An Abstract Interpretation-Based Data Leakage Static Analysis

Filip Drobnjaković, Pavle Subotić, Caterina Urban

▶ **To cite this version:**

Filip Drobnjaković, Pavle Subotić, Caterina Urban. An Abstract Interpretation-Based Data Leakage Static Analysis. Microsoft Research; Inria Paris; École Normale Supérieure. 2022. hal-03926245v2

HAL Id: hal-03926245
https://inria.hal.science/hal-03926245v2

Submitted on 23 Apr 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

An Abstract Interpretation-Based Data Leakage Static Analysis

Filip Drobnjaković[1], Pavle Subotić[1], and Caterina Urban[2]

[1] Microsoft, Serbia
[2] 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 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. We implemented it in the NBLyzer framework and we demonstrate its performance and precision on 2111 Jupyter notebooks from the Kaggle competition platform.

1 Introduction

As artificial intelligence (AI) continues its unprecedented impact on society, ensuring machine learning (ML) models are accurate is crucial. To this end, ML models need to be correctly trained and tested. This iterative task is typically performed within data science notebook environments [18,8]. A notable bug that can be introduced during this process is known as a *data leakage* [17]. Data leakages have been identified as a pervasive problem by the data science community [9,10,16]. In a number of recent cases data leakages crippled the performance of real-world risk prediction systems with dangerous consequences in high-stakes applications such as child welfare [1] and healthcare [23].

Data leakages arise when dependent data is used to train and test a model. This can come in the form of overlapping data sets or, more insidiously, by library transformations that create indirect data dependencies.

Example 1 (Motivating Example). Consider the following excerpt of a data science notebook (based on *569.ipynb* from our benchmarks, and written in the small language that we introduce in Section 3.3):

```
1  data = read("data.csv")
2  X_norm = normalize(X)
3  X_train = X_norm.select[[⌊0.025 * R_X_norm⌋ + 1, ..., R_X_norm]][]
4  X_test = X_norm.select[[0, ..., ⌊0.025 * R_X_norm⌋]][]
5  train(X_train)
6  test(X_test)
```

Line 1 reads data from a CSV file and line 2 normalizes it. Line 3 and 4 split the data into training and testing segments (we write R_x for the number of data rows stored in x). Finally, line 5 trains a ML model, and line 6 tests its accuracy.

In this case, a data leakage is introduced because line 2 performs a normalization, *before* line 3 and 4 split into train and test data. This implicitly uses the mean over the entire dataset to perform the data transformation and, as a result, the train and test data are implicitly dependent on each other. In our experimental evaluation (cf. Section 5) 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 line 2 is *tainted*.

Mainstream methods rely on detecting data leakages retroactively [10,17]. Given a suspicious result, e.g., an overly accurate model, data analysis methods are used to identify data dependencies. However, a reasonable result may avoid suspicion from a data scientist until the model is already deployed. This is a natural use case for *static analysis* to detect data leakages at development time.

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` originate from previously normalized data (despite being disjoint).

Our static analysis (cf. Section 4) is designed within the abstract interpretation framework [5]: 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 be sufficiently precise, cf. Section 5). Moreover, it allows a clear comparisons with other related static analyses, e.g., information flow and taint analyses (cf. Section 6). Finally, this design principle allowed us to identify and overcome issues and shortcomings of previous data leakage analysis attempts [20,19].

We implemented our analysis in the NBLyzer [20] framework. 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.

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. The first row of non-empty data frames contains the labels of the data frame columns. Let $\mathbb{D} \stackrel{\text{def}}{=} \bigcup_{r \in \mathbb{N}} \bigcup_{c \in \mathbb{N}} \mathbb{V}^{r \times c}$ be the set of all possible data frame. Given a 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 *final states* of the program 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 [2]:

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

In the rest of the paper, we write $\llbracket P \rrbracket$ 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 [5,6]. This allows checking property satisfaction by set inclusion (see below) also across abstractions (cf. Theorems 1 and 2). 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* $\Lambda \in \mathcal{P}(\mathcal{P}(\Sigma^{+\infty}))$: $\Lambda \stackrel{\text{def}}{=} \{\Upsilon\}$, i.e., the property of "being the program with trace semantics Υ". Let $(\!|P|\!)$ denote the collecting semantics of a program P. Then, a program P satisfies a given property $\mathcal{H} \in \mathcal{P}(\mathcal{P}(\Sigma^{+\infty}))$ if and only if its collecting semantics is a subset of $\mathcal{H}$: $P \models \mathcal{H} \Leftrightarrow (\!|P|\!) \subseteq \mathcal{H}$. 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 2 ((In)dependent Data Frame Variables). Let us consider a program P with a single input data frame variable reading data frames with four rows and one single column with values in $\{3, 9\}$, i.e., data frames in $\bigcup_{r \in \{1,2,3,4\}} \{3, 9\}^r$ (cf. Section 2.1). Imagine that P 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 below shows all possible train and test data resulting from all possible input data of this program:

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'	

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 Equation 2) 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 Equation 2, with train data differing at line 2 and test data differing at both lines 1 and 2).

Instead, the table below shows all possible resulting train and test data if the normalization is performed *after* the split into train and test data:

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

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.

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 $\sigma \in [\![P]\!]$, 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 can now define when used data frame variables are independent in a program with trace semantics $[\![P]\!]$:

$$
\begin{aligned}
&\textsc{ind}([\![P]\!]) \stackrel{\text{def}}{=} \forall \sigma \in [\![P]\!], i \in \mathrm{I}_P, r \in R_i \colon \textsc{unch}(\sigma, i, r, \mathrm{U}_P^{\text{test}}) \vee \textsc{unch}(\sigma, i, r, \mathrm{U}_P^{\text{train}}) \\
&\textsc{unch}(\sigma, i, r, U) \stackrel{\text{def}}{=} \forall \bar{v} \in \mathbb{V}^{C_i} \colon \sigma(i)[r] \neq \bar{v} \Rightarrow (\exists \sigma' \in [\![P]\!] \colon \\
&\qquad \sigma'(i)[r] = \bar{v} \wedge \sigma(i) \stackrel{\bar{r}}{=} \sigma'(i) \wedge \sigma(\mathrm{I}_P \setminus \{i\}) = \sigma'(\mathrm{I}_P \setminus \{i\}) \wedge \sigma(U) = \sigma'(U))
\end{aligned} \tag{2}
$$

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, $\sigma(i) \stackrel{\bar{r}}{=} \sigma'(i)$ stands for $\forall r' \in R_i \colon r' \neq r \Rightarrow \sigma(i)[r'] = \sigma'(i)[r']$ (i.e., the data frame value of $i \in \mathrm{I}_P$ remains unchanged for any row $r' \neq r$), and $\sigma(X) = \sigma'(X)$ stands for $\forall x \in X \colon \sigma(x) = \sigma'(x)$. 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 $r \in R_i$) 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 $[\![P]\!]$. 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), instead of fixing the choice beforehand. Note also that UNCH 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 $r \in R_i$ 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. In terms of input data (non-)usage [22], this 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 [22] 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 can now be formally defined as the set $\mathcal{I} \stackrel{\text{def}}{=} \{[\![P]\!] \in \mathcal{P}(\Sigma^{+\infty}) \mid \text{IND}([\![P]\!])\}$ of programs (semantics) that use independent data for training and testing ML models. Thus $P \models \mathcal{I} \Leftrightarrow (\!|P|\!) \subseteq \mathcal{I}$.

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

$$\langle \mathcal{P}\left(\mathcal{P}\left(\Sigma^{+\infty}\right)\right), \subseteq \rangle \underset{\alpha_{I \rightsquigarrow U}}{\overset{\gamma_{I \rightsquigarrow U}}{\leftrightarrows}} \langle \mathcal{P}\left((\mathbb{X} \times \mathbb{N}) \times (\mathbb{X} \times \mathbb{N})\right), \supseteq \rangle \tag{3}$$

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 $\alpha_{I \rightsquigarrow U} \colon \mathcal{P}\left(\mathcal{P}\left(\Sigma^{+\infty}\right)\right) \rightarrow \mathcal{P}\left((\mathbb{X} \times \mathbb{N}) \times (\mathbb{X} \times \mathbb{N})\right)$ is:

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

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 changed to $\bar{v}$ (and all else remains the same, i.e., $\sigma(i) \stackrel{\bar{r}}{=} \sigma'(i) \wedge \sigma(I \setminus \{i\}) = \sigma'(I \setminus \{i\})$).

Note that our dependency abstraction generalizes that of Cousot [3] to non-deterministic programs and multi-dimensional data frame variables, thus tracking dependencies between portions of data frames. As in [3], 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 3).

Example 3 (Dependencies Between Data Frame Variables). Let us consider again the program P from Example 2. 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 / test data yields $\{i[1] \rightsquigarrow o_{\text{train}}[j], i[2] \rightsquigarrow o_{\text{train}}[j], i[3] \rightsquigarrow o_{\text{test}}[j], i[4] \rightsquigarrow o_{\text{test}}[j]\}$, $j \in \{1, 2\}$, 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, $\gamma_{\mathrm{I}_P \rightsquigarrow \mathrm{U}_P}(D) \stackrel{\text{def}}{=} \bigcup\{S \mid \alpha_{\mathrm{I}_P \rightsquigarrow \mathrm{U}_P}(S) \supseteq D\}$.

We can now define the *dependency semantics* $\Lambda_{I \rightsquigarrow U} \in \mathcal{P}\left((\mathbb{X} \times \mathbb{N}) \times (\mathbb{X} \times \mathbb{N})\right)$ 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 $(\!|P|\!)_{\rightsquigarrow}$ 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. $P \models \mathcal{I} \Leftrightarrow (\!|P|\!)_{\rightsquigarrow} \supseteq \alpha_{\mathrm{I}_P \rightsquigarrow \mathrm{U}_P}(\mathcal{I})$

3.3 Data Leakage Semantics

As hinted by Example 3, 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 $\Lambda_{\rightsquigarrow^+}$ pointwise [7] 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:

$$\langle \mathcal{P}\left((\mathbb{X} \times \mathbb{N}) \times (\mathbb{X} \times \mathbb{N})\right), \supseteq\rangle \underset{\dot{\alpha}}{\overset{\dot{\gamma}}{\leftrightarrows}} \langle \mathbb{X} \rightarrow (\mathbb{N} \rightarrow \mathcal{P}\left(\mathbb{X} \times \mathbb{N}\right)), \dot{\supseteq}\rangle \tag{4}$$

where the abstraction $\dot{\alpha}\colon \mathcal{P}\left((\mathbb{X} \times \mathbb{N}) \times (\mathbb{X} \times \mathbb{N})\right) \rightarrow (\mathbb{X} \rightarrow (\mathbb{N} \rightarrow \mathcal{P}\left(\mathbb{X} \times \mathbb{N}\right)))$ is:

$$\dot{\alpha}(D) \stackrel{\text{def}}{=} \lambda x \in \mathbb{X}\colon (\lambda r \in \mathbb{N}\colon \{i[r'] \mid i \in \mathbb{X}, r' \in \mathbb{N}, i[r'] \rightsquigarrow x[r] \in D\}) \tag{5}$$

Example 4 (Data Leakage Semantics). Let us consider again the last dependencies in Example 3: $\{i[1] \rightsquigarrow o_{\text{train}}[j], i[2] \rightsquigarrow o_{\text{train}}[j], i[3] \rightsquigarrow o_{\text{test}}[j], i[4] \rightsquigarrow o_{\text{test}}[j]\}$, $j \in \{1, 2\}$. Its abstraction following Equation 5 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 of the set of dependencies resulting from performing normalization *before* splitting into train and test data 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: $\dot{\gamma}(m) \stackrel{\text{def}}{=} \bigcap \{D \mid \dot{\alpha}(D) \mathrel{\dot{\supseteq}} m\}$.

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 the dependency semantics $\Lambda_{\rightsquigarrow}$: $\dot{\Lambda} \stackrel{\text{def}}{=} \dot{\alpha}(\Lambda_{\rightsquigarrow})$. In the following, we write $(\!|\dot{P}|\!)$ 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. $P \models \mathcal{I} \Leftrightarrow (\!|\dot{P}|\!) \mathrel{\dot{\supseteq}} \dot{\alpha}(\alpha_{\mathrm{I}_P \rightsquigarrow \mathrm{U}_P}(\mathcal{I}))$

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. *$P \models \mathcal{I} \Leftrightarrow \forall o_1 \in \mathrm{U}_P^{\text{train}}, o_2 \in \mathrm{U}_P^{\text{test}}\colon (\!|\dot{P}|\!)o_1 \cap (\!|\dot{P}|\!)o_2 = \emptyset$ where, with a slightly abuse of notation, $(\!|\dot{P}|\!)o$ stands for $\bigcup_{r \in \mathrm{dom}((\!|\dot{P}|\!)o)} (\!|\dot{P}|\!)o(r)$, i.e., the union of all sets $(\!|\dot{P}|\!)o(r)$ of rows of input data frame variables in the range of $(\!|\dot{P}|\!)o$ the data leakage semantics $(\!|\dot{P}|\!)$ for the used data frame variable o*

Example 5 (Continued from Example 4). The first map in Example 4 satisfies Lemma 1 since 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 second map in Example 4 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.

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 $\dot{\Lambda}_I$ 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 handles more advanced constructs such as branches, loops, and procedures calls, cf. Appendix D).

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

where $\mathit{name} \in \mathbb{W}$ 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 Python pandas) reads data from an input file and stores it into a variable y . The *select* statement (loosely corresponding to library functions such as `iloc`, `loc`, etc., in Python 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 merge operations between data frames (the *concat* and *join* operations roughly match the default Python pandas `concat` and `merge` library functions, respectively). The *function* statements modify a data frame x either by normalizing it (with the *normalize* function) or by applying some *other* function. The *normalize* function produces a *tainted* data frame y (representing normalization functions such as standardization or scaling in Python Sklearn). We assume that any *other* function does not produce tainted data frames. Finally, *use* statements employs data frames for either training ($f = \text{train}$) or testing ($f = \text{test}$) a ML model.

Constructive Data Leakage Semantics We can now instantiate the definition of our data leakage semantics $\dot{\Lambda}_I$ 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 input data frame variables I_P is given (with a slight abuse of notation, for simplicity) by the set of data files read by *source* statements, i.e., $\text{I}_P \overset{\text{def}}{=} i[\![P]\!] = i[\![S_n]\!] \circ \cdots \circ i[\![S_1]\!]\emptyset$, where $i[\![y = \text{read}(\mathit{name})]\!]I \overset{\text{def}}{=} I \cup \{\mathit{name}\}$ and $i[\![S]\!]I \overset{\text{def}}{=} I$ for any other statement S in P. Similarly, we define the set of used variables $\text{U}_P \overset{\text{def}}{=} u[\![P]\!] = u[\![S_n]\!] \circ \cdots \circ u[\![S_1]\!]\emptyset$, where $u[\![f(X)]\!]U \overset{\text{def}}{=} U \cup X$ and $u[\![S]\!]U \overset{\text{def}}{=} U$ for any other statement S, and analogously for $\text{U}_P^{\text{train}} \subseteq \text{U}_P$ (when $f = \text{train}$) and $\text{U}_P^{\text{test}} \subseteq \text{U}_P$ (when $f = \text{test}$).

Our constructive data leakage semantics is $(\!|\dot{P}|\!) \overset{\text{def}}{=} s[\![S_n]\!] \circ \cdots \circ s[\![S_1]\!]\dot{\emptyset}$ where $\dot{\emptyset}$ is the totally undefined function and the semantic function $s[\![S]\!]$ for each statement

S in P is defined as follows:

$$s[\![y = \text{read}(name)]\!]m \stackrel{\text{def}}{=} m\left[y \mapsto \lambda r \in R_{\text{read}()} \colon \{name[r]\}\right]$$
$$s[\![y = x.\text{select}[\bar{r}][C]]\!]m \stackrel{\text{def}}{=} m\left[y \mapsto \lambda r \in R_{x.\text{select}[\bar{r}][C]} \colon m(x)(\bar{r}[r])\right]$$
$$s[\![y = \text{concat}(x_1, x_2)]\!]m \stackrel{\text{def}}{=}$$
$$m\left[y \mapsto \lambda r \in R_{\text{concat}(x_1,x_2)} \colon \begin{cases} m(x_1)r & r \leq |\text{dom}(m(x_1))| \\ m(x_2)(r - |\text{dom}(m(x_1))|) & r > |\text{dom}(m(x_1))| \end{cases}\right]$$
$$s[\![y = \text{join}(x_1, x_2)]\!]m \stackrel{\text{def}}{=} m\left[y \mapsto \lambda r \in R_{\text{join}(x_1,x_2)} \colon m(x_1)\overleftarrow{r} \cup m(x_2)\overrightarrow{r}\right]$$
$$s[\![y = \text{normalize}(x)]\!]m \stackrel{\text{def}}{=} m\left[y \mapsto \lambda r \in R_{\text{normalize}(x)} \colon \bigcup_{r' \in \text{dom}(m(x))} m(x)r'\right]$$
$$s[\![y = \text{other}(x)]\!]m \stackrel{\text{def}}{=} m\left[y \mapsto \lambda r \in R_{\text{other}(x)} \colon m(x)r\right]$$
$$s[\![\text{use}(x)]\!]m \stackrel{\text{def}}{=} m$$

The semantics of the *source* statement maps each row r of a read data frame y to (the set containing) the corresponding row in the read data file ($name[r]$). 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 *concat* operation between two 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 ($\overleftarrow{r}$ and $\overrightarrow{r}$) in x_1 and x_2. 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 normalize function is a tainting function so the semantics for the *normalize* function 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.

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

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 $\langle \mathcal{C}, \sqsubseteq_C, \sqcup_C, \sqcap_C, \bot_C, \top_C \rangle$ where $\mathcal{C} \stackrel{\text{def}}{=} \mathcal{P}(\mathbb{W}) \cup \{\top_C\}$; $\mathbb{W}$ 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 $\mathcal{C}$ 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).

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 [4] *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 [14] for reference).

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 6 (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 $\phi_R^{-1}(3) = 13$).

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 $\mathcal{L} \stackrel{\text{def}}{=} \mathbb{W} \times \mathcal{C} \times \mathcal{R}$ contains triples of a data file name $file \in \mathbb{W}$, a column over-approximation $C \in \mathcal{C}$, and a row over-approximation $R \in \mathcal{R}$. In the following, we write file_R^C for the abstract data frame $\langle \mathrm{file}, C, R \rangle \in \mathcal{L}$. The partial order $\sqsubseteq_L$ compares abstract data frames derived from the same data files: $X_R^C \sqsubseteq_L Y_{R'}^{C'} \stackrel{\text{def}}{\Leftrightarrow} X = Y \land C \sqsubseteq_C C' \land R \sqsubseteq R'$.

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

$$\mathrm{overlap}(X_R^C, Y_{R'}^{C'}) \stackrel{\text{def}}{\Leftrightarrow} X = Y \land C \sqcap_C C' \neq \emptyset \land R \sqcap R' \neq \bot_R \tag{6}$$

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 7 (Abstract Data Frames). Let $\mathrm{file}_{[10,14]}^{\{id, city\}}$ abstract a data frame with columns $\{id, city\}$ and rows $\{10, 11, 12, 13, 14\}$ derived from a data source $file$. The abstract data frame $\mathrm{file}_{[12,15]}^{\{country\}}$ does not overlap with it, while $\mathrm{file}_{[12,15]}^{\{id\}}$ does. Joining $\mathrm{file}_{[10,14]}^{\{id, city\}}$ with $\mathrm{file}_{[12,15]}^{\{country\}}$ yields $\mathrm{file}_{[10,15]}^{\{id, city, country\}}$. Instead, the meet with $file_{[12,15]}^{\{id\}}$ yields $\mathrm{file}_{[12,14]}^{\{id\}}$. Finally, the constraining $\mathrm{file}_{[10,15]}^{\{id, city, country\}} \downarrow_{[1,2]}^{\{city\}}$ results in $\mathrm{file}_{[11,12]}^{\{city\}}$ (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 $\mathbb{L}(\mathbb{C}, \mathbb{R})$.

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 6). 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:

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

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

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

Example 8 (Abstract Data Frame Sets). Let us consider the join of two abstract data frame sets $S_1 = \left\{\text{file1}_{[1,10]}^{\{id\}}, \text{file2}_{[0,100]}^{\{name\}}\right\}$ and $S_2 = \left\{\text{file1}_{[9,12]}^{\{id\}}, \text{file3}_{[0,100]}^{\{zip\}}\right\}$ Before reduction, we obtain $\left\{\text{file1}_{[1,10]}^{\{id\}}, \text{file1}_{[9,12]}^{\{id\}}, \text{file2}_{[0,100]}^{\{name\}}, \text{file3}_{[0,100]}^{\{zip\}}\right\}$. The reduction operation makes the set canonical: $\left\{\text{file1}_{[1,12]}^{\{id\}}, \text{file2}_{[0,100]}^{\{name\}}, \text{file3}_{[0,100]}^{\{zip\}}\right\}$.

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

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 on.

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 $B \in \mathbb{B} \stackrel{\text{def}}{=} \{\text{UNTANINTED}, \text{MAYBE-TAINTED}\}$. 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 UNTAINTED $\leq$ MAYBE-TAINTED. The bottom element $\bot_A$ is $\langle \emptyset, \text{UNTAINTED} \rangle$.

Finally, we define the concretization $\gamma \colon (\mathbb{X} \to (\mathbb{N} \to \mathcal{A})) \to (\mathbb{X} \to (\mathbb{N} \to \mathcal{P}(\mathbb{X} \times \mathbb{N})))$: $\gamma(m) \stackrel{\text{def}}{=} \lambda x \in \mathbb{X} \colon (\lambda r \in \mathbb{N} \colon \gamma_A(m(x)))$, where $\gamma_A \colon \mathcal{A} \to \mathcal{P}(\mathbb{X} \times \mathbb{N})$ is $\gamma_A(\langle S, B \rangle) \stackrel{\text{def}}{=} \left\{X[r] \mid X_R^C \in S, r \in \gamma_R(R)\right\}$ (with $\gamma_R \colon \mathcal{R} \to \mathcal{P}(\mathbb{N})$ being the concretization function for row abstractions, cf. Section 4.1). 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 $(\!|\dot{P}|\!)^\natural \stackrel{\text{def}}{=} a[\![S_n]\!] \circ \cdots \circ a[\![S_1]\!]\dot{\bot}_A$ where $\dot{\bot}_A$ maps all data frame variables to $\bot_A$ and the abstract semantic function $a[\![S]\!]$ for each statement in P is defined as follows:

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

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

$$
\begin{aligned}
a[\![y = \mathrm{op}(x_1, x_2)]\!]m &\stackrel{\mathrm{def}}{=} m\,[y \mapsto \langle m_s(x_1) \sqcup_S m_s(x_2), m_b(x_1) \vee m_b(x_2)\rangle] \\
a[\![y = \mathrm{normalize}(x)]\!]m &\stackrel{\mathrm{def}}{=} m\,[y \mapsto \langle m_s(x), \mathrm{TRUE}\rangle] \\
a[\![y = \mathrm{other}(x)]\!]m &\stackrel{\mathrm{def}}{=} m\,[y \mapsto \langle m_s(x), m_b(x)\rangle] \\
a[\![\mathrm{use}(x)]\!]m &\stackrel{\mathrm{def}}{=} m
\end{aligned}
$$

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 ($name^{\top_C}_{[0,\infty]}$). 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) 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 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 *normalize* function 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. [20], we do not perform any renaming or resetting of the data source mapping (cf. Section 4.2 in [20]) 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. $P \models \mathcal{I} \Leftarrow \gamma(\dot{(\!|P|\!)}^\natural) \mathrel{\dot{\supseteq}} \dot{\alpha}(\alpha_{\rightsquigarrow^+}(\mathcal{I}))$

Similarly, we have the sound but not complete counterpart of Lemma 1 for practically checking absence of data leakage:

Lemma 2.

$$
\begin{aligned}
P \models \mathcal{I} \Leftarrow\; & \forall o_1 \in \mathrm{U}_P^{\mathrm{train}}, o_2 \in \mathrm{U}_P^{\mathrm{test}}: \\
& \bigcup_{r_1 \in \mathrm{dom}(\gamma(\dot{(\!|P|\!)}^\natural) o_1)} \gamma(\dot{(\!|P|\!)}^\natural) o_1(r_1) \;\cap \bigcup_{r_2 \in \mathrm{dom}(\gamma(\dot{(\!|P|\!)}^\natural) o_2)} \gamma(\dot{(\!|P|\!)}^\natural) o_2(r_2) = \emptyset \\
& \Leftrightarrow \forall o_1 \in \mathrm{U}_P^{\mathrm{train}}, o_2 \in \mathrm{U}_P^{\mathrm{test}}: \forall X^C_R \in \dot{(\!|P|\!)}^\natural_s o_1, Y^{C'}_{R'} \in \dot{(\!|P|\!)}^\natural_s o_2: \\
& \neg\mathrm{overlap}(X^C_R, Y^{C'}_{R'}) \wedge \Big(X = Y \Rightarrow \neg\dot{(\!|P|\!)}^\natural_b o_1 \wedge \neg\dot{(\!|P|\!)}^\natural_b o_2\Big)
\end{aligned}
$$

Specifically, Lemma 2 allows 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, $\neg\mathrm{overlap}(X^C_R, Y^{C'}_{R'})$, cf. Equation 6) 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 9 (Motivating Example (Continued)). The data leakage analysis of our motivating example (cf. Example 1) is the following:

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

$$a\left[\!\left[\texttt{X = data.select[][\{"X_1", "X_2", "y"\}]}\right]\!\right] m_1 =$$
$$\left(m_2 \stackrel{\text{def}}{=} m_1\left[X \mapsto \left\langle \left\{ \text{data.csv}^{\{``X_1", ``X_2", ``y''\}}_{[0,\infty]} \right\}, \text{FALSE} \right\rangle\right]\right)$$

$$a\left[\!\left[\texttt{X_norm = normalize(X)}\right]\!\right] m_2 = \left(m_3 \stackrel{\text{def}}{=} m_2\left[X_\text{norm} \mapsto \left\langle \left\{ \text{data.csv}^{\{``X_1", ``X_2", ``y''\}}_{[0,\infty]} \right\}, \text{TRUE} \right\rangle\right]\right)$$

$$a\left[\!\left[\texttt{X_train = X_norm.select[[}\lfloor 0.025 * R_{\texttt{X_norm}} \rfloor \texttt{ + 1, ..., } R_{\texttt{X_norm}}\texttt{]][]}\right]\!\right] m_3 =$$
$$\left(m_4 \stackrel{\text{def}}{=} m_3\left[X_\text{train} \mapsto \left\langle \left\{ \text{data.csv}^{\{``X_1", ``X_2", ``y''\}}_{[\lfloor 0.025 * R_{X_\text{norm}} \rfloor + 1, R_{X_\text{norm}}]} \right\}, \text{TRUE} \right\rangle\right]\right)$$

$$a\left[\!\left[\texttt{X_test = X_norm.select[[0, ..., }\lfloor 0.025 * R_{\texttt{X_norm}} \rfloor\texttt{]][]}\right]\!\right] m_4 =$$
$$\left(m_5 \stackrel{\text{def}}{=} m_4\left[X_\text{test} \mapsto \left\langle \left\{ \text{data.csv}^{\{``X_1", ``X_2", ``y''\}}_{[0, \lfloor 0.025 * R_{X_\text{norm}} \rfloor]} \right\}, \text{TRUE} \right\rangle\right]\right)$$

$$a\left[\!\left[\texttt{test(X_test)}\right]\!\right](a\left[\!\left[\texttt{train(X_train)}\right]\!\right] m_5) = m_5$$

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

5 Experimental Evaluation

We implemented our static analysis into the open source NBLyzer [20] framework for data science notebooks. NBLyzer performs the analysis starting on an individual code cell (*intra-cell analysis*) and, based on the resulting abstract state, it proceeds to analyze *valid* successor code cells (*inter-cell analysis*). Whether a code cell is a valid successor or not, is specified by an analysis-dependent *cell propagation condition*. We refer to the original NBLyzer paper [20] and to Appendix C and D for further details.

We evaluated[3] our analysis against the data leakage analysis previously implemented in NBLyzer [20], using the same benchmarks. These are notebooks written to succeed in non-trivial real data science tasks and can be assumed to closely represent code written by non-novice data scientists.

Table 1: Alarms raised by the previous [20] vs our analysis.

Analysis	TP		FP
	Taint	Overlap	
[20]	10	0	2
Ours	**10**	**15**	**2**

We summarize the results in Table 1. For each reported alarm, we engaged 4 data scientists at Microsoft to determine true (TP) and false positives (FP). We further classified the true positives as due to a normalization taint (Taint) or overlapping data frames (Overlap). Our analysis found 10 Taint data leakages in 5 notebooks,

[3] Experiments done on a Ryzen 9 6900HS with 24GB DDR5 running Ubuntu 22.04.

and 15 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 [11]). The previous analysis only found 10 Taint leakages in 5 notebooks. It could not Overlap data leakages because it cannot reason at the granularity of partial data frames. The cost for our more precise analysis is a mere 7% slowdown. Both analyses reported 2 false positives, due to different objects having the same function name (e.g., `LabelEncoder` and `StandardScaler` both having the function `fit_transform`). This issue can be solved by introducing object sensitive type tracking in our analysis. We leave it for future work.

The full experimental evaluation is described in Appendix E.

6 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 [22] and the definition of dependency abstraction used by Cousot [3]. In particular, among other things, the generalization involves reasoning about dependencies (and thus data usage relationships) between multi-dimensional variables. In [22], 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 [22]). 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 [3], Cousot shows that information flow, slicing, non-interference, dye, and taint analyses are all further abstractions of his proposed framework (cf. Section 7 in [3]). 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.

Static Analysis for Data Science. Static analysis for data science is an emerging area in the program analysis community [21]. Some notable static analyses for data science scripts include an analysis for ensuring correct shape dimensions in TensorFlow programs [12], an analysis for constraining inputs based on program constraints [22], and a provenance analysis [15]. In addition, static analyses have been proposed for data science notebooks [20,13]. NBLyzer [20] contains an ad-hoc data leakage analysis that detects Taint data leakages [20]. An extension to handle Overlap data leakages was sketched in [19] (but no analysis precision results were reported). However, both these analysis are not formal-

ized nor formally proven sound[4]. In contrast, we introduce a sound data leakage analysis that has a rigorous semantic underpinning.

7 Conclusion

We have presented an approach for detecting data leakages statically. We provide a formal and rigorous derivation from the standard program collecting semantics, via successive abstraction to a final sound and computable static analysis definition. We implement our analysis in the NBLyzer framework and demonstrate clear improvements upon previous ad-hoc data leakage analyses.

Acknowledgements. We thank our colleagues at Microsoft Azure Data Labs and Microsoft Development Centre Serbia for all their feedback and support.

References

1. Chouldechova, A., Prado, D.B., Fialko, O., Vaithianathan, R.: A Case Study of Algorithm-Assisted Decision Making in Child Maltreatment Hotline Screening Decisions. In: FAT (2018)
2. Cousot, P.: Constructive Design of a Hierarchy of Semantics of a Transition System by Abstract Interpretation. Electronic Notes in Theoretical Computer Science **277**(1-2), 47–103 (2002)
3. Cousot, P.: Abstract Semantic Dependency. In: SAS. pp. 389–410 (2019)
4. Cousot, P., Cousot, R.: Static Determination of Dynamic Properties of Programs. In: Second International Symposium on Programming. pp. 106–130 (1976)
5. Cousot, P., Cousot, R.: Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: POPL. pp. 238–252 (1977)
6. Cousot, P., Cousot, R.: Systematic Design of Program Analysis Frameworks. In: POPL. pp. 269–282 (1979)
7. Cousot, P., Cousot, R.: Higher Order Abstract Interpretation (and Application to Comportment Analysis Generalizing Strictness, Termination, Projection, and PER Analysis. In: ICCL. pp. 95–112 (1994)
8. Guzharina, A.: 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/` (2020), accessed: 22-01-22
9. Kapoor, S., Narayanan, A.: Leakage and the reproducibility crisis in machine-learning-based science. Patterns **4**(9), 100804 (2023)
10. Kaufman, S., Rosset, S., Perlich, C., Stitelman, O.: Leakage in Data Mining: Formulation, Detection, and Avoidance. ACM Transactions on Knowledge Discovery from Data **6**(4) (2012)
11. Kharkar, A., Moghaddam, R.Z., Jin, M., Liu, X., Shi, X., Clement, C., Sundaresan, N.: Learning to Reduce False Positives in Analytic Bug Detectors. In: ICSE. p. 1307–1316 (2022)

[4] We found a number of soundness issues in [19] when working on our formalization.

12. Lagouvardos, S., Dolby, J., Grech, N., Antoniadis, A., Smaragdakis, Y.: Static Analysis of Shape in TensorFlow Programs. In: ECOOP. pp. 15:1–15:29 (2020)
13. Macke, S., Gong, H., Lee, D.J.L., Head, A., Xin, D., Parameswaran, A.G.: Fine-Grained Lineage for Safer Notebook Interactions. CoRR **abs/2012.06981** (2020), `https://arxiv.org/abs/2012.06981`
14. Miné, A.: Weakly Relational Numerical Abstract Domains. Ph.D. thesis, École Polytechnique, Palaiseau, France (2004), `https://tel.archives-ouvertes.fr/tel-00136630`
15. Namaki, M.H., Floratou, A., Psallidas, F., Krishnan, S., Agrawal, A., Wu, Y., Zhu, Y., Weimer, M.: Vamsa: Automated Provenance Tracking in Data Science Scripts. In: KDD. pp. 1542–1551 (2020)
16. Nisbet, R., Miner, G., Yale, K.: Handbook of Statistical Analysis and Data Mining Applications. Academic Press, Boston, second edition edn. (2018). `https://doi.org/https://doi.org/10.1016/C2012-0-06451-4`
17. Papadimitriou, P., Garcia-Molina, H.: A Model for Data Leakage Detection. In: ICDE. pp. 1307–1310 (2009)
18. Perkel, J.: Why Jupyter is Data Scientists' Computational Notebook of Choice. Nature **563**, 145–146 (11 2018)
19. Subotić, P., Bojanić, U., Stojić, M.: Statically Detecting Data Leakages in Data Science Code. In: SOAP. pp. 16–22 (2022)
20. Subotić, P., Milikić, L., Stojić, M.: A Static analysis Framework for Data Science Notebooks. In: ICSE. pp. 13–22 (2022)
21. Urban, C.: Static analysis of data science software. In: SAS. pp. 17–23 (2019)
22. Urban, C., Müller, P.: An Abstract Interpretation Framework for Input Data Usage. In: ESOP. pp. 683–710 (2018)
23. 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.: External Validation of a Widely Implemented Proprietary Sepsis Prediction Model in Hospitalized Patients. JAMA Internal Medicine (2021)

A Proofs for Section 3 (Concrete Data Leakage Semantics)

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

Proof. Let $P \models \mathcal{I}$. From the subset inclusion in Section 3.1, we have that $(\!|P|\!) \subseteq \mathcal{I}$. Thus, from the Galois connection in Equation 3 (note the inverse $\supseteq$ order in the abstract domain!), we have $\alpha_{\mathrm{I}_P \rightsquigarrow \mathrm{U}_P}((\!|P|\!)) \supseteq \alpha_{\mathrm{I}_P \rightsquigarrow \mathrm{U}_P}(\mathcal{I})$. From the definition of $(\!|P|\!)_{\rightsquigarrow}$, we can then conclude that $(\!|P|\!)_{\rightsquigarrow} \supseteq \alpha_{\mathrm{I}_P \rightsquigarrow \mathrm{U}_P}(\mathcal{I})$.

Vice versa, let $(\!|P|\!)_{\rightsquigarrow} \supseteq \alpha_{\mathrm{I}_P \rightsquigarrow \mathrm{U}_P}(\mathcal{I})$. From the definition of $(\!|P|\!)_{\rightsquigarrow}$, we have $\alpha_{\mathrm{I}_P \rightsquigarrow \mathrm{U}_P}((\!|P|\!)) \supseteq \alpha_{\mathrm{I}_P \rightsquigarrow \mathrm{U}_P}(\mathcal{I})$, and from the Galois connection in Equation 3 we have $(\!|P|\!) \subseteq \gamma_{\mathrm{I}_P \rightsquigarrow \mathrm{U}_P}(\alpha_{\mathrm{I}_P \rightsquigarrow \mathrm{U}_P}(\mathcal{I}))$. From the definition of $\gamma_{\mathrm{I}_P \rightsquigarrow \mathrm{U}_P}$, we have $(\!|P|\!) \subseteq \mathcal{I}$ and we can thus conclude $(\!|P|\!) \subseteq \mathcal{I}$.

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

Proof. The proof is analogous to that of Theorem 1. Let $P \models \mathcal{I}$. We have that $(\!|P|\!)_{\rightsquigarrow} \supseteq \alpha_{\mathrm{I}_P \rightsquigarrow \mathrm{U}_P}(\mathcal{I})$ from Theorem 1. From the Galois connection in Equation 4, we have $\dot{\alpha}((\!|P|\!)_{\rightsquigarrow}) \dot{\supseteq} \dot{\alpha}(\alpha_{\mathrm{I}_P \rightsquigarrow \mathrm{U}_P}(\mathcal{I}))$. Thus, from the definition of $(\!|\dot{P}|\!)$, we can conclude that $(\!|\dot{P}|\!) \dot{\supseteq} \dot{\alpha}(\alpha_{\mathrm{I}_P \rightsquigarrow \mathrm{U}_P}(\mathcal{I}))$.

Vice versa, let $(\!|\dot{P}|\!) \dot{\supseteq} \dot{\alpha}(\alpha_{\mathrm{I}_P \rightsquigarrow \mathrm{U}_P}(\mathcal{I}))$. From the definition of $(\!|\dot{P}|\!)$ we have $\dot{\alpha}((\!|P|\!)_{\rightsquigarrow}) \dot{\supseteq} \dot{\alpha}(\alpha_{\mathrm{I}_P \rightsquigarrow \mathrm{U}_P}(\mathcal{I}))$, and from the Galois connection in Equation 4 we have $(\!|P|\!)_{\rightsquigarrow} \supseteq \dot{\gamma}(\dot{\alpha}(\alpha_{\mathrm{I}_P \rightsquigarrow \mathrm{U}_P}(\mathcal{I})))$. From the definition of $\dot{\gamma}$ derived from $\dot{\alpha}$, we have $(\!|P|\!)_{\rightsquigarrow} \supseteq \alpha_{\mathrm{I}_P \rightsquigarrow \mathrm{U}_P}(\mathcal{I})$ and from Theorem 1 we can thus conclude $P \models \mathcal{I}$.

B Proofs for Section 4 (Data Leakage Analysis)

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

Proof (Sketch). 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.

C NBLyzer Framework

NBLYZER is designed specifically to adapt to the unique out-of-order execution semantics of notebooks. It performs the analysis starting on an individual code cell (*intra-cell analysis*) and, based on the resulting abstract state, it proceeds to analyze *valid* successor code cells (*inter-cell analysis*). Whether a code cell is a valid successor or not, is specified by an analysis-dependent *cell propagation ϕ-condition*.

Let F_c be the abstract transformer that performs the analysis of an individual code cell c. The inter-cell analysis process of NBLYZER is visualized by the propagation tree in Figure 1. At the intra-analysis level of each code cell c, the

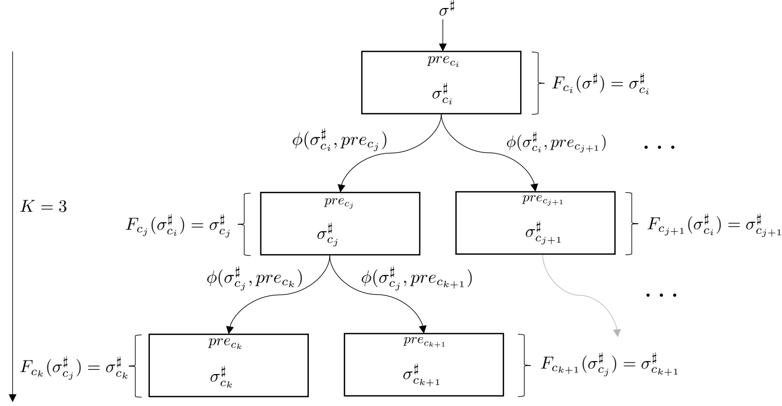

Fig. 1: Inter-cell analysis

abstract transformer F_c is applied to the current abstract state $\sigma^\sharp$ and returns an updated abstract state, i.e., $F_c(\sigma^\sharp) = \sigma^{\sharp\prime}$. The updated abstract state is propagated from one cell c to another cell c' if the ϕ-condition holds. The ϕ-condition depends on the incoming abstract state $F_c(\sigma^\sharp) = \sigma^{\sharp\prime}$ and a *cell pre-condition* $pre_{c'}$ for c'. The cell pre-condition contains, e.g., unbound variables (i.e., variables used but not defined within the cell) and namespaces for libraries. The cell pre-conditions used in our analysis are specified in Section D.

Each propagation branch may terminate due to the following four cases:

1. the depth (number of individual code cells) of the propagation reaches a given (finite) *propagation bound* $K \in \mathbb{N}$;
2. the ϕ-condition does not hold for all code cells in the notebook;
3. a fixpoint subsumption occurred: the nth time, $n > 1$, a cell was analyzed does not result in a change in the abstract state;
4. a error e.g., a data leakage, has been detected that halts the propagation.

It is also possible to not specify a finite propagation bound, i.e., $K = \infty$; in this case, condition (1) is ignored. When all propagation branches terminate, the inter-cell analysis terminates.

D Implementation

We integrate our static analysis based on our approach described in Section 4 into the open source NBLyzer [20] framework for data science notebooks.

NBLyzer provides the fixpoint machinery to execute an abstract semantics on a notebook as well as a reference data leakage analysis that we aim to improve on in this paper. Please refer to Appendix C for an overview of NBLyzer. Below we describe the main tasks required to integrate our data leakage analyzer into NBLyzer as well as several limitations of our implementation.

Cell Propagation (ϕ): In the NBLyzer framework each analysis, aside from implementing an abstract domain and abstract transfer functions, needs to define a ϕ-condition to determine if propagation should continue to another cell. To achieve good performance, ϕ must be defined as strong as possible while not sacrificing soundness i.e., we do not want to miss any interesting execution sequences (e.g., containing a bug) by terminating prematurely.

Moreover, each cell has a set of pre-condition variables pre which we define as a subset of of unbound variables and namespaces that are used to invoke functions in the knowledge base or propagated to other cells. This includes include namespaces for libraries. For our data leakage analysis, only namespaces that relate to functions in our knowledge base (see below) are considered. We therefore specify the ϕ-condition for inter-cell propagation as follows:

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

where $m = \sigma_c^\sharp$ is the abstract state resulting from the analysis of the individual code cell c. This rule stipulates the condition by which a successor cell should be analyzed. That is, if any variable that has rows (not $\bot$) in the abstract state of the current notebook cell, is also unbound in the successor notebook cell, we proceed to propagate the abstract state.

Support for Functions: We support inter-procedural analysis via function inlining/cloning. If a function has been in an executed cell, we inline its body in any subsequent call site before processing the cell. In the case the definition does not exist in a predecessor cell, we treat the function as a undefined function.

E Extended Evaluation

E.1 Experimental Setup

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 both versions of NBLyzer, one running our data leakage static analysis and the other the data leakage analysis from [20]. We used default NBLyzer settings (e.g., $K = 5$).

Benchmarks We use a data science notebook benchmark suite consisting of 4 Kaggle competitions that has previously been used to evaluate data science static analyzers [15,20]. We analyzed 2111 over 2413 notebooks, resulting in 7378 notebook executions. We excluded 302 notebooks because they could not be digested by our analyzer (i.e., syntax errors, JSON decoding errors, etc.). The benchmark characteristics are summarized in Table 2.

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. 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 has on average 3 functions and 0.1 classes defined.

Table 2: Kaggle Notebook Benchmark Characteristics

Characteristic	Mean	SD	Max	Min
Cells (per-notebook)	23.58	20.21	182	1
Lines of code (per-cell)	9.12	13.55	257	1
Branching inst. (per-cell)	0.43	2.49	76	0
Functions (per-notebook)	3.33	7.11	72	0
Classes (per-notebook)	0.14	0.64	11	0
parse error cell (per-notebook)	0.5	0.98	20	0
Variables (per-cell)	8.2	2.3	552	0
Unbound variables (per-cell)	2.1	1.06	12	0

Use Case As described in [20], the use case is a data leakage detector for notebooks in an Integrated Development Environment (IDE). The analysis is triggered by an event (such as a cell execution) and provides a *what-if* analysis, namely: *"if you do this event then the following future cell executions may lead to a data leakage"*. The analyzer should identify data leakages with a *soft* analysis deadline of 1 second in accordance to the RAIL performance model[5].

Experimental methodology For every notebook we perform our analysis for a *valid execution*. We define a valid execution as a non-empty sequence of cells that starts with a cell that does not contain unbound variables and thus can commence a valid execution.

E.2 Extended Precision Evaluation

To further provide the reader with an appreciation of how data leakages manifest in our benchmarks, a code snippet showing an overlapping data error (in contrast to the normalization taint error in our motivating example) discovered from our benchmarks is shown in Figure 2. Here repeated training and testing occurs on the same training set. In the code, a data frame is first loaded to a variable called `df`. After several exploratory data analysis (EDA) steps (not shown), the features (columns which are inputs to the model) and the target (prediction column) is split into two data frames, called `X` and `y` respectively. These two data frame variables are split into four (`X_train`, `X_test`, `y_train`, `y_test`) reflecting the training and testing subsets. The location of the split is defined by the `split` variable. The programmer, appears unsure about the semantics of `iloc` and assumes 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 testing contain the row at the value of `split`. The model, represented by the `lr_clf` variable is created and the training is performed (invocation of `fit` method), using the train suffixed data frames. In the last cell, prediction is made (invocation of `predict` function) however with overlapping data. We note, this type of bug cannot be detected using a taint analysis. We elaborate on this further in Section 6.

[5] `https://web.dev/rail/`

Imports, initialization etc.

...

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))
```

Fig. 2: Example of overlapping data bug

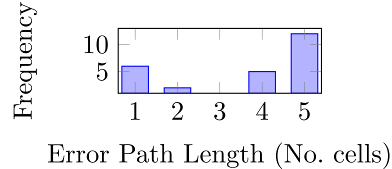

Fig. 3: Error path length frequency

While our formalization in Section 3 is sound w.r.t. our simple example language. We report several sources of unsoundness in our actual analysis implementation. Firstly, we do not support every Python 3 construct and instead focus on the most common constructs observed in notebook code. Constructs such as dynamic code evaluation, reference aliasing are not supported. We remark that data science Python programs are relatively simple compared to general Python programs. They tend to have largely linear control-flow and generally have a single call-site per function making cloning/inlining a reasonable choice. From our benchmarks we could only detect that 0.5% of notebooks required an alias analysis (e.g., assigning data frames by reference) and for this reason we did not integrate an alias analysis. We find that it is infeasible to manually inspect all notebooks, and thus it is impractical to produce a recall rate for our benchmarks.

The data leakages found had cell execution traces varied between $1-5$ cells in length indicating that the majority of bugs (76%) manifested over several notebook cells, in a non-sequential cell order. We summarise these findings in the histogram in Figure 3.

Given a simple knowledge base the coverage of our analysis is subject to an adequate formulation of the knowledge base which is elementary in our implementation, following the implementation in [20]. Moreover, we believe lower-quality notebooks e.g., university subject assignments, may contain more data

leakages. We leave additional modeling and precision investigation on other benchmarks for future work.

We also note that we were not able to find instances that required widening and did not experience infinite (or very high) ascending chains being computed when analyzing our benchmarks. For this to occur we would need code that iterates on data frame rows, which is not common. Standard interval widening [5] can be applied if required.

E.3 Performance Evaluation

We compare the runtimes of our analyzer (Extension) to the NBLyzer data leakage analysis from [20] (NBLyzer) and show their average executions per notebook in Figure 4 (note, a single notebook may have several executions). Overall, our analysis experiences a slowdown of 7%. For the vast majority, the difference in runtime is negligible. However, for several cases we experience a noticeable slowdown which we believe is due to our extended semantics that requires more data to be stored in the abstract state and more complex operations (join etc.). On the other hand, we also see the opposite occurring on a small number of benchmarks, where our analysis is faster. Here our speedup is due to discovering a bug and terminating execution while NBLyzer continues to propagate. As with NBLyzer the majority (over 99%) of analyses completed in less than 1 second except for a few cases. Pathological cases occur typically due to a notebook having large inter-connectivity between cells thus resulting in a large number of execution traces.

Overall, we find that considering the increase in bug detection, the 7% slowdown is a small price to pay and does not appear to significantly degrade the user experience for the interactive notebook use case.

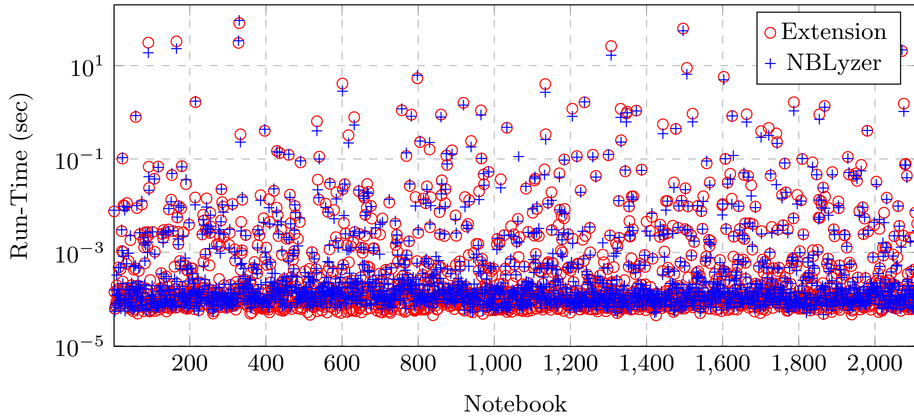

Fig. 4: Average Execution Run-times Per Notebook of Extension vs NBLyzer