can define the concretization function $\gamma_{I \rightsquigarrow U} : \mathcal{P}((\mathbb{X} \times \mathbb{N}) \times (\mathbb{X} \times \mathbb{N})) \to \mathcal{P}(\mathcal{P}(\Sigma^{+\infty}))$ as:

$$\gamma_{\mathrm{I}_P \rightsquigarrow \mathrm{U}_P}(D) \stackrel{\text{def}}{=} \bigcup \{S \mid \alpha_{\mathrm{I}_P \rightsquigarrow \mathrm{U}_P}(S) \supseteq D\}. \tag{8}$$

We can now define the *dependency semantics* $\Lambda_{I \rightsquigarrow U} \in \mathcal{P}((\mathbb{X} \times \mathbb{N}) \times (\mathbb{X} \times \mathbb{N}))$ by abstraction of the collecting semantics Λ: $\Lambda_{I \rightsquigarrow U} \stackrel{\text{def}}{=} \alpha_{I \rightsquigarrow U}(\Lambda)$. In the rest of the paper, we write $\Lambda(\!|P|\!)_{\rightsquigarrow}$ to denote the dependency semantics of a program P, leaving the sets of data frame variables of interest I and U implicitly set to I_P and U_P, respectively. The dependency semantics remains sound and complete:

Theorem 1. $P \models \mathcal{I} \Leftrightarrow \Lambda(\!|P|\!)_{\rightsquigarrow} \supseteq \alpha_{\mathrm{I}_P \rightsquigarrow \mathrm{U}_P}(\mathcal{I})$

Proof. Let $P \models \mathcal{I}$. From the subset inclusion in Section 3.1, we have that $\Lambda(\!|P|\!) \subseteq \mathcal{I}$. Thus, from the Galois connection in Equation 7 (note the inverse $\supseteq$ order in the abstract domain!), we have $\alpha_{\mathrm{I}_P \rightsquigarrow \mathrm{U}_P}(\Lambda(\!|P|\!)) \supseteq \alpha_{\mathrm{I}_P \rightsquigarrow \mathrm{U}_P}(\mathcal{I})$. From the definition of $\Lambda(\!|P|\!)_{\rightsquigarrow}$, we can then conclude that $\Lambda(\!|P|\!)_{\rightsquigarrow} \supseteq \alpha_{\mathrm{I}_P \rightsquigarrow \mathrm{U}_P}(\mathcal{I})$.

Vice versa, let $\Lambda(\!|P|\!)_{\rightsquigarrow} \supseteq \alpha_{\mathrm{I}_P \rightsquigarrow \mathrm{U}_P}(\mathcal{I})$. From the definition of $\Lambda(\!|P|\!)_{\rightsquigarrow}$, we have $\alpha_{\mathrm{I}_P \rightsquigarrow \mathrm{U}_P}(\Lambda(\!|P|\!)) \supseteq \alpha_{\mathrm{I}_P \rightsquigarrow \mathrm{U}_P}(\mathcal{I})$, and from the Galois connection in Equation 7 we have $\Lambda(\!|P|\!) \subseteq \gamma_{\mathrm{I}_P \rightsquigarrow \mathrm{U}_P}(\alpha_{\mathrm{I}_P \rightsquigarrow \mathrm{U}_P}(\mathcal{I}))$. From the definition of $\gamma_{\mathrm{I}_P \rightsquigarrow \mathrm{U}_P}$, we have $\Lambda(\!|P|\!) \subseteq \mathcal{I}$ and we can thus conclude $\Lambda(\!|P|\!) \subseteq \mathcal{I}$. □

3.3. Data Leakage Semantics

As hinted by Example 5, we observe that for detecting data leakage (resp. verifying absence of data leakage), we care in particular about which rows of input data frame variables the used data frame variables depend on. In case of data leakage (resp. absence of data leakage), data frame variables used for different purposes will depend on *overlapping* (resp. *disjoint*) sets of rows of input data frame variables. Thus, we further abstract the dependency semantics $\Lambda_{I \rightsquigarrow U}$ pointwise (Cousot and Cousot, 1994) into a map for each data frame variable associating with each data frame row index the set of (input) variables (indexed at some row) from which it depends on.

Formally, we define the following Galois connection:

$$\langle \mathcal{P}((\mathbb{X} \times \mathbb{N}) \times (\mathbb{X} \times \mathbb{N})), \supseteq \rangle \underset{\dot{\alpha}}{\overset{\dot{\gamma}}{\leftrightarrows}} \langle \mathbb{X} \to (\mathbb{N} \to \mathcal{P}(\mathbb{X} \times \mathbb{N})), \dot{\supseteq} \rangle \tag{9}$$

where the partial order $\dot{\supseteq}$ is the standard point-wise lifting of $\supseteq$, i.e., $m_1 \dot{\supseteq} m_2 \Leftrightarrow \forall x \in \mathbb{X} \colon \forall r \in \mathbb{N} \colon m_1(x)r \supseteq m_2(x)r$, and with $\dot{\alpha} : \mathcal{P}((\mathbb{X} \times \mathbb{N}) \times (\mathbb{X} \times \mathbb{N})) \to (\mathbb{X} \to (\mathbb{N} \to \mathcal{P}(\mathbb{X} \times \mathbb{N})))$, the abstraction function, defined as follows:

$$\dot{\alpha}(D) \stackrel{\text{def}}{=} \lambda x \in \mathbb{X} \colon \left(\lambda r \in \mathbb{N} \colon \left\{ i[r'] \;\middle|\; i \in \mathbb{X}, r' \in \mathbb{N}, i[r'] \rightsquigarrow x[r] \in D \right\}\right) \tag{10}$$

.

Example 6 (Data Leakage Semantics)**.** *Let us consider again program Q in Example 3 and its dependency semantics $\Lambda(\!|Q|\!)_{\rightsquigarrow}$ in Example 5: $\{i[1] \rightsquigarrow o_{\text{train}}[j], i[2] \rightsquigarrow o_{\text{train}}[j], i[3] \rightsquigarrow o_{\text{test}}[j], i[4] \rightsquigarrow o_{\text{test}}[j]\}$, $j \in \{1, 2\}$. Its abstraction $\dot{\alpha}(\Lambda(\!|Q|\!)_{\rightsquigarrow})$, following Equation 10, is the following map:*

$$\lambda x\colon \begin{cases} \lambda r\colon \begin{cases} \{i[1], i[2]\} & r = 1 \\ \{i[1], i[2]\} & r = 2 \end{cases} & x = o_{\text{train}} \\ \lambda r\colon \begin{cases} \{i[3], i[4]\} & r = 1 \\ \{i[3], i[4]\} & r = 2 \end{cases} & x = o_{\text{test}} \end{cases}$$

Instead, the abstraction $\dot{\alpha}(\Lambda(\!|P|\!)_{\rightsquigarrow})$ of the dependency semantics of program P in Example 3 is the following map:

$$\lambda x\colon \begin{cases} \lambda r\colon \begin{cases} \{i[1], i[2], i[3], i[4]\} & r = 1 \\ \{i[1], i[2], i[3], i[4]\} & r = 2 \end{cases} & x = o_{\text{train}} \\ \lambda r\colon \begin{cases} \{i[1], i[2], i[3], i[4]\} & r = 1 \\ \{i[1], i[2], i[3], i[4]\} & r = 2 \end{cases} & x = o_{\text{test}} \end{cases}$$

The abstraction function $\dot{\alpha}$ is another complete join morphism so it uniquely determines the concretization function $\dot{\gamma}\colon (\mathbb{X} \to (\mathbb{N} \to \mathcal{P}(\mathbb{X} \times \mathbb{N}))) \to \mathcal{P}((\mathbb{X} \times \mathbb{N}) \times (\mathbb{X} \times \mathbb{N}))$:

$$\dot{\gamma}(m) \stackrel{\text{def}}{=} \bigcap \{D \mid \dot{\alpha}(D) \mathrel{\dot{\supseteq}} m\}. \tag{11}$$

We finally derive our *data leakage semantics* $\dot{\Lambda} \in \mathbb{X} \to (\mathbb{N} \to \mathcal{P}(\mathbb{X} \times \mathbb{N}))$ by abstraction of $\Lambda_{\rightsquigarrow}$:

$$\dot{\Lambda} \stackrel{\text{def}}{=} \dot{\alpha}(\Lambda_{\rightsquigarrow}). \tag{12}$$

In the following, we write $\dot{\Lambda}(\!|P|\!)$ for the data leakage semantics of a program P. The abstraction $\dot{\alpha}$ does not lose any information, so we still have both soundness and completeness:

Theorem 2. $P \models \mathcal{I} \Leftrightarrow \dot{\Lambda}(\!|P|\!) \mathrel{\dot{\supseteq}} \dot{\alpha}(\alpha_{\mathrm{I}_P \rightsquigarrow \mathrm{U}_P}(\mathcal{I}))$

Proof. The proof is analogous to that of Theorem 1. Let $P \models \mathcal{I}$. We have that $\Lambda(\!|P|\!)_{\rightsquigarrow} \supseteq \alpha_{\mathrm{I}_P \rightsquigarrow \mathrm{U}_P}(\mathcal{I})$ from Theorem 1. From the Galois connection in Equation 9, we have $\dot{\alpha}(\Lambda(\!|P|\!)_{\rightsquigarrow}) \mathrel{\dot{\supseteq}} \dot{\alpha}(\alpha_{\mathrm{I}_P \rightsquigarrow \mathrm{U}_P}(\mathcal{I}))$. Thus, from the definition of $\dot{\Lambda}(\!|P|\!)$, we can conclude that $\dot{\Lambda}(\!|P|\!) \mathrel{\dot{\supseteq}} \dot{\alpha}(\alpha_{\mathrm{I}_P \rightsquigarrow \mathrm{U}_P}(\mathcal{I}))$.

Vice versa, let $\dot{\Lambda}(\!|P|\!) \mathrel{\dot{\supseteq}} \dot{\alpha}(\alpha_{\mathrm{I}_P \rightsquigarrow \mathrm{U}_P}(\mathcal{I}))$. From the definition of $\dot{\Lambda}(\!|P|\!)$ we have $\dot{\alpha}(\Lambda(\!|P|\!)_{\rightsquigarrow}) \mathrel{\dot{\supseteq}} \dot{\alpha}(\alpha_{\mathrm{I}_P \rightsquigarrow \mathrm{U}_P}(\mathcal{I}))$, and from the Galois connection in Equation 9 we have $\Lambda(\!|P|\!)_{\rightsquigarrow} \supseteq \dot{\gamma}(\dot{\alpha}(\alpha_{\mathrm{I}_P \rightsquigarrow \mathrm{U}_P}(\mathcal{I})))$. From the definition of $\dot{\gamma}$ (cf. Equation 11), we have $\Lambda(\!|P|\!)_{\rightsquigarrow} \supseteq \alpha_{\mathrm{I}_P \rightsquigarrow \mathrm{U}_P}(\mathcal{I})$ and from Theorem 1 we can thus conclude $P \models \mathcal{I}$. □

With a slight abuse of notation, let $\dot{\Lambda}(\!|P|\!)o$ stands for $\bigcup_{r \in \mathbb{N}} \dot{\Lambda}(\!|P|\!)o(r)$, i.e., the union of all sets $\dot{\Lambda}(\!|P|\!)o(r)$ of rows of input data frame variables in the range of $\dot{\Lambda}(\!|P|\!)o$ of the data leakage semantics $\dot{\Lambda}(\!|P|\!)$ for a given used data frame variable o. We can now equivalently verify absence of data leakage by checking that data frames used for different purposes depend on disjoint (rows of) input data:

Lemma 1. $P \models \mathcal{I} \Leftrightarrow \forall o_1 \in \mathrm{U}_P^{\text{train}}, o_2 \in \mathrm{U}_P^{\text{test}}\colon \dot{\Lambda}(\!|P|\!)o_1 \cap \dot{\Lambda}(\!|P|\!)o_2 = \emptyset$

Proof. The proof follows immediately from the definition of the absence of data leakage property $\mathcal{I}$ (cf. Equation 5), which requires data frame variables used for training and testing to depend from separate (rows of the) input data sources (cf. Figure 4). From the definition of the data leakage semantics $\dot{\Lambda}$ (cf. Equation 12), this is equivalent to checking that there is no intersection between the input data frame rows that data frame variables used for different purposes depend from. □

Example 7 (Continued from Example 6)**.** *The data leakage semantics $\dot{\Lambda}(\!|Q|\!) \stackrel{def}{=} \dot{\alpha}(\Lambda(\!|Q|\!)_{\rightsquigarrow})$ in Example 6 for program Q in Example 3 satisfies Lemma 1: the set $\{i[1], i[2]\}$ of input data frame rows on which o_{train} depends is* disjoint *from the set $\{i[3], i[4]\}$ of input data frame rows on which o_{test} depends. Thus, performing min-max normalization* after *splitting into train and test data does not create data leakage.*

This is not the case for the data leakage semantics $\dot{\Lambda}(\!|P|\!) \stackrel{def}{=} \dot{\alpha}(\Lambda(\!|P|\!)_{\rightsquigarrow})$ in Example 6 for program P of Example 3, where the sets of input data frame rows from which o_{train} and o_{test} depend are identical, indicating data leakage when normalization is done before *the split into train and test data.*

3.4. Small Data Frame-Manipulating Language

The formal treatment so far is language independent. In the rest of this section, we give a constructive definition of our data leakage semantics $\dot{\Lambda} \in \mathbb{X} \to (\mathbb{N} \to \mathcal{P}(\mathbb{X} \times \mathbb{N}))$ for a small data frame-manipulating language which we then use to illustrate our data leakage analysis in the next section. Note that the actual implementation of the analysis additionally handles more advanced constructs such as conditional branches, loops, and procedures calls, cf. Section 5. However, as these constructs are not critical for reasoning about data leakage, we intentionally do not discuss them in this section to streamline the presentation.

We consider a sequential language without procedures nor references. The only variable data type is the set $\mathbb{D}$ of data frames. Programs in the language are sequences of statements, which belong to either of the following classes:

1. **source**: $y = \text{read}(name)$	$name \in \mathbb{W}$
2. **select**: $y = x.\text{select}[\bar{r}][C]$	$\bar{r} \in \mathbb{N}^{k \leq R_x}, C \subseteq \mathsf{hdr}(x)$
3. **merge**: $y = op(x_1, x_2)$	$x_1, x_2 \in \mathbb{X}, op \in \{\bowtie, \text{concat}, \text{join}\}$
4. **function**: $y = f(x)$	$x \in \mathbb{X}, f \in \{\text{aggregate}, \text{normalize}, \text{other}\}$
5. **use**: $f(X)$	$X \subseteq \mathbb{X}, f \in \{\text{train}, \text{test}\}$

where $\mathbb{W}$ be the set of all possible strings of characters in a chosen alphabet and $name \in \mathbb{W}$ is a (string) data file name; we write R_x and $\mathsf{hdr}(x)$ for the number of rows and set of labels of the columns of the data frame (value) stored into the variable x. The *source* statement (representing library functions such as `read_csv`, `read_excel`, etc., in Pandas) reads data from an input file and stores it into a variable y. For simplicity, we assume that programs do not read the same input file multiple times. The *select* statement (loosely corresponding to library functions such as `iloc`, `loc`, etc., in Pandas) returns a subset data frame y of x, based on an array of row indexes $\bar{r}$ and a set of column labels C . The selection parameters $\bar{r}$ and C are optional: when missing the selection includes all rows or columns of the given data frame. The *merge* statements are binary operations between data frames: $\bowtie \quad \in \quad \{+, -, *, \div, =, \leq, \dots\}$, while the *concat* and *join* operations roughly match the default Pandas `concat` and `merge` library functions, respectively. The *function* statements modify a data frame x either by tainting it (with the *aggregate* or *normalize* functions) or by applying some *other* function. The *aggregate* function represents aggregation functions such as `sum`, `mean`, `min`, or `max` in Pandas, and the *normalize* function represents normalization functions such as standardization or scaling in Sklearn. We assume that any *other* function does not produce tainted data frames. Finally, *use* statements employs data frames for either training (f = train) or testing (f = test) a ML model.

Example 8 (Feature Normalization Leakage (Continued))**.** *The following is the notebook execution in Example 1 written in our small language (where R_X is the number of rows of the data frame stored in the variable X):*

```
1  data = read("credit.csv")
2  X = data.select[][{"Age", "Amount"}]
3  y = data.select[]["Approved"]
4  X = normalize(X)
5  X_train = X.select[[⌊0.025 * R_X⌋ + 1, ..., R_X]][]
6  y_train = y.select[[⌊0.025 * R_y⌋ + 1, ..., R_y]][]
7  X_test = X.select[[0, ..., ⌊0.025 * R_X⌋]][]
8  y_test = y.select[[0, ..., ⌊0.025 * R_y⌋]][]
9  train({X_train, y_train})
10 test({X_test, y_test})
```

Example 9 (Feature Engineering Leakage (Continued))**.** *The following is Example 2 written in our small language:*

```
1  data = read("credit.csv")
2  X1 = data.select[]["Age"]
3  v = data.select[]["Amount"]
4  m = aggregate(data)
5  a = m.select[]["Amount"]
6  X2 = v > a
7  y = data.select[]["Approved"]
8  X1_train = X1.select[[⌊0.025 * R_X1⌋ + 1, ..., R_X1]][]
9  X2_train = X2.select[[⌊0.025 * R_X2⌋ + 1, ..., R_X2]][]
```

$$s[\![y = \text{read}(name)]\!]m \stackrel{\text{def}}{=} m\left[y \mapsto \lambda r \in \mathbb{N}\colon \begin{cases} \{y[r]\langle y, r\rangle\} & r < R_{\text{read}(name)} \\ \emptyset & \text{otherwise} \end{cases}\right]$$

$$s[\![y = x.\text{select}[\bar{r}][C]]\!]m \stackrel{\text{def}}{=} m\left[y \mapsto \lambda r \in \mathbb{N}\colon \begin{cases} m(x)(\bar{r}[r]) & r < R_{x.\text{select}[\bar{r}][C]} \\ \emptyset & \text{otherwise} \end{cases}\right]$$

$$s[\![y = x_1 \bowtie x_2]\!]m \stackrel{\text{def}}{=} m\left[y \mapsto \lambda r \in \mathbb{N}\colon \begin{cases} m(x_1)r \cup m(x_2)r & r < R_{x_1 \bowtie x_2} \\ \emptyset & \text{otherwise} \end{cases}\right]$$

$$s[\![y = \text{concat}(x_1, x_2)]\!]m \stackrel{\text{def}}{=} m\left[y \mapsto \lambda r \in \mathbb{N}\colon \begin{cases} m(x_1)r & r \le |\text{dom}(m(x_1))| \\ m(x_2)(r - |\text{dom}(m(x_1))|) & |\text{dom}(m(x_1))| < r < R_{\text{concat}(x_1,x_2)} \\ \emptyset & \text{otherwise} \end{cases}\right]$$

$$s[\![y = \text{join}(x_1, x_2)]\!]m \stackrel{\text{def}}{=} m\left[y \mapsto \lambda r \in \mathbb{N}\colon \begin{cases} m(x_1)\overleftarrow{r} \cup m(x_2)\overrightarrow{r} & r < R_{\text{join}(x_1,x_2)} \\ \emptyset & \text{otherwise} \end{cases}\right]$$

$$s[\![y = \text{taint}(x)]\!]m \stackrel{\text{def}}{=} m\left[y \mapsto \lambda r \in \mathbb{N}\colon \begin{cases} \bigcup_{r' \in \text{dom}(m(x))} m(x)r' & r < R_{\text{taint}(x)} \\ \emptyset & \text{otherwise} \end{cases}\right] \qquad \text{taint} \in \{\text{aggregate}, \text{normalize}\}$$

$$s[\![y = \text{other}(x)]\!]m \stackrel{\text{def}}{=} m\left[y \mapsto \lambda r \in \mathbb{N}\colon \begin{cases} m(x)r & r < R_{\text{other}(x)} \\ \emptyset & \text{otherwise} \end{cases}\right]$$

$$s[\![\text{use}(x)]\!]m \stackrel{\text{def}}{=} m$$

Figure 6: Constructive data leakage semantics $s[\![S]\!] \in (\mathbb{X} \to (\mathbb{N} \to \mathcal{P}(\mathbb{X} \times \mathbb{N}))) \to (\mathbb{X} \to (\mathbb{N} \to \mathcal{P}(\mathbb{X} \times \mathbb{N})))$ for each statement S in a program P.

```
10  y_train = y.select[[⌊0.025 * R_y⌋ + 1, ..., R_y]][]
11  X1_test = X1.select[[0, ..., ⌊0.025 * R_X1⌋]][]
12  X2_test = X2.select[[0, ..., ⌊0.025 * R_X2⌋]][]
13  y_test = y.select[[0, ..., ⌊0.025 * R_y⌋]][]
14  train({X1_train, X2_train, y_train})
15  test({X1_test, X2_test, y_test})
```

3.5. Constructive Data Leakage Semantics

We can now instantiate the definition of our data leakage semantics $\dot{\Lambda} \in \mathbb{X} \to (\mathbb{N} \to \mathcal{P}(\mathbb{X} \times \mathbb{N}))$ with our small data frame-manipulating language. Given a program $P \equiv S_1, \ldots, S_n$ written in our small language (where $S_1, \ldots, S_n$ are statements), the set of its input data frame variables I_P contains the data frame variables assigned by *source* statements in the program, i.e., $\text{I}_P \stackrel{\text{def}}{=} i[\![P]\!] = i[\![S_n]\!] \circ \cdots \circ i[\![S_1]\!]\emptyset$, where

$$i[\![y = \text{read}(name)]\!]I \stackrel{\text{def}}{=} I \cup \{y\}$$

and $i[\![S]\!]I \stackrel{\text{def}}{=} I$ for any other statement S in P. Similarly, we define $\text{U}_P \stackrel{\text{def}}{=} u[\![P]\!] = u[\![S_n]\!] \circ \cdots \circ u[\![S_1]\!]\emptyset$, the the set of used variables, where

$$u[\![f(X)]\!]U \stackrel{\text{def}}{=} U \cup X$$

and $u[\![S]\!]U \stackrel{\text{def}}{=} U$ for any other statement S, and analogously for $\text{U}_P^{\text{train}} \subseteq \text{U}_P$ (if $f = \text{train}$) and $\text{U}_P^{\text{test}} \subseteq \text{U}_P$ (if $f = \text{test}$).

Our constructive data leakage semantics is:

$$(\!|\dot{P}|\!) \stackrel{\text{def}}{=} s[\![S_n]\!] \circ \cdots \circ s[\![S_1]\!]\dot{\emptyset} \tag{13}$$

where $\dot{\emptyset}$ is $\lambda x \in \mathbb{X}\colon \lambda r \in \mathbb{N}\colon \emptyset$ and the semantic function $s[\![S]\!] \in (\mathbb{X} \to (\mathbb{N} \to \mathcal{P}(\mathbb{X} \times \mathbb{N}))) \to (\mathbb{X} \to (\mathbb{N} \to \mathcal{P}(\mathbb{X} \times \mathbb{N})))$ for each statement S in P is defined in Figure 6. Given a map $m \in \mathbb{X} \to (\mathbb{N} \to \mathcal{P}(\mathbb{X} \times \mathbb{N}))$, we write $m[y \leftarrow d]$ to

replace the mapping of y in m with d:

$$m[y \leftarrow d] \stackrel{\text{def}}{=} \lambda x\colon \begin{cases} d & x = y \\ m(x) & \text{otherwise} \end{cases}$$

The semantics of the *source* statement maps each row r of an input data frame y to (the set containing) y indexed at row r. The semantics of the *select* statement maps each row r of the resulting data frame y to the set of data sources ($m(x)$) of the corresponding row ($\bar{r}[r]$) in the original data frame. The semantics of binary operations $\bowtie$ between two data frames x_1 and x_2 maps each row r of the resulting data frame y to the union of the sets of data sources of row r in x_1 and x_2. The *concat* between data frames x_1 and x_2 yields a data frame with all rows of x_1 followed by all rows of x_2. Thus, the semantics of *concat* statements accordingly maps each row r of the resulting data frame y to the set of data sources of the corresponding row in x_1 (if $r \leq |\text{dom}(m(x_1))|$, that is, r falls within the size of x_1) or x_2 (if $r > |\text{dom}(m(x_1))|$). Instead, the *join* operation combines two data frames x_1 and x_2 based on a(n index) column and yields a data frame containing only the rows that have a matching value in both x_1 and x_2. Thus, the semantics of *join* statements maps each row r of the resulting data frame y to the union of the sets of data sources of the corresponding rows in x_1 and x_2: $\overleftarrow{r}$ (resp. $\overrightarrow{r}$) refers to the row in x_1 (resp. x_2) that participated in the join to produce row r. We consider only one type of join operation (inner join) for simplicity, but other types (outer, left, or right join) can be similarly defined. The semantics for a *taint* function, with *taint* $\in$ {aggregate, normalize}, introduces dependencies for each row r in the normalized data frame y with the data sources ($m(x)$) of each row r' of the data frame before normalization. Instead, the semantics of *other* (non-tainting) functions maintains the same dependencies ($m(x)r$) for each row r of the modified data frame y. Finally, *use* statements do not modify any dependency so the semantics of *use* statements leaves the dependencies map unchanged.

Example 10 (Feature Normalization Leakage (Continued))**.** *The data leakage semantics for the notebook execution in Example 8 is the following, where R is the number of rows in the CSV file "credit.csv":*

$$s[\![\texttt{data = read("credit.csv")}]\!]\dot{\emptyset} = \left(m_1 \stackrel{def}{=} \lambda x\colon \begin{cases} \lambda r\colon \begin{cases} \{data[r]\} & r<R \\ \emptyset & otherwise \end{cases} & x = data \\ undefined & otherwise \end{cases} \right)$$

In the map m_1 resulting after the first (source) statement each row r of the data frame variable `data` *only depends on itself (i.e., the corresponding row in the "credit.csv" file).*

$$s[\![\texttt{X = data.select[][{"Age", "Amount"}]}]\!]m_1 = \left(m_2 \stackrel{def}{=} m_1 \left[X \mapsto \lambda r\colon \begin{cases} \{data[r]\} & r<R \\ \emptyset & otherwise \end{cases} \right] \right)$$

$$s[\![\texttt{y = data.select[][{"Approved"}]}]\!]m_2 = \left(m_3 \stackrel{def}{=} m_2 \left[y \mapsto \lambda r\colon \begin{cases} \{data[r]\} & r<R \\ \emptyset & otherwise \end{cases} \right] \right)$$

The column selection does not alter the dependencies: in the map m_3, each row of `X` *and* `y` *still depends on the corresponding row in the input CSV file.*

$$s[\![\texttt{X = normalize(X)}]\!]m_3 = \left(m_4 \stackrel{def}{=} m_3 \left[X \mapsto \lambda r\colon \begin{cases} \bigcup_{r' \in \{0,\ldots,R\}} \{data[r']\} & r<R \\ \emptyset & otherwise \end{cases} \right] \right)$$

The normalize *function introduces dependencies between each row r of* `X` *and all rows of the input CSV file.*

$$s[\![\texttt{X\_train = X.select[}[\lfloor 0.025 * R_X \rfloor \texttt{ + 1, ..., } R_X\texttt{]][]}]\!]m_4 = \left(m_5 \stackrel{def}{=} m_4 \left[X\_\text{train} \mapsto \lambda r\colon \begin{cases} \bigcup_{r' \in \{0,\ldots,R\}} \{data[r']\} & \lfloor 0.025*R \rfloor + 1 < r < R \\ \emptyset & otherwise \end{cases} \right] \right)$$

$$s[\![\texttt{y\_train = y.select[}[\lfloor 0.025 * R_y \rfloor \texttt{ + 1, ..., } R_y\texttt{]][]}]\!]m_5 = \left(m_6 \stackrel{def}{=} m_5 \left[y\_\text{train} \mapsto \lambda r\colon \begin{cases} \{data[r]\} & \lfloor 0.025*R \rfloor + 1 < r < R \\ \emptyset & otherwise \end{cases} \right] \right)$$

$$s[\![\texttt{X\_test = X.select[[0, ..., } \lfloor 0.025 * R_X \rfloor \texttt{]] []}]\!] m_6 = \left(m_7 \stackrel{def}{=} m_6 \left[X\_\text{test} \mapsto \lambda r\text{: } \begin{cases} \bigcup_{r' \in \{0,\ldots,R\}} \{data[r']\} & r < \lfloor 0.025 * R \rfloor \\ \emptyset & otherwise \end{cases} \right] \right)$$

$$s[\![\texttt{y\_test = y.select[[0, ..., } \lfloor 0.025 * R_y \rfloor \texttt{]] []}]\!] m_7 = \left(m_8 \stackrel{def}{=} m_7 \left[y\_\text{test} \mapsto \lambda r\text{: } \begin{cases} \{data[r]\} & r < \lfloor 0.025 * R \rfloor \\ \emptyset & otherwise \end{cases} \right] \right)$$

$$s[\![\texttt{train(\{X\_train, y\_train\})}]\!] m_8 = m_8$$

$$s[\![\texttt{test(\{X\_test, y\_test\})}]\!] m_8 = m_8$$

In the final map m_8, we have $\bigcup_{r \in \mathbb{N}} m_8(o)r = \bigcup_{r' \in \{0,\ldots,R\}} \{data[r']\}$ for each used variable $o \in \{\texttt{X\_train}, \texttt{X\_test}\}$. Thus, $\bigcup_{r \in \mathbb{N}} m_8(\texttt{X\_train})r \cap \bigcup_{r \in \mathbb{N}} m_8(\texttt{X\_test})r \neq \emptyset$ and, from Lemma 1, the program does not satisfy the absence of data leakage property $\mathcal{I}$.

Example 11 (Feature Engineering Leakage (Continued))**.** *The data leakage semantics for Example 9 is the following (again, R is the number of rows in "credit.csv"):*

$$s[\![\texttt{data = read("credit.csv")}]\!] \dot{\emptyset} = \left(m_1 \stackrel{def}{=} \lambda x\text{: } \begin{cases} \lambda r\text{: } \begin{cases} \{data[r]\} & r < R \\ \emptyset & otherwise \end{cases} & x = data \\ undefined & otherwise \end{cases} \right)$$

$$s[\![\texttt{X1 = data.select[] [\{"Age"\}]}]\!] m_1 = \left(m_2 \stackrel{def}{=} m_1 \left[X1 \mapsto \lambda r\text{: } \begin{cases} \{data[r]\} & r < R \\ \emptyset & otherwise \end{cases} \right] \right)$$

$$s[\![\texttt{v = data.select[] [\{"Amount"\}]}]\!] m_2 = \left(m_3 \stackrel{def}{=} m_2 \left[v \mapsto \lambda r\text{: } \begin{cases} \{data[r]\} & r < R \\ \emptyset & otherwise \end{cases} \right] \right)$$

$$s[\![\texttt{m = aggregate(data)}]\!] m_3 = \left(m_4 \stackrel{def}{=} m_3 \left[m \mapsto \lambda r\text{: } \begin{cases} \bigcup_{r' \in \{0,\ldots,R\}} \{data[r']\} & r = 0 \\ \emptyset & otherwise \end{cases} \right] \right)$$

The aggregate *function collapses the data frame stored into* `data` *to a single row which depends from all rows of the input CSV file.*

$$s[\![\texttt{a = m.select[] [\{"Amount"\}]}]\!] m_4 = \left(m_5 \stackrel{def}{=} m_4 \left[X \mapsto \lambda r\text{: } \begin{cases} \bigcup_{r' \in \{0,\ldots,R\}} \{data[r']\} & r = 0 \\ \emptyset & otherwise \end{cases} \right] \right)$$

$$s[\![\texttt{X2 = v > a}]\!] m_5 = \left(m_6 \stackrel{def}{=} m_5 \left[X2 \mapsto \lambda r\text{: } \begin{cases} \bigcup_{r' \in \{0,\ldots,R\}} \{data[r']\} & r < R \\ \emptyset & otherwise \end{cases} \right] \right)$$

$$s[\![\texttt{y = data.select[] [\{"Approved"\}]}]\!] m_6 = \left(m_7 \stackrel{def}{=} m_6 \left[y \mapsto \lambda r\text{: } \begin{cases} \{data[r]\} & r < R \\ \emptyset & otherwise \end{cases} \right] \right)$$

$$s[\![\texttt{X1\_train = X1.select[[}\lfloor 0.025 * R_{X1} \rfloor \texttt{ + 1, ..., } R_{X1} \texttt{]] []}]\!] m_7 = \left(m_8 \stackrel{def}{=} m_7 \left[X\_\text{train} \mapsto \lambda r\text{: } \begin{cases} \{data[r]\} & \lfloor 0.025 * R \rfloor + 1 < r < R \\ \emptyset & otherwise \end{cases} \right] \right)$$

$$s[\![\texttt{X2\_train = X2.select[[}\lfloor 0.025 * R_{X2} \rfloor \texttt{ + 1, ..., } R_{X2} \texttt{]] []}]\!] m_8 = \left(m_9 \stackrel{def}{=} m_8 \left[X\_\text{train} \mapsto \lambda r\text{: } \begin{cases} \bigcup_{r' \in \{0,\ldots,R\}} \{data[r']\} & \lfloor 0.025 * R \rfloor + 1 < r < R \\ \emptyset & otherwise \end{cases} \right] \right)$$

$$s[\![\texttt{y\_train = y.select[[}\lfloor 0.025 * R_y \rfloor \texttt{ + 1, ..., } R_y \texttt{]] []}]\!] m_9 = \left(m_{10} \stackrel{def}{=} m_9 \left[y\_\text{train} \mapsto \lambda r\text{: } \begin{cases} \{data[r]\} & \lfloor 0.025 * R \rfloor + 1 < r < R \\ \emptyset & otherwise \end{cases} \right] \right)$$

$$s[\![\texttt{X1\_test = X1.select[[0, ..., } \lfloor 0.025 * R_X \rfloor \texttt{]] []}]\!] m_{10} = \left(m_{11} \stackrel{def}{=} m_{10} \left[X\_\text{test} \mapsto \lambda r\text{: } \begin{cases} \{data[r]\} & r < \lfloor 0.025 * R \rfloor \\ \emptyset & otherwise \end{cases} \right] \right)$$

$$s[\![\texttt{X2\_test = X2.select[[0, ..., } \lfloor 0.025 * R_X \rfloor \texttt{]] []}]\!] m_{11} = \left(m_{12} \stackrel{def}{=} m_{11} \left[X\_\text{test} \mapsto \lambda r\text{: } \begin{cases} \bigcup_{r' \in \{0,\ldots,R\}} \{data[r']\} & r < \lfloor 0.025 * R \rfloor \\ \emptyset & otherwise \end{cases} \right] \right)$$

$$s\llbracket \texttt{y\_test = y.select[[0, ..., } \lfloor 0.025 * R_y \rfloor \texttt{]]]} \rrbracket m_{12} = \left(m_{13} \stackrel{def}{=} m_{12} \left[y\_\text{test} \mapsto \lambda r\text{: } \begin{cases} \{data[r]\} & r < \lfloor 0.025 * R \rfloor \\ \emptyset & otherwise \end{cases} \right] \right)$$

$$s\llbracket \texttt{train(\{X\_train, y\_train\})} \rrbracket m_{13} = m_{13}$$

$$s\llbracket \texttt{test(\{X\_test, y\_test\})} \rrbracket m_{13} = m_{13}$$

In the final map m_{13}, we have $\bigcup_{r\in\mathbb{N}} m_{13}(X2\_\text{train})r \cap \bigcup_{r\in\mathbb{N}} m_{13}(X2\_\text{test})r \neq \emptyset$ and, from Lemma 1, this program as well does not satisfy the absence of data leakage property $\mathcal{I}$.

Remark.. For simplicity, we assumed that programs do not read the same input file multiple times. If we remove such assumption, we need to additionally maintain a map $F_P \in \mathrm{I}_P \to \mathbb{W}$ from input data frame variables and the corresponding data file names ($F_P(y) = \mathit{name}$ if program P contains the source statement $y = \text{read}(\mathit{name})$), and use it to perform a replacement on the data sources of the data leakage semantics of the program, i.e., yielding $\dot{\Lambda}(\!|P|\!) \in \mathbb{X} \to (\mathbb{N} \to \mathcal{P}(\mathbb{W} \times \mathbb{N}))$, before applying Lemma 1.

4. Data Leakage Analysis

In this section, we abstract our concrete data leakage semantics to obtain a sound data leakage static analysis. In essence, our analysis keeps track of (an over-approximation of) the data source cells each data frame variable depends on (to detect potential explicit data source overlaps). Plus, it tracks whether data source cells are tainted, i.e., modified by a library function in such a way that could introduce data leakage (by implicit indirect data source overlaps).

4.1. Data Sources Abstract Domain

4.1.1. Data Frame Abstract Domain.

We over-approximate data sources by means of a parametric data frame abstract domain $\mathbb{L}(\mathbb{C}, \mathbb{R})$, where the parameter abstract domains $\mathbb{C}$ and $\mathbb{R}$ track data sources columns and rows, respectively. We illustrate below two simple instances of these domains.

Column Abstraction. We propose an instance of $\mathbb{C}$ that over-approximates the set of column labels in a data frame. As, in practice, data frame labels are pretty much always strings, the elements of $\mathbb{C}$ belong to a complete lattice $\langle C, \sqsubseteq_C, \sqcup_C, \sqcap_C, \bot_C, \top_C \rangle$ where $C \stackrel{\text{def}}{=} \mathcal{P}(\mathbb{W}) \cup \{\top_C\}$; $\mathbb{W}$ is the set of all possible strings of characters in a chosen alphabet and $\top_C$ represents a lack of information on which columns a data frame *may* have (abstracting any data frame). Elements in C are ordered by set inclusion extended with $\top_C$ being the largest element: $C_1 \sqsubseteq_C C_2 \stackrel{\text{def}}{\Leftrightarrow} C_2 = \top_C \vee (C_1 \neq \top_C \wedge C_1 \subseteq C_2)$. Similarly, join $\sqcup_C$ and meet $\sqcap_C$ are set inclusion and set intersection, respectively, extended to account for $\top_C$:

$$C_1 \sqcup_C C_2 \stackrel{\text{def}}{=} \begin{cases} \top_C & C_1 = \top_C \vee C_2 = \top_2 \\ C_1 \cup C_2 & \text{otherwise} \end{cases} \qquad C_1 \sqcap_C C_2 \stackrel{\text{def}}{=} \begin{cases} C_1 & C_2 = \top_C \\ C_2 & C_1 = \top_C \\ C_1 \cap C_2 & \text{otherwise} \end{cases}$$

Finally, the bottom $\bot_C$ is the empty set $\emptyset$ (abstracting an empty data frame).

If we were to extend our small language in Section 3.4 with a **drop**: $y = x.\text{drop}[\bar{r}][C]$ statement – roughly modeling the `drop` function in Pandas – it would be useful to complement our column abstraction with an under-approximation $\overline{\mathbb{C}}$ of the surely missing column labels in a data frame. The elements of $\overline{\mathbb{C}}$ belong to the complete lattice $\langle C, \preccurlyeq_C, \curlyvee_C, \curlywedge_C, \top_C, \bot_C \rangle$ where $C_1 \preccurlyeq_C C_2 \stackrel{\text{def}}{\Leftrightarrow} C_1 = \top_C \vee (C_2 \neq \top_C \wedge C_2 \subseteq C_1)$, i.e., C_1 is more precise than C_2 when C_1 contains *more* missing column labels than C_2. Join $\curlyvee_C$ and meet $\curlywedge_C$ are defined accordingly:

$$C_1 \curlyvee_C C_2 \stackrel{\text{def}}{=} \begin{cases} C_1 & C_2 = \top_C \\ C_2 & C_1 = \top_C \\ C_1 \cap C_2 & \text{otherwise} \end{cases} \qquad C_1 \curlywedge_C C_2 \stackrel{\text{def}}{=} \begin{cases} \top_C & C_1 = \top_C \vee C_2 = \top_2 \\ C_1 \cup C_2 & \text{otherwise} \end{cases}$$

An empty data frame is represented by the top element $\top_C$ (i.e., all column labels are missing), while the bottom $\bot_C$ represents a full data frame, in which the set of missing columns is the empty set $\emptyset$ (i.e., no column labels are missing).

Row Abstraction. Unlike columns, data frame rows are not named. Moreover, data frames typically have a large number of rows and often ranges or rows are added to or removed from data frames. Thus, the abstract domain of intervals Cousot and Cousot (1976) *over the natural numbers* is a suitable instance of $\mathbb{R}$. The elements of $\mathbb{R}$ belong to the complete lattice $\langle \mathcal{R}, \sqsubseteq_R, \sqcup_R, \sqcap_R, \bot_R, \top_R \rangle$ with the set $\mathcal{R}$ defined as $\mathcal{R} \stackrel{\text{def}}{=} \{[l,u] \mid l \in \mathbb{N}, u \in \mathbb{N} \cup \{\infty\}, l \leq u\} \cup \{\bot_R\}$. The top element $\top_R$ is $[0, \infty]$. Intervals in $\mathbb{R}$ abstract (sets of) row indexes: the concretization function $\gamma_R \colon \mathcal{R} \to \mathcal{P}(\mathbb{N})$ is such that $\gamma_R(\bot_R) \stackrel{\text{def}}{=} \emptyset$ and $\gamma_R([l,u]) \stackrel{\text{def}}{=} \{r \in \mathbb{N} \mid l \leq r \leq u\}$. The interval domain partial order ($\sqsubseteq_R$) and operators for join ($\sqcup_R$) and meet ($\sqcap_R$) are defined as usual (e.g., see Miné's PhD thesis Miné (2004) for reference). Standard interval widening Cousot and Cousot (1977) can be used to enforce termination if required in presence of loops, e.g., code that iterates over data frame rows, which is not common. Indeed, in our experimental evaluation (cf. Section 6) we did not experience infinite (or very high) ascending chains requiring a widening when analyzing our benchmarks.

In addition, we associate with each interval $R \in \mathcal{R}$ another interval $\mathsf{idx}(R)$ of indices: $\mathsf{idx}(\bot_R) \stackrel{\text{def}}{=} \bot_R$ and $\mathsf{idx}([l,u]) \stackrel{\text{def}}{=} [0, u-l]$; this essentially establishes a map $\phi_R \colon \mathbb{N} \to \mathbb{N}$ between elements of $\gamma_R(R)$ (ordered by $\leq$) and elements of $\gamma_R(\mathsf{idx}(R))$ (also ordered by $\leq$). In the following, given an interval $R \in \mathcal{R}$ and an interval of indices $[i,j] \in \mathcal{R}$ (such that $[i,j] \sqsubseteq_R R$), we slightly abuse notation and write $\phi_R^{-1}([i,j])$ for the sub-interval of R between the indices i and j, i.e., we have that $\gamma_R(\phi_R^{-1}([i,j])) \stackrel{\text{def}}{=} \left\{r \in \gamma(R) \mid \phi^{-1}(i) \leq r \leq \phi^{-1}(j)\right\}$. We need this operation to soundly abstract consecutive row selections (cf. Section 4).

Example 12 (Row Abstraction)**.** *Let us consider the interval* $[10,14] \in \mathcal{R}$ *with index* $\mathsf{idx}(R) = [0,4]$*. We have an isomorphism* ϕ_R *between* $\{10,11,12,13,14\}$ *and* $\{0,1,2,3,4\}$*. Let us consider now the interval of indices* $[1,3]$*. We then have* $\phi_R^{-1}([1,3]) = [11,13]$ *(since* $\phi_R^{-1}(1) = 11$ *and* $\phi_R^{-1}(3) = 13$*).*

Disjunctive intervals Cousot and Cousot (1992); Gange et al. (2021) offer an alternative instance of $\mathbb{R}$ that is much more expressive (and, thus, precise) but also more costly. In our experimental evaluations (cf. Section 6) we did not find the need for this added cost.

Note that, by default, the `train_test_split` function in Sklearn shuffles the data rows before splitting. This is not an issue for our analysis as it does not keep track of the data frame contents (the tracked dependencies are in fact between data row indices, abstracting away their data values). If we were to abstract data frame contents as well, we can take shuffling into account by complementing our abstract domain $\mathbb{R}$ with a SHUFFLED boolean flag, to track row dependencies modulo shuffling of the values of the data stored into those rows.

Data Frame Abstraction. The elements of the data frame abstract domain $\mathbb{L}(\mathbb{C}, \mathbb{R})$ belong to a partial order $\langle \mathcal{L}, \sqsubseteq_L \rangle$ where $\mathcal{L} \stackrel{\text{def}}{=} \mathbb{X} \times C \times \mathcal{R}$ contains triples of an *input* data frame variable $X \in \mathbb{X}$, a column over-approximation $C \in C$, and a row over-approximation $R \in \mathcal{R}$. In the following, we write X_R^C for the abstract data frame $\langle X, C, R \rangle \in \mathcal{L}$. The partial order $\sqsubseteq_L$ compares abstract data frames derived from the same data files: $X_R^C \sqsubseteq_L Y_{R'}^{C'} \stackrel{\text{def}}{\Leftrightarrow} X = Y \wedge C \sqsubseteq_C C' \wedge R \sqsubseteq R'$.

We also define a predicate for whether abstract data frames overlap:

$$\text{overlap}(X_R^C, Y_{R'}^{C'}) \stackrel{\text{def}}{\Leftrightarrow} X = Y \wedge C \sqcap_C C' \neq \emptyset \wedge R \sqcap R' \neq \bot_R \tag{14}$$

and partial join ($\sqcup_L$) and meet ($\sqcap_L$) over data frames from the same data files:

$$X_{R_1}^{C_1} \sqcup_L X_{R_2}^{C_2} \stackrel{\text{def}}{=} X_{R_1 \sqcup_R R_2}^{C_1 \sqcup_C C_2} \qquad X_{R_1}^{C_1} \sqcap_L X_{R_2}^{C_2} \stackrel{\text{def}}{=} X_{R_1 \sqcap_R R_2}^{C_1 \sqcap_C C_2}$$

Finally, we define a constraining operator $\downarrow_R^C$ that restricts an abstract data frame to given column and row over-approximations: $X_R^C \downarrow_{R'}^{C'} \stackrel{\text{def}}{=} X_{\phi^{-1}(\mathsf{idx}(R) \sqcap_R R')}^{C \sqcap_C C'}$. Note that here the definition makes use of $\mathsf{idx}(R)$ and $\phi^{-1}([i,j])$ to compute the correct row over-approximation.

Example 13 (Abstract Data Frames)**.** *Let* $X_{[10,14]}^{\{id,city\}}$ *abstract a data frame with rows* $\{10,11,12,13,14\}$ *and columns* $\{id, city\}$*. The abstract data frame* $X_{[12,15]}^{\{country\}}$ *does not overlap with it, while* $X_{[12,15]}^{\{id\}}$ *does. Joining* $X_{[10,14]}^{\{id,city\}}$ *with* $X_{[12,15]}^{\{country\}}$ *yields* $X_{[10,15]}^{\{id,city,country\}}$*. Instead, the meet with* $X_{[12,15]}^{\{id\}}$ *yields* $X_{[12,14]}^{\{id\}}$*. Finally, the constraining* $X_{[10,15]}^{\{id,city,country\}} \downarrow_{[1,2]}^{\{city\}}$ *results in* $X_{[11,12]}^{\{city\}}$ *(since* $\phi_{[10,15]}^{-1}(1) = 11$ *and* $\phi_{[10,15]}^{-1}(2) = 12$*).*

In the rest of this section, for brevity, we simply write $\mathbb{L}$ instead of $\mathbb{L}(\mathbb{C}, \mathbb{R})$.

4.1.2. Data Frame Set Abstract Domain.

Data frame variables may depend on multiple data sources. We thus lift our abstract domain $\mathbb{L}$ to an abstract domain $\mathbb{S}(\mathbb{L})$ of sets of abstract data frames. The elements of $\mathbb{S}(\mathbb{L})$ belong to a lattice $\langle \mathcal{S}, \sqsubseteq_S, \sqcup_S, \sqcap_S \rangle$ with $\mathcal{S} \stackrel{\text{def}}{=} \mathcal{P}(\mathcal{L})$. Sets of abstract data frames in $\mathcal{S}$ are maintained in a *canonical form* such that no abstract data frames in a set can be overlapping (cf. Equation 14). The partial order $\sqsubseteq_S$ between canonical sets relies on the partial order between abstract data frames: $S_1 \sqsubseteq_S S_2 \stackrel{\text{def}}{\Leftrightarrow} \forall L_1 \in S_1 \exists L_2 \in S_2 \colon L_1 \sqsubseteq_L L_2$.

The join ($\sqcup_S$) and meet ($\sqcap_S$) operators perform a set union and set intersection, respectively, followed by a reduction to make the result canonical:

$$S_1 \sqcup_S S_2 \stackrel{\text{def}}{=} \textsc{reduce}^{\sqcup_L}(S_1 \cup S_2) \qquad S_1 \sqcap_S S_2 \stackrel{\text{def}}{=} \textsc{reduce}^{\sqcap_L}(S_1 \cap S_2)$$

where $\textsc{reduce}^{op}(S) \stackrel{\text{def}}{=} \{L_1 op L_2 \mid L_1, L_2 \in S, \text{overlap}(L_1, L_2)\} \cup \{L_1 \in S \mid \forall L_2 \in S \setminus \{L_1\} : \neg\text{overlap}(L_1, L_2)\}$.

Finally, we lift $\downarrow_R^C$ by element-wise application: $S \downarrow_R^{C\,\text{def}}= \left\{ L \downarrow_R^C \mid L \in S \right\}$.

Example 14 (Abstract Data Frame Sets)**.** *Let us consider the join of abstract data frame sets* $S_1 = \left\{ X1^{\{id\}}_{[1,10]}, X2^{\{name\}}_{[0,100]} \right\}$ *and* $S_2 = \left\{ X1^{\{id\}}_{[9,12]}, X3^{\{zip\}}_{[0,100]} \right\}$*. Before reduction, we obtain* $\left\{ X1^{\{id\}}_{[1,10]}, X1^{\{id\}}_{[9,12]}, X2^{\{name\}}_{[0,100]}, X3^{\{zip\}}_{[0,100]} \right\}$*. The reduction operation makes the set canonical:* $\left\{ X1^{\{id\}}_{[1,12]}, X2^{\{name\}}_{[0,100]}, X3^{\{zip\}}_{[0,100]} \right\}$.

In the following, for brevity, we omit $\mathbb{L}$ and simply write $\mathbb{S}$ instead of $\mathbb{S}(\mathbb{L})$.

4.1.3. Data Frame Sources Abstract Domain.

We can now define the domain $\mathbb{X} \to \mathbb{A}(\mathbb{S})$ that we use for our data leakage analysis. Elements in this abstract domain are maps from data frame variables in $\mathbb{X}$ to elements of a data frame sources abstract domain $\mathbb{A}(\mathbb{S})$, which over-approximates the (input) data frame variables (indexed at some row) from which a data frame variable depends.

In particular, elements in $\mathbb{A}(\mathbb{S})$ belong to a lattice $\langle \mathcal{A}, \sqsubseteq_A, \sqcup_A, \sqcap_A, \bot_A \rangle$ where $\mathcal{A} \stackrel{\text{def}}{=} \mathcal{S} \times \mathbb{B}$ contains pairs $\langle S, B \rangle$ of a data frame set abstraction in $S \in \mathcal{S}$ and a boolean flag in $B \in \mathbb{B} \stackrel{\text{def}}{=} \{\textsc{untainted}, \textsc{maybe-tainted}\}$. In the following, given an abstract element $m \in \mathbb{X} \to \mathcal{A}$ of $\mathbb{X} \to \mathbb{A}(\mathbb{S})$ and a data frame variable $x \in \mathbb{X}$, we write $m_s(x) \in \mathcal{S}$ and $m_b(x) \in \mathbb{B}$ for the first and second component of the pair $m(x) \in \mathcal{A}$, respectively.

The abstract domain operators apply component operators pairwise: $\sqsubseteq_A \stackrel{\text{def}}{=} \sqsubseteq_S \times \leq$, $\sqcup_A \stackrel{\text{def}}{=} \sqcup_S \times \vee$, $\sqcap_A \stackrel{\text{def}}{=} \sqcap_S \times \wedge$, where $\leq$ in $\mathbb{B}$ is such that $\textsc{untainted} \leq \textsc{maybe-tainted}$. The bottom element $\bot_A$ is $\langle \emptyset, \textsc{untainted} \rangle$.

Finally, we define the concretization function $\gamma \colon (\mathbb{X} \to \mathcal{A}) \to (\mathbb{X} \to (\mathbb{N} \to \mathcal{P}(\mathbb{X} \times \mathbb{N})))$:

$$\gamma(m) \stackrel{\text{def}}{=} \lambda x \in \mathbb{X} \colon (\lambda r \in \mathbb{N} \colon \gamma_A(m(x))) \tag{15}$$

where $\gamma_A \colon \mathcal{A} \to \mathcal{P}(\mathbb{X} \times \mathbb{N})$ is $\gamma_A(\langle S, B \rangle) \stackrel{\text{def}}{=} \left\{ X[r] \mid X^C_R \in S, r \in \gamma_R(R) \right\}$ (with $\gamma_R \colon \mathcal{R} \to \mathcal{P}(\mathbb{N})$ being the concretization function for row abstractions, cf. Section 4.1.1). Note that, γ_A does not use $B \in \mathbb{B}$ nor $C \in \mathcal{C}$. These are devices uniquely needed by our abstract semantics (that we define below) to track (and approximate the concrete actual) dependencies across program statements.

4.2. Abstract Data Leakage Semantics

Our data leakage analysis is given by $(\!|\dot{P}|\!)^\sharp \stackrel{\text{def}}{=} a[\![S_n]\!] \circ \cdots \circ a[\![S_1]\!] \dot{\bot}_A$ where $\dot{\bot}_A$ maps all data frame variables to $\bot_A$ and the abstract semantic function $a[\![S]\!] \in (\mathbb{X} \to \mathcal{A}) \to (\mathbb{X} \to \mathcal{A})$ for each statement in P is defined in Figure 7. The abstract semantics of the *source* statement simply maps a read data frame variable y to the untainted abstract data frame set containing the abstraction of the read data file ($y^{\top_C}_{[0,\infty]}$). The abstract semantics of the *select* statement maps the resulting data frame variable y to the abstract data frame set $m_s(x)$ associated with the original data frame variable x; in order to soundly propagate (abstract) dependencies, $m_s(x)$ is constrained by $\downarrow^C_{[\min(\bar{r}),\max(\bar{r})]}$ (cf. Section 4.1.2) only if $m_s(x)$ is untainted. The abstract semantics of *merge* statements merges the abstract data frame sets $m_s(x_1)$ and $m_s(x_2)$ and taint flags $m_b(x_1)$ and $m_b(x_2)$ associated with the given data frame variables x_1 and x_2. Note that such semantics is a sound but rather imprecise abstraction, in particular, for the *join* operation. More precise abstractions can be easily defined, at the cost of also abstracting data frame contents. The abstract semantics of *function* statements

$$a[\![y = \text{read}(name)]\!]m \stackrel{\text{def}}{=} m\left[y \mapsto \langle \left\{ y^{\top_C}_{[0,\infty]} \right\}, \textsc{false} \rangle\right]$$

$$a[\![y = x.\text{select}[\bar{r}][C]]\!]m \stackrel{\text{def}}{=} m\left[y \mapsto \begin{cases} \langle m_s(x) \downarrow^C_{[\min(\bar{r}),\max(\bar{r})]}, m_b(x)\rangle & \neg m_b(x) \\ \langle m_s(x), m_b(x)\rangle & \text{otherwise} \end{cases}\right]$$

$$a[\![y = \text{op}(x_1, x_2)]\!]m \stackrel{\text{def}}{=} m\left[y \mapsto \langle m_s(x_1) \sqcup_S m_s(x_2), m_b(x_1) \vee m_b(x_2)\rangle\right]$$

$$a[\![y = \text{taint}(x)]\!]m \stackrel{\text{def}}{=} m\left[y \mapsto \langle m_s(x), \textsc{true}\rangle\right] \qquad \text{taint} \in \{\text{aggregate}, \text{normalize}\}$$

$$a[\![y = \text{other}(x)]\!]m \stackrel{\text{def}}{=} m\left[y \mapsto \langle m_s(x), m_b(x)\rangle\right]$$

$$a[\![\text{use}(x)]\!]m \stackrel{\text{def}}{=} m$$

Figure 7: Abstract data leakage semantics $a[\![S]\!] \in (\mathbb{X} \to \mathcal{A}) \to (\mathbb{X} \to \mathcal{A})$ for each statement S in a program P.

maps the resulting data frame variable y to the abstract data frame set $m_s(x)$ associated with the original data frame variable x; the *aggregate* and *normalize* functions sets the taint flag to TRUE, while *other* functions leave the taint flag $m_b(x)$ unchanged. Note that, unlike the analysis sketched by Subotić et al. (2022), we do not perform any renaming or resetting of the data source mapping (cf. Section 4.2 in (Subotić et al., 2022)) but we keep tracking dependencies with respect to the input data frame variables. Finally, the abstract semantics of *use* statements leave the abstract dependencies map unchanged.

The abstract data leakage semantics $(\!|\dot{P}|\!)^\natural$ is *sound*:

Theorem 3. $P \models \mathcal{I} \Leftarrow \gamma((\!|\dot{P}|\!)^\natural) \mathrel{\dot{\supseteq}} \dot{\alpha}(\alpha_{\mathrm{I}_P \rightsquigarrow \mathrm{U}_P}(\mathcal{I}))$

Proof. The proof follows from the definition of abstract data leakage semantics $(\!|\dot{P}|\!)^\natural$ and that of the concretization function γ, observing that all abstract semantic functions $a[\![S]\!]$ for a statement S in P always over-approximate the set of input data sources from which a data frame variable depends on. □

Similarly, we have the sound but not complete counterpart of Lemma 1 for practically checking absence of data leakage (where, similarly, $\gamma((\!|\dot{P}|\!)^\natural)o$ stands for $\bigcup_{r\in\mathbb{N}} \gamma((\!|\dot{P}|\!)^\natural)o(r)$):

Lemma 2. $P \models \mathcal{I} \Leftarrow \forall o_1 \in \mathrm{U}^{\text{train}}_P, o_2 \in \mathrm{U}^{\text{test}}_P \colon \gamma((\!|\dot{P}|\!)^\natural)o_1 \cap \gamma((\!|\dot{P}|\!)^\natural)o_2 = \emptyset$

In particular, the antecedent in Lemma 2 is equivalent to

$$\forall o_1 \in \mathrm{U}^{\text{train}}_P, o_2 \in \mathrm{U}^{\text{test}}_P \colon \forall X^C_R \in (\!|\dot{P}|\!)^\natural_s o_1, Y^{C'}_{R'} \in (\!|\dot{P}|\!)^\natural_s o_2 \colon \neg\text{overlap}(X^C_R, Y^{C'}_{R'}) \wedge \left(X = Y \Rightarrow \neg(\!|\dot{P}|\!)^\natural_b o_1 \wedge \neg(\!|\dot{P}|\!)^\natural_b o_2\right)$$

allowing us to verify absence of data leakage by checking that any pair of abstract data frames sources X^C_R and $Y^{C'}_{R'}$ for data respectively used for training (i.e., o_1) and testing (i.e., o_2) are disjoint (i.e, $\neg\text{overlap}(X^C_R, Y^{C'}_{R'})$, cf. Equation 14) and untainted (i.e., $\neg(\!|\dot{P}|\!)^\natural_b o_1 \wedge \neg(\!|\dot{P}|\!)^\natural_b o_2$ if $X = Y$, that is, if they originate from the same data file).

Example 15 (Feature Normalization Leakage (Continued))**.** *The data leakage analysis of the notebook execution in Example 8 is the following:*

$$a[\![\texttt{data = read("credit.csv")}]\!]\dot{\perp}_A = \left(m_1 \stackrel{def}{=} \lambda x\colon \begin{cases} \langle \{data^{\top_C}_{[0,\infty]}\}, \textsc{false}\rangle & x = data \\ undefined & otherwise \end{cases}\right)$$

$$a[\![\texttt{X = data.select[][[}\texttt{"Age", "Amount"}\texttt{]]}]\!]m_1 = \left(m_2 \stackrel{def}{=} m_1\left[X \mapsto \langle \{data^{\{\text{“}Age\text{”},\text{“}Amount\text{”}\}}_{[0,\infty]}\}, \textsc{false}\rangle\right]\right)$$

$$a[\![\texttt{y = data.select[][[}\texttt{"Approved"}\texttt{]]}]\!]m_2 = \left(m_3 \stackrel{def}{=} m_2\left[y \mapsto \langle \{data^{\{\text{“}Approved\text{”}\}}_{[0,\infty]}\}, \textsc{false}\rangle\right]\right)$$

$$a[\![\texttt{X = normalize(X)}]\!]m_3 = \left(m_4 \stackrel{def}{=} m_3\left[X \mapsto \langle \{data^{\{\text{“}Age\text{”},\text{“}Amount\text{”}\}}_{[0,\infty]}\}, \textsc{true}\rangle\right]\right)$$

$$a[\![\texttt{X\_train = X.select[[}0.025 * R_X\texttt{] + 1, ..., }R_X\texttt{]][]}]\!]m_4 = \left(m_5 \stackrel{def}{=} m_4\left[X\_\text{train} \mapsto \langle \{data^{\{\text{“}Age\text{”},\text{“}Amount\text{”}\}}_{[\lfloor 0.025 * R_X\rfloor + 1, R_X]}\}, \textsc{true}\rangle\right]\right)$$

$a[\![\texttt{y\_train = y.select[[}\lfloor 0.025 * R_y \rfloor \texttt{ + 1, ..., } R_y\texttt{]]}\,\texttt{[]}]\!]m_5 = \left(m_6 \stackrel{def}{=} m_5\left[y\_\text{train} \mapsto \left\langle \left\{ data^{\{\text{“Approved”}\}}_{[\lfloor 0.025 * R_y \rfloor + 1, R_y]} \right\}, \text{FALSE} \right\rangle\right]\right)$

$a[\![\texttt{X\_test = X.select[[0, ..., }\lfloor 0.025 * R_X \rfloor\texttt{]]}\,\texttt{[]}]\!]m_6 = \left(m_7 \stackrel{def}{=} m_6\left[X\_\text{test} \mapsto \left\langle \left\{ data^{\{\text{“Age”},\text{“Amount”}\}}_{[0, \lfloor 0.025 * R_X \rfloor]} \right\}, \text{TRUE} \right\rangle\right]\right)$

$a[\![\texttt{y\_test = y.select[[0, ..., }\lfloor 0.025 * R_y \rfloor\texttt{]]}\,\texttt{[]}]\!]m_7 = \left(m_8 \stackrel{def}{=} m_7\left[y\_\text{test} \mapsto \left\langle \left\{ data^{\{\text{“Approved”}\}}_{[0, \lfloor 0.025 * R_y \rfloor]} \right\}, \text{FALSE} \right\rangle\right]\right)$

$a[\![\texttt{train(\{X\_train, X\_test\})}]\!]m_8 = m_8$

$a[\![\texttt{test(\{X\_test, y\_test \})}]\!]m_8 = m_8$

At the end of the analysis, X_train $\in \mathrm{U}^{\text{train}}$ and X_test $\in \mathrm{U}^{\text{test}}$ depend on disjoint but tainted *abstract data frames derived from the same input file credit.csv. Thus, the absence of data leakage check from Lemma 2 (rightfully) fails.*

Example 16 (Feature Engineering Leakage (Continued))**.** *The data leakage analysis for Example 9 is the following:*

$$a[\![\texttt{data = read("credit.csv")}]\!]\dot{\perp}_A = \begin{pmatrix} m_1 \stackrel{def}{=} \lambda x\colon \begin{cases} \left\langle \left\{ data^{\top_C}_{[0,\infty]} \right\}, \text{FALSE} \right\rangle & x = data \\ undefined & otherwise \end{cases} \end{pmatrix}$$

$a[\![\texttt{X1 = data.select[]}\,\texttt{[\{"Age"\}]}]\!]m_1 = \left(m_2 \stackrel{def}{=} m_1\left[X1 \mapsto \left\langle \left\{ data^{\{\text{“Age”}\}}_{[0,\infty]} \right\}, \text{FALSE} \right\rangle\right]\right)$

$a[\![\texttt{v = data.select[]}\,\texttt{[\{"Amount"\}]}]\!]m_2 = \left(m_3 \stackrel{def}{=} m_2\left[v \mapsto \left\langle \left\{ data^{\{\text{“Amount”}\}}_{[0,\infty]} \right\}, \text{FALSE} \right\rangle\right]\right)$

$a[\![\texttt{m = aggregate(data)}]\!]m_3 = \left(m_4 \stackrel{def}{=} m_3\left[m \mapsto \left\langle \left\{ data^{\top_C}_{[0,\infty]} \right\}, \text{TRUE} \right\rangle\right]\right)$

$a[\![\texttt{a = m.select[]}\,\texttt{[\{"Amount"\}]}]\!]m_4 = \left(m_5 \stackrel{def}{=} m_4\left[a \mapsto \left\langle \left\{ data^{\{\text{“Amount”}\}}_{[0,\infty]} \right\}, \text{TRUE} \right\rangle\right]\right)$

$a[\![\texttt{X2 = v > a}]\!]m_5 = \left(m_6 \stackrel{def}{=} m_5\left[X2 \mapsto \left\langle \left\{ data^{\{\text{“Amount”}\}}_{[0,\infty]} \right\}, \text{TRUE} \right\rangle\right]\right)$

$a[\![\texttt{y = data.select[]}\,\texttt{[\{"Approved"\}]}]\!]m_6 = \left(m_7 \stackrel{def}{=} m_6\left[y \mapsto \left\langle \left\{ data^{\{\text{“Approved”}\}}_{[0,\infty]} \right\}, \text{FALSE} \right\rangle\right]\right)$

$a[\![\texttt{X1\_train = X1.select[[}\lfloor 0.025 * R_{X1} \rfloor\texttt{ + 1, ..., } R_{X1}\texttt{]]}\,\texttt{[]}]\!]m_7 = \left(m_8 \stackrel{def}{=} m_7\left[X1\_\text{train} \mapsto \left\langle \left\{ data^{\{\text{“Age”}\}}_{[\lfloor 0.025 * R_{X1} \rfloor + 1, R_{X1}]} \right\}, \text{FALSE} \right\rangle\right]\right)$

$a[\![\texttt{X2\_train = X2.select[[}\lfloor 0.025 * R_{X2} \rfloor\texttt{ + 1, ..., } R_{X2}\texttt{]]}\,\texttt{[]}]\!]m_8 = \left(m_9 \stackrel{def}{=} m_8\left[X2\_\text{train} \mapsto \left\langle \left\{ data^{\{\text{“Amount”}\}}_{[\lfloor 0.025 * R_{X2} \rfloor + 1, R_{X2}]} \right\}, \text{TRUE} \right\rangle\right]\right)$

$a[\![\texttt{y\_train = y.select[[}\lfloor 0.025 * R_y \rfloor\texttt{ + 1, ..., } R_y\texttt{]]}\,\texttt{[]}]\!]m_9 = \left(m_{10} \stackrel{def}{=} m_9\left[y\_\text{train} \mapsto \left\langle \left\{ data^{\{\text{“Approved”}\}}_{[\lfloor 0.025 * R_y \rfloor + 1, R_y]} \right\}, \text{FALSE} \right\rangle\right]\right)$

$a[\![\texttt{X1\_test = X1.select[[0, ..., }\lfloor 0.025 * R_{X1} \rfloor\texttt{]]}\,\texttt{[]}]\!]m_{10} = \left(m_{11} \stackrel{def}{=} m_{10}\left[X1\_\text{test} \mapsto \left\langle \left\{ data^{\{\text{“Age”}\}}_{[0, \lfloor 0.025 * R_{X1} \rfloor]} \right\}, \text{FALSE} \right\rangle\right]\right)$

$a[\![\texttt{X2\_test = X2.select[[0, ..., }\lfloor 0.025 * R_{X2} \rfloor\texttt{]]}\,\texttt{[]}]\!]m_{11} = \left(m_{12} \stackrel{def}{=} m_{11}\left[X2\_\text{test} \mapsto \left\langle \left\{ data^{\{\text{“Amount”}\}}_{[0, \lfloor 0.025 * R_{X2} \rfloor]} \right\}, \text{TRUE} \right\rangle\right]\right)$

$a[\![\texttt{y\_test = y.select[[0, ..., }\lfloor 0.025 * R_y \rfloor\texttt{]]}\,\texttt{[]}]\!]m_{12} = \left(m_{13} \stackrel{def}{=} m_{12}\left[y\_\text{test} \mapsto \left\langle \left\{ data^{\{\text{“Approved”}\}}_{[0, \lfloor 0.025 * R_y \rfloor]} \right\}, \text{FALSE} \right\rangle\right]\right)$

$a[\![\texttt{train(\{X\_train, X\_test\})}]\!]m_{13} = m_{13}$

$a[\![\texttt{test(\{X\_test, y\_test \})}]\!]m_{13} = m_{13}$

Here too the absence of data leakage check from Lemma 2 rightfully fails because X2_train $\in \mathrm{U}^{\text{train}}$ and X2_test $\in \mathrm{U}^{\text{test}}$ depend on tainted abstract data frames derived from the same input file.

Remark. If we allow programs to read the same input file multiple times, we need once again (cf. end of Section 3) to additionally maintain a map $F_P \in \mathrm{I}_P \to \mathbb{W}$ from input data frame variables and the corresponding data file names, and use it to perform a replacement on the data sources of the abstract data leakage semantics of the program, i.e., yielding $(\!|\dot{P}|\!)^\natural \in \mathbb{X} \to (\mathbb{W} \times C \times \mathcal{R}) \times \mathbb{B}$, before applying Lemma 2.

5. Implementation

We implemented a data leakage static analysis based on our approach described in Section 4 into NBLyzer (Subotić et al., 2022), an open-source (available on GitHub: `https://github.com/microsoft/NBLyzer`) static analysis framework for Python data science notebooks. We compare to the pre-existing preliminary data leakage analysis already implemented in NBLyzer in our experimental evaluation in the next section.

NBLyzer is tailored to the unique interactive environment of data science notebooks, where cells can be executed in non-sequential order. The ideal use for NBLyzer is its integration in an Integrated Development Environment

(IDE). Static analyses in NBLyzer can then be triggered by *events* (e.g., the highlighting or execution of a specific code cell) and essentially answer *what-if* questions regarding the execution of a code cell, i.e., "what happens if this cell is executed?". A fixpoint computation engine systematically executes the abstract semantics to propagate an abstract analysis state across individual cells (*intra-cell analysis*) and to valid successor cells (*inter-cell analysis*), pruning away unfeasible sequences of cell executions on-the-fly. A user-defined parameter $K \in \{1, \ldots, \infty\}$ bounds the depth of the analysis, i.e., the length of the execution sequences.

Cell Propagation Condition. In order to determine if the abstract analysis state propagation should continue to another cell, NBLyzer relies on an analysis-specific ϕ-condition. To achieve good performance, ϕ must be defined to be as strong (i.e., restrictive) as possible while not sacrificing soundness, i.e., to avoid missing any interesting execution sequences (e.g., containing a bug) by terminating the analysis prematurely.

For our data leakage static analysis, the ϕ condition takes as input the abstract analysis state m resulting from the analysis on an individual code cell, and the pre-summary pre_c of a candidate successor code cell c, i.e., the set of unbound data frame variables in c. It is defined as follows:

$$\phi(m, pre_c) \stackrel{\text{def}}{=} pre_c \neq \emptyset \wedge pre_c \subseteq \left\{ v \in \text{dom}(m) \;\middle|\; X_R^C \in m(v), R \neq \bot \right\} \tag{16}$$

The cell c should be analyzed if its unbound data frame variables in pre_c map to an abstract data frame set with at least one abstract data frame X_R^C that is non-empty ($R \neq \bot$) in the current abstract analysis state m.

Knowledge Base. In order to determine which data science library functions correspond to the classes of program statements that we employ in our static analysis (cf. Section 4.2), we assume the existence of a knowledge base KB that classifies functions accordingly.

Practically, in our evaluation we used the same simple knowledge base as that used by NBLyzer (Subotić et al., 2022), which classifies the most commonly used function in the Pandas and Sklearn data science libraries, i.e., `read_csv`, `fillna`, `fit`, `predict`, etc. In particular, this knowledge base classifies as *tainting* all functions used for data aggregation (i.e., `sum`, `mean`, `max`, `groupby`, etc.), data normalization and scaling (i.e., `fit_transform`, etc.), and data imputation (i.e., `fillna`, `interpolate`, etc.).

Interprocedural Analysis. We support interprocedural analysis via function inlining. We inline the body of functions defined in executed cells at any subsequent call site. We treat functions as undefined (thus, coarsely over-approximating their behavior) when their definition does not exist in a predecessor code cell.

Soundness. While the formalization of our data leakage analysis in Section 4 is sound with respect to the concrete semantics on our small data-frame manipulating language (cf. Section 3), we report certain sources of unsoundness in our actual analysis implementation. In particular, we only support a subset of Python 3, focusing on the code construct most commonly observed in data science notebooks. Specifically, we do not support code constructs such as dynamic code evaluation or reference aliasing. We remark, however, that data science Python code is relatively simple compared to general Python code. Among the data science notebooks used for our experimental evaluation (cf. Section 6) only 0.5% of them contained instances of assignments of data frame variables by reference, requiring an alias analysis, and none of them contained dynamic code evaluation instances.

6. Experimental Evaluation

We evaluated the precision and performance of our implementation as a data leakage detector in an IDE. In such a use case, in accordance to the RAIL performance model, the analyzer should identify data leakages with a *soft* analysis deadline of 1 second (rai, 2022). We additionally compared against the pre-existing data leakage analysis already implemented in NBLyzer.

6.1. Experimental Setup

6.1.1. Environment

All experiments were performed on a Ryzen 9 6900HS with 24GB DDR5 running Ubuntu 22.04. Python 3.10.6 was used to execute two versions of NBLyzer: one – the default one – running the pre-existing data leakage analysis (Subotić et al., 2022), and the other – the new and extended one – running our data leakage static analysis. We used the default NBLyzer configuration settings (e.g., $K = 5$).

Table 1: Characteristics of the data science notebooks in the Kaggle benchmark suite used for our evaluation.

Characteristic		Mean	Std Dev	Max	Min
Notebook-Wise	Cells	23.58	20.21	182	1
	Functions	3.33	7.11	72	0
	Classes	0.14	0.64	11	0
	Parse Error	0.5	0.98	20	0
Cell-Wise	Code Lines	9.12	13.55	257	1
	Branching Instructions	0.43	2.49	76	0
	Variables	8.2	2.3	552	0
	Unbound Variables	2.1	1.06	12	0

Table 2: Alarms raised by the pre-existing data leakage analysis in NBLyzer and our data leakage analysis.

Analysis	True Positives		False Positives	
	Taint	Overlap	Taint	Overlap
Subotić et al. (2022)	10	0	2	0
Ours	**10**	**15**	**2**	**0**

6.1.2. Benchmarks

For the evaluation, we use a data science notebook benchmark suite consisting of 4 Kaggle (kag, 2022) competitions that has previously been used to evaluate data science static analyzers (Namaki et al., 2020; Subotić et al., 2022). We analyzed 2111 over 2413 notebooks. We excluded 302 notebooks because they could not be digested by our analyzer (i.e., syntax errors, JSON decoding errors, etc.). All notebooks are written to succeed in a non-trivial data science competition task and can be assumed to closely represent code of non-novice data scientists.

The benchmark characteristics are summarized in Table 1. On average the notebooks in the benchmark suite have 24 cells, where each cell on average has 9 lines of code. On average, branching instructions appear in 33% of cells. Each notebook defined on average 3 functions and 0.1 classes.

6.1.3. Methodology

We ran NBLyzer's and our data leakage analysis on all *valid executions* of each data science notebook in our benchmark suite, i.e., any non-empty sequence of code cells starting with a code cell without unbound variables (i.e, an empty pre-summary, cf. Section 5). The code cells in these sequences can be in any order, they do not need to follow the sequential order of definition of in the notebook. From the 2111 notebooks in our suite, we analyzed a total of 7378 notebook executions.

6.2. Precision Results

To evaluate the precision of our data leakage static analysis we compared it on our benchmark suite to the one previously implemented in NBLyzer. We summarize the results in Table 2. For each reported alarm, we engaged 4 data scientists at Microsoft to determine true and false positives. We further classified them as due to tainted (Taint) or overlapping data frames (Overlap).

Our analysis found 25 true positives over the 2111 notebooks in our benchmark: 10 executions exhibiting Taint data leakage in 5 notebooks, and 15 executions with Overlap data leakage in 11 notebooks, i.e., a 1.2% bug rate, which adheres to true positive bug rates reported for null pointers in industrial settings (e.g., see (Kharkar et al., 2022)). The previous analysis only found the 10 executions with Taint leakage in 5 notebooks. Many of these notebooks resemble Example 1 and Example 2 discussed in Section 1. The previous analysis could not detect Overlap data leakages – the majority of those found in out suite – because it cannot reason at the granularity of partial data frames, unlike our analysis which can reason at the granularity of data frame rows and columns (cf. Section 4).

In Figure 8 we show an example of a notebook in our suite which exhibits Overlap data leakage. In Cell 5, data is read and stored into the data frame variable `df`. After several exploratory data analysis steps (not shown), in Cell 8, the model target (data frame column containing the labels to be predicted by the model) and the model features

...

In [5]:
```
df = pd.read_csv("heart.csv")
```

...

In [8]:
```
y = df[['target']]
X = df.drop('target', axis=1)

X_train = X.iloc[:split+1]
X_test = X.iloc[split:end]

y_train = y.iloc[:split+1]
y_test = y.iloc[split:end]
```

In [9]:
```
lr_clf = LogisticRegression(solver='liblinear')
train1 = lr_clf.fit(X_train, y_train)
```

In [10]:
```
train_score = accuracy_score(y_test, lr_clf.predict(X_test))
```

Figure 8: Example of data science notebook exhibiting Overlap data leakage.

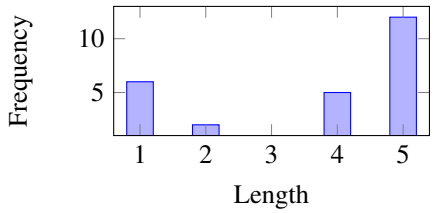

Figure 9: Frequency of the length (number of code cells) of the error executions produced by our analysis.

(the other data frame columns, which are the inputs to the model) are split into two data frame variables `y` and `X` respectively. These are split into four (`X_train`, `X_test`, `y_train`, `y_test`) forming the training and test data sets. The location of the split is defined by the `split` variable. The programmer appears unsure about the semantics of `iloc` and assumes that the end point needs to be incremented to obtain the right split index. As a consequence the data is split such that both training and test data contain the row indexed at the value of `split`. Thus, in Cell 9 and Cell 10, a data leakage occurs because a linear regression model is trained and tested with overlapping data. Other examples of Overlap data leakage in our benchmark suite include notebook executions where data frame variables are used both as input and output of a data split function (e.g., `X, X_test, y, y_test = train_test_split(X, y)`) and the code cell containing the split is (accidentally) skipped in the execution.

In theory, the capability of our analysis to reason at the granularity of rows and columns makes it more precise than the previous also for Taint data leakages (i.e., less prone to false positives), but this improvement did not manifest on our benchmark suite. Both ours and the previous analysis reported 2 false positives in our evaluation: one execution (falsely) exhibiting Taint data leakage in 2 notebooks. The reason for these is the fact that both analyses do not keep track of object types and thus cannot distinguish tainting and non-tainting functions with the same name called on different objects. For instance, both `LabelEncoder` and `StandardScaler` are equipped with a `fit_transform` function, but calling `x.fit_transform(df)` on an object `x` taints the data frame variable `df` if `x` is of type `StandardScalar`, but not if `x` is of type `LabelEncoder`. However, without knowing the type of `x`, both analyses need to over-approximate and always assume that the data frame variable `df` is tainted, independently of the actual type of the object `x`. We leave complementing our analysis with object-sensitive type tracking for future work.

Actionability of Data Leakage Bug Reports. The error executions produced by our analysis varied between 1 and 5 cells in length, indicating that the majority of data leakage bugs (76%) manifested over several notebook code cells. On the other hand, the error traces were localized enough to put little burden on the user while performing a triage of static analysis findings. We summarize these findings in the histogram in Figure 9.

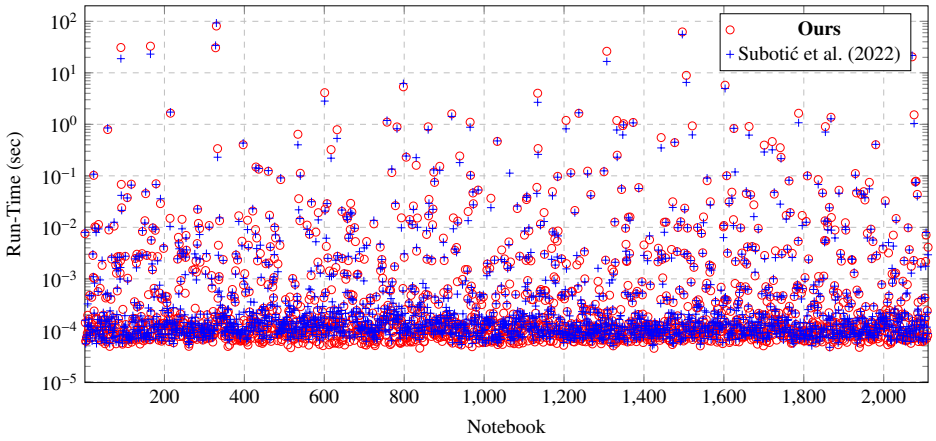

Figure 10: Average analysis run-times per notebook of the pre-existing data leakage analysis in NBLyzer and our data leakage analysis.

6.3. Performance Results

On our benchmark suite, the majority (over 99%) of analyses completed in less than 1 second per notebook, meeting our objective with respect to the RAIL performance model. Among the ~ 1% of cases that do not satisfy the performance model, we find high levels of code cell interconnectedness (51% on average) combined with a large numbers of cells (45 on average). For instance, in our benchmark suite, one notebook has 50 code cells where X_{train} and y_{train} are referenced in 27 of them, yielding a large set of possible execution paths NBLyzer needs to analyze and thus resulting in degraded performance.

We also compare the execution run-time of our analysis and the one previously implemented in NBLyzer in Figure 10, which shows the average analysis run-time across different executions of each data science notebook in our benchmark suite. For the vast majority, the difference in runtime between our and the previous data leakage analysis is negligible. However, for several cases we experience a noticeable slowdown due to our more complex abstract domains and semantics. On a small number of benchmarks our analysis is faster, thanks to its improved precision that allows it to find a bug and terminate while the previous analysis continues executing. Overall, our analysis experiences a slowdown of 7%. Considering the improved bug detection capabilities, we believe that this is a small price to pay and does not appear to significantly degrade the IDE user experience.

7. Related Work

Related Abstract Interpretation Frameworks and Static Analyses. As mentioned in Section 3, our framework generalizes the notion of data usage proposed by Urban and Müller (2018) and the definition of dependency abstraction used by Cousot (2019). In particular, among other things, the generalization involves reasoning about dependencies (and thus data usage relationships) between multi-dimensional variables. In (Urban and Müller, 2018), Urban and Müller show that information flow analyses can be used for reasoning about data usage, albeit with a loss in precision unless one repeats the information flow analysis by setting each time a different input variable as high security variable (cf. Section 8 in (Urban and Müller, 2018)). In virtue of the generalization mentioned above, the same consideration applies to our work, i.e., information flow analyses can be used to reason about data leakage, but with an even higher cost to avoid a precision loss (the analysis needs to be repeated each time for different portions of the input data sources). Analogously, in (Cousot, 2019), Cousot shows that information flow, slicing, non-interference, dye, and taint analyses are all further abstractions of his proposed framework (cf. Section 7 in (Cousot, 2019)). As such, they are also further (and thus less precise) abstractions of our proposed framework. In particular, a taint analysis will only be able to detect a subset of the data leakage bugs that our analysis can find, i.e., those solely originating from library transformations but not those originating from (partially) overlapping data. Vice versa, our proposed analysis could also be used as a more fine-grained information flow or taint analysis. Mazzucato et al. (2024) recently proposed a quantitative generalization of data usage (Urban and Müller, 2018) yielding a static analysis for determining the

impact of input data on the program computations, parametrized in the definition of impact. It could be interesting to generalize this analysis in our context as it could allow detecting target leakage by identifying input data having more impact on the prediction of a trained machine learning model.

Static Analysis for Data Science. Static analysis for data science is an emerging area in the program analysis community (Urban, 2019). Some notable static analyses for data science scripts include an analysis for ensuring correct shape dimensions in TensorFlow programs (Lagouvardos et al., 2020), an analysis for constraining inputs based on program constraints (Urban and Müller, 2018), and a provenance analysis (Namaki et al., 2020). More recently, Dolcetti et al. (2024) proposed a linter for data science code, which assigns abstract datatypes to program variables and checks them for consistency when calling data science libraries functions.

Wang et al. (2020) highlight the prevalence of poor-quality code in data science (Jupyter) notebooks, which often fail to adhere to the recommended Python coding practices, e.g., containing unused variables, deprecated functions, and style violations according to PEP8 standards. The authors argue that these shortcomings pose risks, particularly given the educational nature of many Jupyter notebooks, as poor practice might propagate to learners and future developers. They call for systematic analysis and tools to improve coding quality in notebooks to ensure their effectiveness as a medium for scientific and educational communication. Since then, a few static analyses have been proposed for data science notebooks (Macke et al., 2020; Subotić et al., 2022). NBLyzer (Subotić et al., 2022) focuses on data science notebooks used for machine learning and contains an ad-hoc data leakage analysis that detects Taint data leakages (Subotić et al., 2022). An extension to handle Overlap data leakages was sketched by Subotic et al. (2022) but no analysis precision results were reported. However, both these analysis are not formalized nor formally proven sound[1]. In contrast, we introduce a sound data leakage analysis that has a rigorous semantic underpinning. Recently, Liu et al. (2023) proposed a data leakage analysis based on (Subotic et al., 2022) on top of which they additionally proposed an approach to automatically repair the code. Our analysis could be integrated into their framework.

Data Leakage Detection and Avoidance. Several techniques are available to prevent and identify data leakage in data science workflows. Traditional approaches often rely on manual inspection (Kaufman et al., 2011), where data is reviewed when a leakage is suspected. While effective in some cases, this process is labor-intensive and prone to oversight. Another common strategy to mitigate leakage is the use of structured data science pipelines (Biswas et al., 2022). These pipelines organize the phases of sourcing, cleaning, splitting, normalization, and training in a way that avoids critical errors, such as performing normalization before data splitting. However, implementing such pipelines requires significant manual effort and modifications to existing code, making them less prevalent among the millions of data scientists, particularly in notebook-based environments. Data provenance and lineage tools (Namaki et al., 2020) offer another approach by building dependency graphs that trace data transformations and help uncover leakage points. While useful, these tools are limited in their ability to detect subtle or latent instances of data leakage and are primarily effective for retrospective analysis after the issue has manifested. Dynamic analysis tools, such as (pyd, 2022; Chorev et al., 2022), instrument the code during execution to detect leakage in real-time. These tools, while promising, often introduce runtime overhead and require additional boilerplate code. Moreover, they still demand some degree of manual inspection to validate and interpret the results. Nonetheless, dynamic techniques hold potential for reducing false positives when integrated effectively.

8. Conclusion

We have presented an approach based on abstract interpretation for detecting data leakages caused by train-test contamination in machine learning statically, at development time. We have provided a formal and rigorous derivation from the standard program collecting semantics, via successive abstractions to a final sound and computable static analysis definition. We have implemented our approach in the NBLyzer static analysis framework and evaluated it on a large corpus of data science notebooks obtained from the Kaggle competition platform. Results demonstrated that our technique effectively detects Taint and Overlap data leakages with a high degree of precision, outperforming prior approaches, at the cost of a small overall performance slowdown.

[1] We found a number of soundness issues in Subotic et al. (2022) when working on our formalization.

There are several promising directions for future work. First of all, we could improve the precision of our analysis by additionally keeping track of data frame contents via a value analysis. For this we could leverage simple abstract type information or leverage more complex (non-relational or relational) numerical and string abstract domains. We can further complement our analysis with an object-sensitive type analysis and an alias analysis. A more interesting direction is to tackle other forms of data leakage such as target and group leakage, with a quantitative generalization of our static analysis and the synergistic use of data correlation analyses. Finally, to enhance the experience of the target users of our static analysis, i.e., data scientists, it would be good to add features for interactive debugging, such as providing actionable insights or recommendations for resolving detected issues.

References

, 2018. Index, in: Nisbet, R., Miner, G., Yale, K. (Eds.), Handbook of Statistical Analysis and Data Mining Applications (Second Edition). second edition ed.. Academic Press, Boston, pp. 783–792. URL: https://www.sciencedirect.com/science/article/pii/B9780124166325099898, doi:https://doi.org/10.1016/B978-0-12-416632-5.09989-8.

, 2020. We downloaded 10m jupyter notebooks from github this is what we learned. https://blog.jetbrains.com/datalore/2020/12/17/we-downloaded-10-000-000-jupyter-notebooks-from-github-this-is-what-we-learned/. Accessed: 22-01-22.

, 2022. Kaggle. http://kaggle.com. Accessed: 2022-09-30.

, 2022. leak-detect. https://github.com/abhayspawar/leak-detect. Accessed: 2022-03-08.

, 2022. RAIL Model. https://web.dev/rail/. Accessed: 2022-09-30.

Biswas, S., Wardat, M., Rajan, H., 2022. The art and practice of data science pipelines: A comprehensive study of data science pipelines in theory, in-the-small, and in-the-large, in: ICSE'22: The 44th International Conference on Software Engineering.

Chorev, S., Tannor, P., Israel, D.B., Bressler, N., Gabbay, I., Hutnik, N., Liberman, J., Perlmutter, M., Romanyshyn, Y., Rokach, L., 2022. Deepchecks: A library for testing and validating machine learning models and data. URL: https://arxiv.org/abs/2203.08491, doi:10.48550/ARXIV.2203.08491.

Chouldechova, A., Prado, D.B., Fialko, O., Vaithianathan, R., 2018. A case study of algorithm-assisted decision making in child maltreatment hotline screening decisions, in: Friedler, S.A., Wilson, C. (Eds.), Conference on Fairness, Accountability and Transparency, FAT 2018, 23-24 February 2018, New York, NY, USA, PMLR. pp. 134–148.

Cousot, P., 2002. Constructive Design of a Hierarchy of Semantics of a Transition System by Abstract Interpretation. Electronic Notes in Theoretical Computer Science 277, 47–103.

Cousot, P., 2019. Abstract Semantic Dependency, in: Proc. SAS, pp. 389–410.

Cousot, P., Cousot, R., 1976. Static Determination of Dynamic Properties of Programs, in: Proceedings of the Second International Symposium on Programming, pp. 106–130.

Cousot, P., Cousot, R., 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints, in: Proc. POPL, pp. 238–252.

Cousot, P., Cousot, R., 1979. Systematic design of program analysis frameworks, in: Proc. POPL, pp. 269–282.

Cousot, P., Cousot, R., 1992. Abstract interpretation and application to logic programs. J. Log. Program. 13, 103–179.

Cousot, P., Cousot, R., 1994. Higher Order Abstract Interpretation (and Application to Comportment Analysis Generalizing Strictness, Termination, Projection, and PER Analysis, in: ICCL, pp. 95–112.

Dolcetti, G., Cortesi, A., Urban, C., Zaffanella, E., 2024. Towards a high level linter for data science, in: Proceedings of the 10th ACM SIGPLAN International Workshop on Numerical and Symbolic Abstract Domains, Association for Computing Machinery, New York, NY, USA. p. 18–25. URL: https://doi.org/10.1145/3689609.3689996, doi:10.1145/3689609.3689996.

Drobnjakovic, F., Subotic, P., Urban, C., 2024. An abstract interpretation-based data leakage static analysis, in: Chin, W., Xu, Z. (Eds.), Theoretical Aspects of Software Engineering - 18th International Symposium, TASE 2024, Guiyang, China, July 29 - August 1, 2024, Proceedings, Springer. pp. 109–126. URL: https://doi.org/10.1007/978-3-031-64626-3_7, doi:10.1007/978-3-031-64626-3_7.

Gange, G., Navas, J.A., Schachte, P., Søndergaard, H., Stuckey, P.J., 2021. Disjunctive interval analysis, in: SAS, pp. 144–165.

Kapoor, S., Narayanan, A., 2023. Leakage and the reproducibility crisis in machine-learning-based science. Patterns 4, 100804. URL: https://doi.org/10.1016/j.patter.2023.100804, doi:10.1016/J.PATTER.2023.100804.

Kaufman, S., Rosset, S., Perlich, C., 2011. Leakage in data mining: Formulation, detection, and avoidance, Association for Computing Machinery, New York, NY, USA. p. 556–563. URL: https://doi.org/10.1145/2020408.2020496, doi:10.1145/2020408.2020496.

Kaufman, S., Rosset, S., Perlich, C., Stitelman, O., 2012. Leakage in data mining: Formulation, detection, and avoidance. ACM Trans. Knowl. Discov. Data 6. URL: https://doi.org/10.1145/2382577.2382579, doi:10.1145/2382577.2382579.

Kharkar, A., Moghaddam, R.Z., Jin, M., Liu, X., Shi, X., Clement, C., Sundaresan, N., 2022. Learning to reduce false positives in analytic bug detectors, in: Proceedings of the 44th International Conference on Software Engineering, Association for Computing Machinery, New York, NY, USA. p. 1307–1316. URL: https://doi.org/10.1145/3510003.3510153, doi:10.1145/3510003.3510153.

Lagouvardos, S., Dolby, J., Grech, N., Antoniadis, A., Smaragdakis, Y., 2020. Static analysis of shape in tensorflow programs, in: Hirschfeld, R., Pape, T. (Eds.), 34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference), Schloss Dagstuhl - Leibniz-Zentrum für Informatik. pp. 15:1–15:29.

Liu, Y., Mechtaev, S., Subotic, P., Roychoudhury, A., 2023. Program repair guided by datalog-defined static analysis, in: Chandra, S., Blincoe, K., Tonella, P. (Eds.), Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023, ACM. pp. 1216–1228. URL: https://doi.org/10.1145/3611643.3616363, doi:10.1145/3611643.3616363.

Macke, S., Gong, H., Lee, D.J.L., Head, A., Xin, D., Parameswaran, A.G., 2020. Fine-grained lineage for safer notebook interactions. CoRR abs/2012.06981. URL: https://arxiv.org/abs/2012.06981, arXiv:2012.06981.

Mazzucato, D., Campion, M., Urban, C., 2024. Quantitative input usage static analysis, in: Benz, N., Gopinath, D., Shi, N. (Eds.), NASA Formal Methods - 16th International Symposium, NFM 2024, Moffett Field, CA, USA, June 4-6, 2024, Proceedings, Springer. pp. 79–98. URL: https://doi.org/10.1007/978-3-031-60698-4_5, doi:10.1007/978-3-031-60698-4_5.

Miné, A., 2004. Weakly Relational Numerical Abstract Domains. Ph.D. thesis. École Polytechnique, Palaiseau, France. URL: https://tel.archives-ouvertes.fr/tel-00136630.

Namaki, M.H., Floratou, A., Psallidas, F., Krishnan, S., Agrawal, A., Wu, Y., Zhu, Y., Weimer, M., 2020. Vamsa: Automated provenance tracking in data science scripts, in: Proc. KDD, pp. 1542–1551.

Papadimitriou, P., Garcia-Molina, H., 2009. A model for data leakage detection, in: Proc. ICDE, pp. 1307–1310.

Perkel, J., 2018. Why jupyter is data scientists' computational notebook of choice. Nature 563, 145–146. doi:10.1038/d41586-018-07196-1.

Subotic, P., Bojanic, U., Stojic, M., 2022. Statically detecting data leakages in data science code, in: Gonnord, L., Titolo, L. (Eds.), SOAP '22: 11th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis, San Diego, CA, USA, 14 June 2022, ACM. pp. 16–22. URL: https://doi.org/10.1145/3520313.3534657, doi:10.1145/3520313.3534657.

Subotić, P., Milikić, L., Stojić, M., 2022. A static analysis framework for data science notebooks, in: ICSE'22: The 44th International Conference on Software Engineering.

Urban, C., 2019. Static analysis of data science software, in: Chang, B.E. (Ed.), Static Analysis - 26th International Symposium, SAS 2019, Porto, Portugal, October 8-11, 2019, Proceedings, Springer. pp. 17–23. URL: https://doi.org/10.1007/978-3-030-32304-2_2, doi:10.1007/978-3-030-32304-2_2.

Urban, C., Müller, P., 2018. An Abstract Interpretation Framework for Input Data Usage, in: Proc. ESOP, pp. 683–710.

Wang, J., Li, L., Zeller, A., 2020. Better code, better sharing: on the need of analyzing jupyter notebooks, in: Rothermel, G., Bae, D. (Eds.), ICSE-NIER 2020: 42nd International Conference on Software Engineering, New Ideas and Emerging Results, Seoul, South Korea, 27 June - 19 July, 2020, ACM. pp. 53–56. URL: https://doi.org/10.1145/3377816.3381724, doi:10.1145/3377816.3381724.

Wong, A., Otles, E., Donnelly, J.P., Krumm, A., McCullough, J., DeTroyer-Cooley, O., Pestrue, J., Phillips, M., Konye, J., Penoza, C., Ghous, M.H., Singh, K., 2021. External validation of a widely implemented proprietary sepsis prediction model in hospitalized patients. JAMA internal medicine .