Abstract Interpretation-Based Data Leakage Static Analysis

FILIP DROBNJAKOVIĆ and PAVLE SUBOTIĆ, Microsoft, Serbia
CATERINA URBAN, Inria & ENS | PSL, France

Data leakage is a well-known problem in machine learning. Data leakage occurs when information from outside the training dataset is used to create a model. This phenomenon renders a model excessively optimistic or even useless in the real world since the model tends to leverage greatly on the unfairly acquired information. To date, detection of data leakages occurs post-mortem using run-time methods. However, due to the insidious nature of data leakage, it may not be apparent to a data scientist that a data leakage has occurred in the first place. For this reason, it is advantageous to detect data leakages as early as possible in the development life cycle.

In this paper, we propose a novel static analysis to detect several instances of data leakages during development time. We define our analysis using the framework of abstract interpretation: we define a concrete semantics that is sound and complete, from which we derive a sound and computable abstract semantics. We implement our static analysis inside the open-source NBLyzer static analysis framework and demonstrate its utility by evaluating its performance and precision on over 2000 Kaggle competition notebooks.

CCS Concepts: • **Software and its engineering → Functionality**; **Integrated and visual development environments**.

Additional Key Words and Phrases: static analysis, abstract interpretation, data science

ACM Reference Format:
Filip Drobnjaković, Pavle Subotić, and Caterina Urban. 2018. Abstract Interpretation-Based Data Leakage Static Analysis. In . ACM, New York, NY, USA, 21 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

As artificial intelligence (AI) continues its unprecedented impact on society, ensuring machine learning (ML) models are accurate is crucial to many important facets of life. For ML models to perform accurate inference, they need to be correctly trained and tested. This iterative task is performed typically within data science notebook environments [1, 23] which contain code that utilizes relational data abstractions such as Python pandas data frames [4] to read raw data, transform its contents before training and testing the ML model. Consequently, the correctness of these data science notebooks is vital to ensuring functional AI systems. A notable data science bug that can occur during this process is known as a *data leakage* [22]. 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 *implicitly* derive data from a dataset. For example, input data is commonly split into training and testing data. However, if a transformation such as data normalization is performed before the split, the entire dataset is transformed based on the entire data set and thus any splitting of the data cannot guarantee disjointness due to an indirect dependency. This may result in a seemingly accurate model that does not perform well in practice since the model tends to leverage on the unfairly acquired information.

Conference acronym 'XX, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

Example 1.1 (Motivating Example). Consider the data science notebook below. This notebook is comprised of 6 cells (numbered from $1-6$ in the order they are defined). Lets assume the cells have been executed in sequential order. This is indicated by the numbering on the left-hand-side of each cell. *Cell 1*, imports required libraries. *Cell 2* reads in data from a CVS file. *Cell 3*, transforms the data, *Cell 4*, splits the data into train and test segments. *Cell 5*, trains the model and *Cell 6* tests the model and outputs its accuracy.

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

In [2]:
```
data = pd.read_csv("data.csv")
X = data[["X_1", "X_2"]]
y = data[["y"]]
```

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, random_state=2)
```

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

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

Out[6]:
```
0.67
```

Note, that the output of *Cell 6* indicates a reasonable accuracy i.e., 0.67. This could be a sign of a well trained model. However in this particular case, it is the result of a data leakage. Namely, due to the fact that *Cell 3* performs a standardization of the *entire* data. This implicitly uses the mean of the entire data set to perform the transformation. The split is then performed in *Cell 4 after* the transformation and thus our data is overly optimistic when training and fitting as the train and test data are implicitly dependent on each other. For this example, if we were to normalize after the split, we would get a poor accuracy of 0.33.

The code above highlights the ease of inducing a data leakage. Even though the training and testing data is seemingly disjoint, the fact that the normalization function is called *beforehand* means that a data leakage is possible. Of course, more obvious data leakage can be encountered by simply not performing a split, splitting so that the data is not disjoint etc. What makes data leakages particularly malignant is that they are silent bugs that will not cause run-time errors, or exceptions. Instead they may only noticed when performing badly on real world data.

Mainstream methods rely on detecting data leakages post mortem [17, 22]. Given a suspicious result e.g., an overly accurate model, data analyses methods are used to identify data dependencies. These methods are powerful when investigating the relationships between data, however, as demonstrated in our example, a reasonable result may indeed avoid suspicion from a data scientist

until the model is already deployed. Therefore, it is advantageous to detect data leakages as early as possible in the development life cycle. This is a natural use case for *static analysis*, i.e., a tool that can detect data leakages at coding time would be a great benefit to data scientists and can be complement existing postmortem data analyses techniques. While a plethora of state-of-the-art static analyzers exist that target common programming bugs, they do not support domain specific bugs such as data leakages and certainly not in data science notebook environments.

In this paper, we propose a novel static analysis for detecting several classes of data leakages in data science notebooks and scripts. Our static analysis is constructed based on abstract interpretation [13]. We firstly define our property of interest, namely, the absence of data leakages in the precise trace semantics. We then use the theory of abstract interpretation to systematically and rigorously derive, by successive abstractions, a sound and computable abstract semantics which we term *data leakage semantics*. Based on our data leakage semantics, we instantiate a static analysis that tracks the origin of data frame cells and determines if two data frames originate from overlapping or tainted data sources. For instance, when a variable is an argument to a function that trains or tests a model, we assert that the variable is *disjoint* and untainted. In our motivating example our analysis can identify that there is a potential taint between `X_test` and `X_train` since they both originate from previously normalized data, despite being disjoint.

We implement our analysis for a subset of Python 3 in the open source NBLyzer [26] static analysis framework that supports data science notebook semantics. We evaluate the performance of our analyzer on 2088 real-world competition notebooks and demonstrate that our approach scales to the performance constraints of interactive notebook environments while detecting 30 real data leakages with a precision of 94%.

We summarize our contributions below:

- We present, to the best of knowledge, the first rigorous formalization of data leakage semantics using the abstract interpretation framework. As a consequence, we propose a dependency semantics that generalizes existing work [11] for multi-dimensional data.
- We define a novel data leakage analysis based on our data leakage semantics that detects data leakages in data frame-manipulating programs.
- We implement our analysis in the NBLyzer static analysis framework for data science notebooks and define analysis specific operators such as ϕ-propagation that incorporates out-of-order execution semantics of notebooks in our analysis.
- We evaluate our analysis on real-world data science notebooks and demonstrate it performs to low-latency notebook environment constraints, and that it can detect data leakages in real-word notebooks with a precision of 94%.

The paper is organized as follows: In Section 2 we provide the necessary background. In Section 3, we define a data leakage semantics by way of successive intermediate abstractions. In Section 4.2, we describe a data leakage analysis that is based on the semantics of Section 3. In Section 5, we describe the implementation of our data leakage analysis in the NBLyzer framework including additional operators for supporting notebook semantics. In Section 6, we provide an extensive evaluation of our data leakage analysis implementation on 2088 real-world competition notebooks. In Section 7, we contrast our analysis to related work and we draw relevant conclusions in Section 8.

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

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

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 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 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 [10, 13]. The set $\Omega \stackrel{\text{def}}{=} \{s \in \Sigma \mid \forall s' \in \Sigma : \langle s, s' \rangle \notin \tau\}$ is the set of *final states* of the program.

In the following, let $\Sigma^n \stackrel{\text{def}}{=} \{s_0 \cdots s_{n-1} \mid \forall i < n : s_i \in \Sigma\}$ be the set of all sequences of exactly n program states. We write ε to denote the empty sequence, i.e., $\Sigma^0 \stackrel{\text{def}}{=} \{\varepsilon\}$. Let $\Sigma^\star \stackrel{\text{def}}{=} \bigcup_{n \in \mathbb{N}} \Sigma^n$ be the set of all finite sequences, $\Sigma^+ \stackrel{\text{def}}{=} \Sigma^\star \setminus \Sigma^0$ be the set of all non-empty finite sequences, Σ^ω be the set of all infinite sequences, $\Sigma^{+\infty} \stackrel{\text{def}}{=} \Sigma^+ \cup \Sigma^\omega$ be the set of all non-empty finite or infinite sequences and $\Sigma^{\star\infty} \stackrel{\text{def}}{=} \Sigma^\star \cup \Sigma^\omega$ be the set of all finite or infinite sequences of program states. In the following, we write $\sigma\sigma'$ for the concatenation of two sequences $\sigma, \sigma' \in \Sigma^{\star\infty}$ (with $\sigma\varepsilon = \varepsilon\sigma = \sigma$, and $\sigma\sigma' = \sigma$ when $\sigma \in \Sigma^\omega$), $T^+ \stackrel{\text{def}}{=} T \cap \Sigma^+$ and $T^\omega \stackrel{\text{def}}{=} T \cap \Sigma^\omega$ for the selection of the non-empty finite sequences and the infinite sequences of $T \in \mathcal{P}\left(\Sigma^{\star\infty}\right)$, and $T \,;\, T' \stackrel{\text{def}}{=} \{\sigma s \sigma' \mid s \in \Sigma \wedge \sigma s \in T \wedge s\sigma' \in T'\}$ for the merging of two sets of sequences $T \in \mathcal{P}\left(\Sigma^+\right)$ and $T' \in \mathcal{P}\left(\Sigma^{+\infty}\right)$, when a finite sequence in T terminates with the initial state of a sequence in T'.

Given a transition system $\langle \Sigma, \tau \rangle$, 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* $\Upsilon \in \mathcal{P}\left(\Sigma^{+\infty}\right)$ 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 [10]:

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

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

The trace semantics fully describes the behavior of a program. However, reasoning about a particular property of a program does not need to consider all aspects of its behavior. In fact, reasoning is facilitated by the design of a semantics that abstracts away from irrelevant details about program executions. In the next sections, we define our property of interest, absence of data leakage, and use abstract interpretation to systematically derive, by successive abstractions of the trace semantics, a semantics that precisely captures this property.

3 DATA LEAKAGE SEMANTICS

3.1 (Absence of) Data Leakage

A *property*, by extension, is the set of elements having such a property [13, 14]. Properties of programs are properties of their semantics. Thus, properties of programs with trace semantics in $\mathcal{P}\left(\Sigma^{+\infty}\right)$ are sets of sets of traces in $\mathcal{P}\left(\mathcal{P}\left(\Sigma^{+\infty}\right)\right)$. The set of program properties forms a complete

boolean lattice $\langle \mathcal{P}\left(\mathcal{P}\left(\Sigma^{+\infty}\right)\right), \subseteq, \cup, \cap, \emptyset, \mathcal{P}\left(\Sigma^{+\infty}\right)\rangle$ for subset inclusion, that is, logical implication. The strongest property is the standard *collecting semantics* $\Lambda \in \mathcal{P}\left(\mathcal{P}\left(\Sigma^{+\infty}\right)\right)$:

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

Let $(\!|P|\!)$ denote the collecting semantics of a particular program P. Then, a program P satisfies a given property $\mathcal{H} \in \mathcal{P}\left(\mathcal{P}\left(\Sigma^{+\infty}\right)\right)$ if and only if its collecting semantics is a subset of $\mathcal{H}$:

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

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

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 machine learning 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 can define an order on data frame variables based on when they are assigned in σ: $x \triangleleft y$ if and only if the data frame variable x is defined before the data frame variable y in σ. All input data frame variables are assigned before any of the used data frame variables: $\forall i \in \mathrm{I}_P, o \in \mathrm{U}_P : i \triangleleft o$. 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 $[\![P]\!]$:

$$\begin{aligned}
&\textsc{independent}([\![P]\!]) \stackrel{\text{def}}{=} \forall \sigma \in [\![P]\!], i \in \mathrm{I}_P, r \in R_i : \\
&\Big(\forall \bar{v} \in \mathbb{V}^{C_i} : \sigma(i)[r] \neq \bar{v} \\
&\quad \Rightarrow \exists \sigma' \in [\![P]\!] : \sigma'(i)[r] = \bar{v} \wedge \sigma(i) \stackrel{\bar{r}}{=} \sigma'(i) \wedge \sigma(\mathrm{I}_P \setminus \{i\}) = \sigma'(\mathrm{I}_P \setminus \{i\}) \wedge \sigma(\mathrm{U}_P^{\text{test}}) = \sigma'(\mathrm{U}_P^{\text{test}})\Big) \\
&\vee \Big(\forall \bar{v} \in \mathbb{V}^{C_i} : \sigma(i)[r] \neq \bar{v} \\
&\quad \Rightarrow \exists \sigma' \in [\![P]\!] : \sigma'(i)[r] = \bar{v} \wedge \sigma(i) \stackrel{\bar{r}}{=} \sigma'(i) \wedge \sigma(\mathrm{I}_P \setminus \{i\}) = \sigma'(\mathrm{I}_P \setminus \{i\}) \wedge \sigma(\mathrm{U}_P^{\text{train}}) = \sigma'(\mathrm{U}_P^{\text{train}})\Big)
\end{aligned} \tag{5}$$

where R_i and C_i stand for $R_{\sigma(i)}$ and $C_{\sigma(i)}$, respectively, $\sigma(i) \stackrel{\bar{r}}{=} \sigma'(i)$ stands for $\forall r' \in R_i : r' \neq r \Rightarrow \sigma(i)[r'] = \sigma'(i)[r']$, and $\sigma(X) = \sigma'(X)$ stands for $\forall x \in X : \sigma(x) = \sigma'(x)$. Intuitively, 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 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 single row $r \in R_i$) of any input data frame variable $i \in \mathrm{I}_P$ in σ. In terms of input data (non-)usage [28], we can say that training and testing *do not use* the same (portions of the) input data sources. Note that here we generalize the notion of data usage proposed by Urban and Müller [28] 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 follows:

$$\mathcal{I} \stackrel{\text{def}}{=} \left\{[\![P]\!] \in \mathcal{P}\left(\Sigma^{+\infty}\right) \mid \textsc{independent}([\![P]\!])\right\} \tag{6}$$

which is the set of programs (or rather, their semantics) which use independent data for training and testing machine learning models. Thus, from Equation 4, we have:

Theorem 3.1. $P \models \mathcal{I} \Leftrightarrow (\!|P|\!) \subseteq \mathcal{I}$

Proof. The proof follows trivially from Equation 4 and the definitions of $(\!|P|\!)$ (cf. Equation 3) and $\mathcal{I}$ (cf. Equation 6). □

In the rest of this section, we use abstract interpretation to 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 (cf. Equation 6), 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:

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

between sets of sets of traces and sets of relations (i.e., dependencies) between data frame variables indexed at some row. The dependency abstraction $\alpha_{\rightsquigarrow^+} : \mathcal{P}\left(\mathcal{P}\left(\Sigma^{+\infty}\right)\right) \to \mathcal{P}\left((\mathbb{X}\times\mathbb{N})\times(\mathbb{X}\times\mathbb{N})\right)$ is:

$$\alpha_{\rightsquigarrow^+}(S) \overset{\text{def}}{=} \alpha_+ \circ \alpha_{\rightsquigarrow}(S) \tag{8}$$

with $\alpha_{\rightsquigarrow} : \mathcal{P}\left(\mathcal{P}\left(\Sigma^{+\infty}\right)\right) \to \mathcal{P}\left((\mathbb{X}\times\mathbb{N})\times(\mathbb{X}\times\mathbb{N})\right)$ mapping sets of sets of traces to direct dependencies and $\alpha_+ : \mathcal{P}\left((\mathbb{X}\times\mathbb{N})\times(\mathbb{X}\times\mathbb{N})\right) \to \mathcal{P}\left((\mathbb{X}\times\mathbb{N})\times(\mathbb{X}\times\mathbb{N})\right)$ adding transitive dependencies. We define the first element of the abstraction composition in Equation 8, $\alpha_{\rightsquigarrow}$, as follows:

$$\alpha_{\rightsquigarrow}(S) \overset{\text{def}}{=} \left\{ i[r] \rightsquigarrow o[r'] \;\middle|\; \begin{array}{c} i \in \mathbb{X}, r \in \mathbb{N}, o \in \mathbb{X}, r' \in \mathbb{N}, (\forall T \in S: \\ \exists \sigma, \sigma' \in T: (\forall x \in \mathbb{X}: i \triangleleft x \triangleleft o \Rightarrow \sigma(x) = \sigma'(x)) \\ \wedge\ \sigma(i) \overset{\bar{r}}{=} \sigma'(i) \wedge \sigma(o)[r'] \neq \sigma'(o)[r']) \end{array} \right\} \tag{9}$$

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 $i \in \mathbb{X}$ at the row indexed by $r \in \mathbb{N}$ and a data frame variable $o \in \mathbb{X}$ 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 are two traces with the same data frame values except at row r of i that lead to different values at row r' of o. We observe that $\alpha_{\rightsquigarrow}$ tracks direct but not transitive dependencies. Thus the second component of the abstraction composition in Equation 8, α_+, computes transitive dependencies as well:

$$\alpha_+(D) \overset{\text{def}}{=} \{ i[r] \rightsquigarrow o[r'] \mid \exists x \in \mathbb{X}, r'' \in \mathbb{N}: i[r] \rightsquigarrow x[r''] \in D \wedge x[r''] \rightsquigarrow o[r'] \in D \} \tag{10}$$

Note that our definition of dependency abstraction (notably Equation 9) generalizes that of Cousot [11] to multi-dimensional data frame variables and thus tracks dependencies between portions of data frames and used variables. As in [11], 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).

We can now define the *dependency semantics* $\Lambda_{\rightsquigarrow^+} \in \left(\mathcal{P}\left(\mathbb{X}\times\mathbb{N}\right)\times\mathbb{X}\right)$ by abstraction of the collecting semantics Λ:

$$\Lambda_{\rightsquigarrow^+} \overset{\text{def}}{=} \alpha_{\rightsquigarrow^+}(\Lambda) \tag{11}$$

In the rest of the paper, we write $(\!|P|\!)_{\rightsquigarrow^+}$ to denote the dependency semantics of a program P.

The dependency semantics remains sound and complete for reasoning about data leakage:

Theorem 3.2. $P \models \mathcal{I} \Leftrightarrow (\!|P|\!)_{\rightsquigarrow^+} \supseteq \alpha_{\rightsquigarrow^+}(\mathcal{I})$

Proof. Let $P \models \mathcal{I}$. From Theorem 3.1, we have that $(\!|P|\!) \subseteq \mathcal{I}$. Thus, from the Galois connection in Equation 7, we have $\alpha_{\rightsquigarrow^+}((\!|P|\!)) \supseteq \alpha_{\rightsquigarrow^+}(\mathcal{I})$. From the definition of $(\!|P|\!)_{\rightsquigarrow^+}$ (cf. Equation 11), we can then conclude that $(\!|P|\!)_{\rightsquigarrow^+} \supseteq \alpha_{\rightsquigarrow^+}(\mathcal{I})$. □

3.3 Data Leakage Semantics

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 from. 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 [15] into a map for each data frame variable associating with each data frame row index the set of (input) data frame variables (indexed at some row) from which it depends on.

Formally, we define the following Galois connection:

$$\langle \mathcal{P}\left((\mathbb{X}\times\mathbb{N})\times(\mathbb{X}\times\mathbb{N})\right), \supseteq\rangle \underset{\dot{\alpha}_I}{\overset{\dot{\gamma}_I}{\leftrightarrows}} \langle \mathbb{X} \to (\mathbb{N} \to \mathcal{P}(\mathbb{X}\times\mathbb{N})), \dot{\supseteq}\rangle \tag{12}$$

where the abstraction and concretization function are parameterized by a set I of input variables. The abstraction $\dot{\alpha}_I \colon \mathcal{P}\left((\mathbb{X}\times\mathbb{N})\times(\mathbb{X}\times\mathbb{N})\right) \to (\mathbb{X} \to (\mathbb{N} \to \mathcal{P}(\mathbb{X}\times\mathbb{N})))$ is defined as follows:

$$\dot{\alpha}_I(D) \overset{\text{def}}{=} \lambda x \in \mathbb{X} \colon (\lambda r \in \mathbb{N} \colon \{i[r'] \mid i \in I, r' \in \mathbb{N}, i[r'] \rightsquigarrow x[r] \in D\}) \tag{13}$$

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

$$\dot{\Lambda}_I \overset{\text{def}}{=} \dot{\alpha}_I(\Lambda_{\rightsquigarrow^+}) \tag{14}$$

For a particular data frame-manipulating program P the set I is the set of input data frame variables I_P. In the following, we leave I implicit and write $(\!|\dot{P}|\!)$ for the data leakage semantics of P.

The abstraction $\dot{\alpha}_I$ does not lose any information, so we still have both soundness and completeness for reasoning about data leakage:

Theorem 3.3. $P \models \mathcal{I} \Leftrightarrow (\!|\dot{P}|\!) \mathrel{\dot{\supseteq}} \dot{\alpha}(\alpha_{\rightsquigarrow^+}(\mathcal{I}))$

Proof. The proof is analogous to that of Theorem 3.2. Let $P \models \mathcal{I}$. We have that $(\!|P|\!)_{\rightsquigarrow^+} \supseteq \alpha_{\rightsquigarrow^+}(\mathcal{I})$ from Theorem 3.2. From the Galois connection in Equation 12, we have $\dot{\alpha}((\!|P|\!)_{\rightsquigarrow^+}) \mathrel{\dot{\supseteq}} \dot{\alpha}(\alpha_{\rightsquigarrow^+}(\mathcal{I}))$. Thus, from the definition of $(\!|\dot{P}|\!)$ (cf. Equation 14), we can conclude that $(\!|\dot{P}|\!) \mathrel{\dot{\supseteq}} \dot{\alpha}(\alpha_{\rightsquigarrow^+}(\mathcal{I}))$. □

We can now equivalently verify absence of data leakage by checking that data frame variables used for different purposes depend on disjoint (rows of) input data frame variables:

Lemma 3.4.

$$P \models \mathcal{I} \Leftrightarrow \forall o_1 \in \mathrm{U}_P^{\text{train}}, o_2 \in \mathrm{U}_P^{\text{test}} \colon \bigcup_{r_1 \in \mathrm{dom}((\!|\dot{P}|\!)o_1)} (\!|\dot{P}|\!)o_1(r_1) \;\cap \bigcup_{r_2 \in \mathrm{dom}((\!|\dot{P}|\!)o_2)} (\!|\dot{P}|\!)o_2(r_2) = \emptyset$$

3.3.1 Small Data Frame-Manipulating Language. The formal treatment given so far is language independent. In the rest of this section, we provide 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.

We consider a simple sequential language without procedures nor pointers. The only variable data type is the set $\mathbb{D}$ of data frames (cf. Section 2.1). Programs in the language are sequences of statements, which belong to either of the following classes:

(1) **source**: $y = \mathrm{read}(name)$ $\qquad$ $name \in \mathbb{W}$

(2) **select**: $y = x.\text{select}[\bar{r}][C]$ $\quad \bar{r} \in \mathbb{N}^{k \leq R_x}, C \subseteq \text{hdr}(x)$
(3) **merge**: $y = op(x_1, x_2)$ $\quad x_1, x_2 \in \mathbb{X}, op \in \{\text{concat}, \text{join}\}$
(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 $name \in \mathbb{W}$ is a (string) data file name; we slightly abuse notation and write R_x and $\text{hdr}(x)$ for the number of rows and set of labels of the columns of the data frame (value) stored into a variable x. The *source* statements read a data frame from an input file and store it into a variable y; for instance, they represent library functions such as `read_csv`, `read_excel`, etc., in Python pandas. The *select* statements returns a subset data frame y of x, based on an array of row indexes $\bar{r}$ and a set of column labels C; they loosely correspond to library functions such as `iloc`, `loc`, etc., in Python pandas and to select and project operations in relational algebra [24]. 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, which may cause data leakage if used improperly; for example, this represents 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 machine learning model.

Example 3.5 (Motivating Example (Cont.)). The following is a (simplified) version of the notebook execution that lead to a data leakage in our motivating example (cf. Example 1.1) written in our small language:

```
1  data = read("data.csv")
2  X = data.select[][{"X_1", "X_2"}]
3  y = data.select[][{"y"}]
4  X_norm = normalize(X)
5  X_train = X_norm.select[[⌊0.025 * R_X_norm⌋ + 1, ..., R_X_norm]][]
6  X_test = X_norm.select[[0, ..., ⌊0.025 * R_X_norm⌋]][]
7  y_train = y.select[[⌊0.025 * R_y⌋ + 1, ..., R_y]][]
8  y_test = y.select[[0, ..., ⌊0.025 * R_y⌋]][]
9  train(X_train, y_train)
10 test(X_test, y_test)
```

(where, again, we write R_x for the number of rows of the data frame stored in the variable x.)

3.3.2 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., $\text{I}_P \stackrel{\text{def}}{=} i[\![P]\!] = i[\![S_n]\!] \circ \cdots \circ i[\![S_1]\!]\emptyset$, where the semantic function $i[\![S]\!]$ for each statement S in P is defined as follows:

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

Similarly, we define the set of used variables $\mathrm{U}_P \stackrel{\text{def}}{=} u[\![P]\!] = u[\![S_n]\!] \circ \cdots \circ u[\![S_1]\!]\emptyset$, where $u[\![S]\!]$ is:

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

and analogously for $\mathrm{U}_P^{\text{train}} \subseteq \mathrm{U}_P$ (when f = train) and $\mathrm{U}_P^{\text{test}} \subseteq \mathrm{U}_P$ (when f = test).

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

$$\begin{aligned}
s[\![y = \text{read}(name)]\!]m &\stackrel{\text{def}}{=} m\left[y \mapsto \lambda r \in R_{\text{read}()} : \{name[r]\}\right] \\
s[\![y = x.\text{select}[\bar{r}][C]]\!]m &\stackrel{\text{def}}{=} m\left[y \mapsto \lambda r \in R_{x.\text{select}[\bar{r}][C]} : m(x)(\bar{r}[r])\right] \\
s[\![y = \text{concat}(x_1, x_2)]\!]m &\stackrel{\text{def}}{=} m\left[y \mapsto \lambda r \in R_{\text{concat}(x_1,x_2)} : \begin{cases} m(x_1)r & r \leq |\text{dom}(m(x_1))| \\ m(x_2)(r - |\text{dom}(m(x_1))|) & r > |\text{dom}(m(x_1))| \end{cases}\right] \\
s[\![y = \text{join}(x_1, x_2)]\!]m &\stackrel{\text{def}}{=} m\left[y \mapsto \lambda r \in R_{\text{join}(x_1,x_2)} : m(x_1)\overleftarrow{r} \cup m(x_2)\overrightarrow{r}\right] \\
s[\![y = \text{normalize}(x)]\!]m &\stackrel{\text{def}}{=} m\left[y \mapsto \lambda r \in R_{\text{normalize}(x)} : \bigcup_{r' \in \text{dom}(m(x))} m(x)r'\right] \\
s[\![y = \text{other}(x)]\!]m &\stackrel{\text{def}}{=} m\left[y \mapsto \lambda r \in R_{\text{other}(x)} : m(x)r\right] \\
s[\![\text{use}(x)]\!]m &\stackrel{\text{def}}{=} m
\end{aligned} \tag{15}$$

The semantics of *source* statements 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 *select* statements 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 further abstract our 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. 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.

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. We have observed that, in practice, data frame labels are pretty much always strings. Thus, the elements of $\mathbb{C}$ belong to a complete lattice $\langle C, \sqsubseteq_C, \sqcup_C, \sqcap_C, \bot_C, \top_C \rangle$ where $C \stackrel{\text{def}}{=} \mathcal{P}(\mathbb{W}) \cup \{\top_C\}$; $\top_C$ represents a lack of information on which columns a data frame *may* have (abstracting any possible 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 element $\bot_C$ is simply 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 [12] *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 $\mathcal{R} \stackrel{\text{def}}{=} \{[l, u] \mid l \in \mathbb{N}, u \in \mathbb{N} \cup \{\infty\}, l \leq u\} \cup \{\bot_R\}$. 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 (see Mine's PhD thesis [20] for reference, for instance).

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 an isomorphism $\phi_R \colon \mathcal{P}(\mathbb{N}) \to \mathcal{P}(\mathbb{N})$ between $\gamma_R(R)$ (ordered by $\leq$) and $\gamma_R(\mathsf{idx}(R))$ (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 selections of data frame rows in our abstract semantics (cf. Section 4.2).

Example 4.1 (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 $\mathcal{L} \stackrel{\text{def}}{=} \mathbb{W} \times C \times \mathcal{R}$ contains triples of a data file name $file \in \mathbb{W}$, a column over-approximation $C \in C$, and a row over-approximation $R \in \mathcal{R}$. In the following, we write $file_R^C$ for the abstract data frame $\langle file, C, R \rangle \in \mathcal{L}$. The partial order $\sqsubseteq_L$ compares abstract data frames derived from the same data files: $X_R^C \sqsubseteq_L Y_{R'}^{C'} \stackrel{\text{def}}{\Leftrightarrow} X = Y \wedge C \sqsubseteq_C C' \wedge R \sqsubseteq R'$.

We also define a predicate that determines whether two abstract data frames overlap:

$$\text{overlap}(X_R^C, Y_{R'}^{C'}) \stackrel{\text{def}}{\Leftrightarrow} X = Y \wedge C \sqcap_C C' \neq \emptyset \wedge R \sqcap R' \neq \bot_R \tag{16}$$

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

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

Finally, we define a constraining operator $\downarrow^C_R$ that restricts an abstract data frame with given column and row over-approximations: $X^C_R \downarrow^{C'}_{R'} \stackrel{\text{def}}{=} X^{C \sqcap_C C'}_{\phi^{-1}(\mathsf{idx}(R) \sqcap_R R')}$.

Example 4.2 (Abstract Data Frames). Let $file^{\{id,city\}}_{[10,14]}$ be an abstraction of a data frame with columns $\{id, city\}$ and rows $\{10, 12, 13, 14\}$ derived from a data source $file$. The abstract data frame $file^{\{country\}}_{[12,15]}$ does not overlap with $file^{\{id,city\}}_{[10,14]}$, while $file^{\{id\}}_{[12,15]}$ does. Joining $file^{\{id,city\}}_{[10,14]}$ and $file^{\{country\}}_{[12,15]}$ results in the abstract data frame $file^{\{id,city,country\}}_{[10,15]}$. Instead, the meet between $file^{\{id,city\}}_{[10,14]}$ and $file^{\{id\}}_{[12,15]}$ yields the abstract data frame $file^{\{id\}}_{[12,14]}$. Finally, the constraining operation $file^{\{id,city,country\}}_{[10,15]} \downarrow^{\{city\}}_{[1,2]}$ results in $file^{\{city\}}_{[11,12]}$ (since $\phi^{-1}_{[10,15]}(1) = 11$ and $\phi^{-1}_{[10,15]}(2) = 12$).

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 16). The partial order $\sqsubseteq_S$ between canonical sets relies on the partial order between abstract data frames: $S_1 \sqsubseteq_S S_2 \stackrel{\text{def}}{\Leftrightarrow} \forall L_1 \in S_1 \exists L_2 \in S_2 : L_1 \sqsubseteq_L L_2$.

The join ($\sqcup_S$) and meet ($\sqcap_S$) operators perform a set union and set intersection, respectively, followed by a reduction operation to put the resulting set in canonical form:

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

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 the constraining operation $\downarrow^C_R$ by element-wise application: $S \downarrow^C_R \stackrel{\text{def}}{=} \left\{L \downarrow^C_R \mid L \in S\right\}$.

Example 4.3 (Abstract Data Frame Sets). Let us consider the join of two abstract data frame sets $S_1 = \left\{file1^{\{id\}}_{[1,10]}, file2^{\{name\}}_{[0,100]}\right\}$ and $S_2 = \left\{file1^{\{id\}}_{[9,12]}, file3^{\{zip\}}_{[0,100]}\right\}$. Before the reduction, we obtain a non-canonical set: $\left\{file1^{\{id\}}_{[1,10]}, file1^{\{id\}}_{[9,12]}, file2^{\{name\}}_{[0,100]}, file3^{\{zip\}}_{[0,100]}\right\}$. The reduction operation makes the set canonical: $\left\{file1^{\{id\}}_{[1,12]}, file2^{\{name\}}_{[0,100]}, file3^{\{zip\}}_{[0,100]}\right\}$.

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 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 $B \in \mathbb{B} \stackrel{\text{def}}{=} \{\textsc{false}, \textsc{true}\}$, which keeps track of whether the abstract data frames are 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 pair component operators pairwise:

$$\sqsubseteq_A \stackrel{\text{def}}{=} \sqsubseteq_S \times \leq \qquad \sqcup_A \stackrel{\text{def}}{=} \sqcup_S \times \vee \qquad \sqcap_A \stackrel{\text{def}}{=} \sqcap_S \times \wedge$$

where the (total) order $\leq$ in $\mathbb{B}$ is such that $\textsc{false} \leq \textsc{true}$. The bottom element $\bot_A$ is $\langle \emptyset, \textsc{false} \rangle$.

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

$$\gamma(m) \stackrel{\text{def}}{=} \lambda x \in \mathbb{X} \colon\ (\lambda r \in \mathbb{N} \colon \gamma_A(m(x))) \tag{17}$$

where $\gamma_A \colon \mathcal{A} \to \mathcal{P}(\mathbb{X} \times \mathbb{N})$ is defined as follows:

$$\gamma_A(\langle S, B \rangle) \stackrel{\text{def}}{=} \left\{ X[r] \mid X^C_R \in S, r \in \gamma_R(R) \right\} \tag{18}$$

(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 the value of the boolean flag $B \in \mathbb{B}$ nor the column abstraction $C \in \mathcal{C}$. These are uniquely needed by our abstract semantics that we define below.

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:

$$\begin{aligned}
a[\![y = \text{read}(name)]\!]m &\stackrel{\text{def}}{=} m \left[y \mapsto \langle \left\{ name^{\top_C}_{[0,\infty]} \right\}, \textsc{false} \rangle \right] \\
a[\![y = x.\text{select}[\bar{r}][C]]\!]m &\stackrel{\text{def}}{=} m \left[y \mapsto \begin{cases} \langle m_s(x) \downarrow^C_{[\min(\bar{r}), \max(\bar{r})]}, m_b(x) \rangle & \neg m_b(x) \\ \langle m_s(x), m_b(x) \rangle & \text{otherwise} \end{cases} \right] \\
a[\![y = \text{op}(x_1, x_2)]\!]m &\stackrel{\text{def}}{=} m \left[y \mapsto \langle m_s(x_1) \sqcup_S m_s(x_2), m_b(x_1) \vee m_b(x_2) \rangle \right] \\
a[\![y = \text{normalize}(x)]\!]m &\stackrel{\text{def}}{=} m \left[y \mapsto \langle m_s(x), \textsc{true} \rangle \right] \\
a[\![y = \text{other}(x)]\!]m &\stackrel{\text{def}}{=} m \left[y \mapsto \langle m_s(x), m_b(x) \rangle \right] \\
a[\![\text{use}(x)]\!]m &\stackrel{\text{def}}{=} m
\end{aligned} \tag{19}$$

The abstract semantics of *source* statements 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 *select* 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; 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 and assigns the result to the data frame variable y. 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 $\textsc{true}$, while *other* functions leave the taint flag $m_b(x)$ unchanged. Finally, the abstract semantics of *use* statements leave the abstract dependencies map unchanged.

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

Theorem 4.4. $P \models \mathcal{I} \Leftarrow \gamma((\!|\dot{P}|\!)^\natural) \mathrel{\dot{\ni}} \dot{\alpha}(\alpha_{\rightsquigarrow^+}(\mathcal{I}))$

Proof (Sketch). The proof follows from the definition of abstract data leakage semantics $(\!|\dot{P}|\!)^\natural$ and that of the concretization function γ (cf. Equation 17), 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 (cf. Equation 19). □

Similarly, we have the sound but not complete counterpart of Lemma 3.4 for practically checking absence of data leakage:

Lemma 4.5.

$$P \models \mathcal{I} \Leftarrow \forall o_1 \in \mathrm{U}_P^{\text{train}}, o_2 \in \mathrm{U}_P^{\text{test}} : \bigcup_{r_1 \in \mathrm{dom}(\gamma(\dot{(\![P]\!)}^\natural) o_1)} \gamma(\dot{(\![P]\!)}^\natural) o_1(r_1) \cap \bigcup_{r_2 \in \mathrm{dom}(\gamma(\dot{(\![P]\!)}^\natural) o_2)} \gamma(\dot{(\![P]\!)}^\natural) o_2(r_2) = \emptyset$$

$$\Leftrightarrow \forall o_1 \in \mathrm{U}_P^{\text{train}}, o_2 \in \mathrm{U}_P^{\text{test}} : \gamma_A(\dot{(\![P]\!)}^\natural o_1) \cap \gamma_A(\dot{(\![P]\!)}^\natural o_2)$$

$$\Leftrightarrow \forall o_1 \in \mathrm{U}_P^{\text{train}}, o_2 \in \mathrm{U}_P^{\text{test}} :$$

$$\forall X_R^C \in \dot{(\![P]\!)}_s^\natural o_1, Y_{R'}^{C'} \in \dot{(\![P]\!)}_s^\natural o_2 : \neg\mathrm{overlap}(X_R^C, Y_{R'}^{C'}) \wedge \left(X = Y \Rightarrow \neg\dot{(\![P]\!)}_b^\natural o_1 \wedge \neg\dot{(\![P]\!)}_b^\natural o_2\right)$$

Example 4.6 (Motivating Example (Continue)). The data leakage analysis of our motivating example (written in our small language, cf. Example 3.5) is the following:

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

$$a[\![\texttt{X = data.select[][{"X\_1", "X\_2"}]}]\!]m_1 = \left(m_2 \overset{\text{def}}{=} m_1 \left[X \mapsto \langle \left\{ data.csv_{[0,\infty]}^{\{\text{“}X\_1\text{”},\text{“}X\_2\text{”}\}} \right\}, \textsc{false} \rangle\right]\right)$$

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

$$a[\![\texttt{X\_norm = normalize(X)}]\!]m_3 = \left(m_4 \overset{\text{def}}{=} m_3 \left[X\_\text{norm} \mapsto \langle \left\{ data.csv_{[0,\infty]}^{\{\text{“}X\_1\text{”},\text{“}X\_2\text{”}\}} \right\}, \textsc{true} \rangle\right]\right)$$

$$a[\![\texttt{X\_train = X\_norm.select[[⌊0.025 * } R_{\texttt{X\_norm}}\texttt{⌋ + 1, ..., } R_{\texttt{X\_norm}}\texttt{]][]}]\!]m_4 =$$

$$\left(m_5 \overset{\text{def}}{=} m_4 \left[X\_\text{train} \mapsto \langle \left\{ data.csv_{[\lfloor 0.025 * R_{X\_\text{norm}} \rfloor + 1, R_{X\_\text{norm}}]}^{\{\text{“}X\_1\text{”},\text{“}X\_2\text{”}\}} \right\}, \textsc{true} \rangle\right]\right)$$

$$a[\![\texttt{X\_test = X\_norm.select[[0, ..., ⌊0.025 * } R_{\texttt{X\_norm}}\texttt{⌋]][]}]\!]m_5 =$$

$$\left(m_6 \overset{\text{def}}{=} m_5 \left[X\_\text{test} \mapsto \langle \left\{ data.csv_{[0, \lfloor 0.025 * R_{X\_\text{norm}} \rfloor]}^{\{\text{“}X\_1\text{”},\text{“}X\_2\text{”}\}} \right\}, \textsc{true} \rangle\right]\right)$$

$$a[\![\texttt{y\_train = y.select[[⌊0.025 * } R_y\texttt{⌋ + 1, ..., } R_y\texttt{]][]}]\!]m_6 =$$

$$\left(m_7 \overset{\text{def}}{=} m_6 \left[y\_\text{train} \mapsto \langle \left\{ data.csv_{[\lfloor 0.025 * R_y \rfloor + 1, R_y]}^{\{\text{“}y\text{”}\}} \right\}, \textsc{false} \rangle\right]\right)$$

$$a[\![\texttt{y\_test = y.select[[0, ..., ⌊0.025 * } R_y\texttt{⌋]][]}]\!]m_7 =$$

$$\left(m_8 \overset{\text{def}}{=} m_7 \left[y\_\text{test} \mapsto \langle \left\{ data.csv_{[0, \lfloor 0.025 * R_y \rfloor]}^{\{\text{“}y\text{”}\}} \right\}, \textsc{false} \rangle\right]\right)$$

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

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

Note that, at the end of the analysis, $X\_\text{train} \in \mathrm{U}^{\text{train}}$ and $X\_\text{test} \in \mathrm{U}^{\text{test}}$ depend on non-overlapping but *tainted* abstract data frames derived from the same input data file *data.csv*. Thus, the absence of data leakage check from Lemma 4.5 (rightfully) fails.

5 IMPLEMENTATION

We integrate our analysis based on our approach described in Section 4.2 into NBLyzer [26], an open source static analysis framework for data science notebooks. We implement our analysis for a subset of Python 3, focusing on language constructs commonly used in data science notebooks operating on Python pandas [4] data frames. In the following sections we describe the NBLyzer

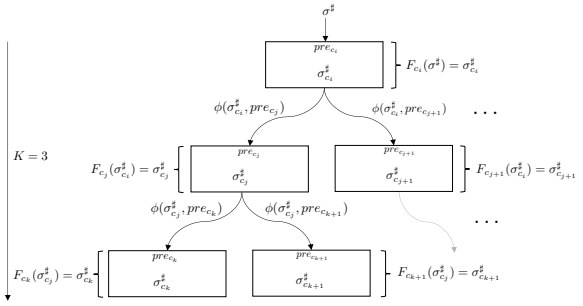


Fig. 1. Inter-cell analysis

framework and outline additional considerations required to integrate our analysis into NBLyzer. In this section we reuse notation from [26] to better explain our integration steps.

5.1 Framework Overview

NBLyzer is designed specifically to adapt to the unique data science notebook development and execution flexibility. 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*. We define the ϕ-condition used by our data leakage analysis in the next Section 5.2.

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 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 the next Section 5.2.

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 n^{th} 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.

5.2 A Data Leakage Detector

Next, we describe our implementation of a data leakage detector based on the abstract semantics of Section 4.2. We outline below additional tasks required to integrate the data leakage detector into NBLyzer.

5.2.1 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 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.

5.2.2 Knowledge Base. We assume a knowledge bases KB_{source}, KB_{norm} and KB_{test} KB_{train} which holds functions that act as a source, introduce data leaks, and be used for testing and training, respectively. Since several data science libraries exist, it is difficult to infer this knowledge automatically. Practically, we instantiate the knowledge based of common library calls in Pandas, Scikit-learn [6] etc.

5.2.3 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 cite before processing the cell. In the case the definition does not exist in a predecessor cell, we treat the function as a undefined function.

5.2.4 User Interface for Data Leakage Analysis. We extend the NBLyzer Microsoft Visual Code (VS Code) frontend to incorporate our data leakage analysis. Here a NBLyzer server process is spawned for every opened notebook. Events e.g., code changes, cell execution, cell creation, cell removal are sent to the server which can trigger an analysis. The analysis result is sent back to VS Code containing error information (error line, column number and cell sequence). This information is reported to the user via the VS Code diagnostics API commonly used by linters, compilers etc. A snapshot of the VS Code extension user interface is shown in Figure 2. Here our analysis highlights the variables `X_selected_train` `X_selected_test` and warns the user that they contribute to a potential data leakage when used together to train and test.

6 EXPERIMENTAL EVALUATION

In this section, we evaluate our implementation of a data leakage detector based on the approach presented in this paper. We evaluate our implementation in NBLyzer on real-world competition data science notebook benchmarks to determine its effectiveness and applicability. We consider the following research questions:

- RQ1: Can our analysis be employed in an interactive, low-latency notebook environment?
- RQ2: Are data leakages present in notebooks and can our analyzer detect data leakages with acceptable precision?

6.1 Experimental Setup

6.1.1 Environment. All experiments were performed on an Apple M1 CPU @ 3.2 GHz with 16 GB RAM running a macOS 12.6.1 operating system. Python 3.10.6 was used to execute NBLyzer running our data leakage static analysis.

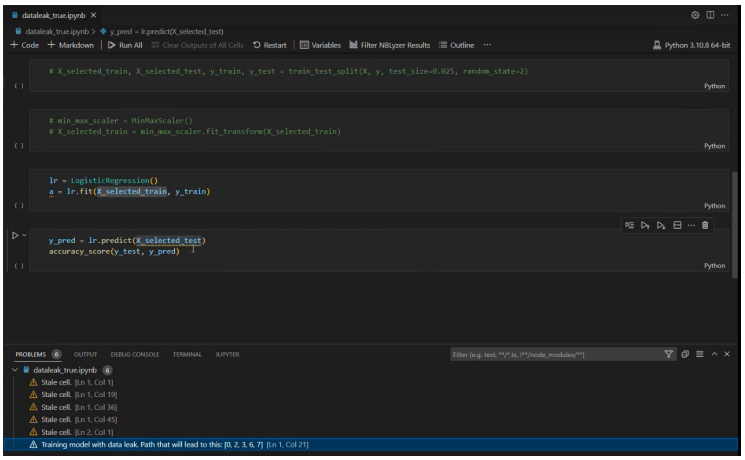


Fig. 2. Data Leakage Detector in VS Code Producing a Data Leakage Warning

Table 1. 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 instructions (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
Non-parsing cells (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

6.1.2 Benchmarks. We use a data science notebook benchmark suite consisting of 4 Kaggle [2] competitions that has previously been used to evaluate data science static analyzers [21, 26]. We evaluate the performance of our data leakage analysis on 2088 of these notebooks, and ignore notebooks which could not be digested by our analysis (i.e., syntax errors, JSON decoding errors etc.). The benchmark characteristics are summarized in Table 1. All notebooks are written for success in a non-trivial data science competition task and can be assume to closely represent code of professional data scientists. On average the notebooks in the benchmark suite have 24 cells, where each cell on average has 9 lines of code. In addition, on average branching instructions appear in 33% of cells. Each notebook has on average 3 functions and 0.1 classes defined.

6.1.3 Use Case. Our use case is a data leakage detector for notebooks in an Integrated Development Environment (IDE) as described in Section 5.2.4. 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 user can configure how many cell executions in the future the analysis looks at by configuring the K bound. The analyzer should identify data leakages with a *soft* analysis deadline of 1 second in accordance to the RAIL performance model [5].

6.1.4 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. An analysis has a parameter K that defines the depth limit of each execution branch of an analysis. Every analysis returns (1) a

set of results i.e., sequences of cells leading to a detected data leakage and (2) performance statistics i.e., run-time, ϕ-rate, etc. We define a *timeout* of 500 seconds.

6.2 Performance Evaluation

6.2.1 Terminating Analyses. We first evaluate the performance for 2088 notebooks in Figure 3 (note this is in log scale) for $K = \infty$. Overall from these notebooks we perform an analysis on 7391 valid executions. We note (as shown in Figure 5c) that 20 executions on distinct notebooks timeout. In contrast we show the execution times for $K = 4$ in Figure 4 (note this is in log scale). In this case, we incorporate all notebook executions as there are no timeouts.

Overall, we see that irrespective of K the vast majority of executions finish in under a minute. For $K = \infty$, 99.6% (all except 33 executions) finish in less than a second and for $K = 4$, 99.9% (all except 5 executions) finish in less than a second. For $K = \infty$, 20 out the of 33 time out and out of the 13 that didn't time out, the average execution-time is 0.18 seconds. When only executions that propagate are considered, the average execution time increases to 6.4 seconds. We note this is mainly due to 4 outliers executions that take several minutes to complete. From these, the maximum execution-time is 414 seconds. For the $K = 4$ executions the average execution was 0.003 seconds with a maximum execution time of 3.9 seconds. If we consider only executions where propagation occurs, then the average execution-time increases to 0.07 seconds. Thus bounding K to $K = 4$ from $K = \infty$ improves the overall average execution time significantly (by 283×) and reduces the number of executions that is over a second by 83%. $K = \infty$ and $K = 4$ are edge cases for K. In Figure 5a we plot the average execution time for various K values (not including any that timeout).

To better understand these results, we investigate the characteristics of the NBLyzer inter-cell fixpoint algorithm for our analysis. Firstly, we measure the propagation rate i.e., ϕ-rate on all executions. We find ϕ is true 1% of the time. When we take into account executions that perform propagation, the ϕ-rate increases to 11% with the largest rates being 87% and the smallest rates being $> 1\%$. This indicates that the execution tree width tends to be narrow on most notebooks, hence explaining one of the reasons for the observed execution-times. We note, varying K does not significantly change the ϕ-rate. Secondly, we measure the average execution-time for different K values in Figure 5a (not including the 19 that timeout) and compare this to the number of fixpoint subsumptions occurring at that given K value. Figure 5 shows that most notebooks reach a global fixpoint at $K = 10$ and $K = \infty$ results in a global fixpoint on all measured notebook executions. Moreover, we observe that despite the fact that the data leakage abstract domain is unbound, we do not encounter any cases where fixpoint subsumption does not occur i.e., a situation requiring widening. This is affirmed by Figure 5 which shows an increase in fixpoint subsumption as we increase K. Again this again explains the executions times as the execution tree is not only typically narrow, but typically shallow.

6.2.2 Timeout Analyses. Next we analyze the 20 notebook executions that cause a timeout at $K = \infty$. In Figure 5c we show the number of notebooks that timeout for every K value. Bounding K to 100 lowers the number of timeouts by one as for $K = 25$ the number remains the same. At $K = 14$ and $K = 12$, 6 less notebooks timeout. At $K = 10$, 5 less timeout. At $K = 8$, 4 less timeout and at $K = 6$, only 2 notebook time out. Upon investigating the notebook code of each timeout, we find they have the following commonalities (1) large notebooks: on average they comprise of 45 cells. (2) a very high inter-cell connectivity: hence a high ϕ-rate averaging 51%.

6.2.3 Usability and K Bounds. We also note that when using NBLyzer in interactive mode ie., in IDE analysis, our experience with users indicates that they naturally impose a K bound regardless of any performance improvements. The reason for this is due to information overload. Recall that notebook cells can be executed on demand, thus a user typically wants a warning anywhere from

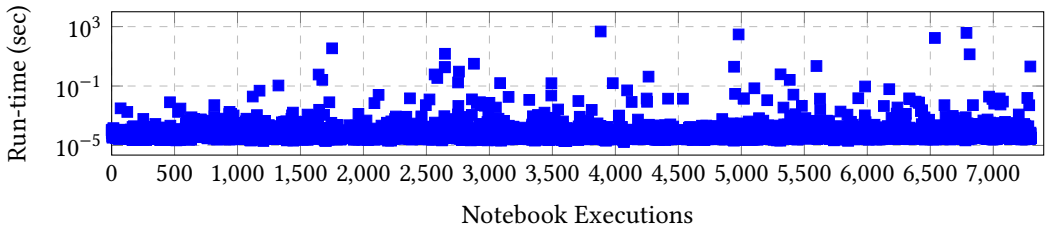


Fig. 3. Run-times for $K = \infty$

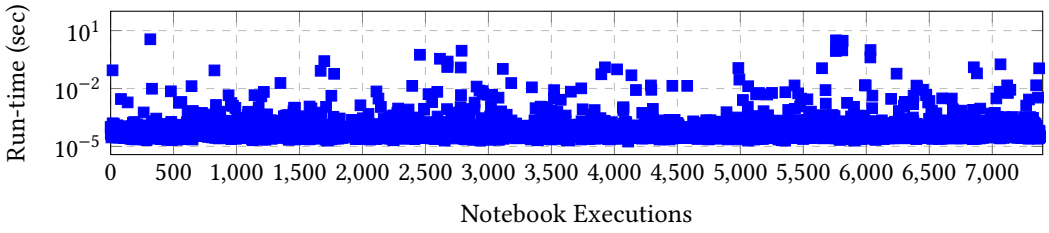
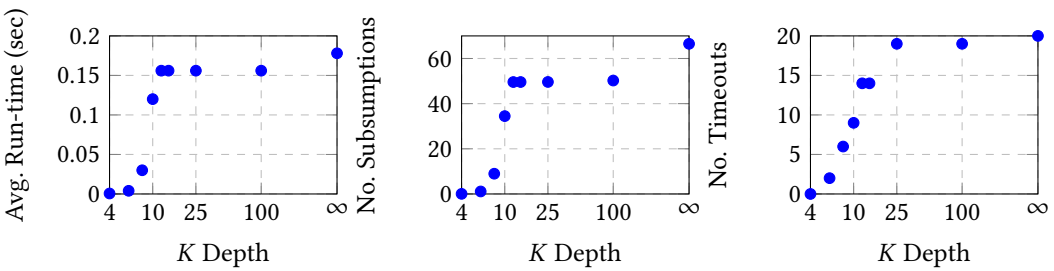


Fig. 4. Run-times for $K = 4$

4-10 cells in the future. Only in an offline/batch mode analysis do we realistically want to analyze till fixpoint. However in these use cases, time constraints are much more relaxed and if used in a CI/CD pipeline typical run-time constraints are in the order of approx. 30 minutes [16].

Overall, for RQ1, we can conclude that our analysis largely performs within the parameters to deploy it in an interactive notebook IDE. However, when $K = \infty$, 30 executions exceed the 1 second deadline. However, we show if we bound K these can be eventually eliminated. For interactive notebooks its rare that users want large traces that predict executions over 10 executions ahead. As we shown in the next subsection, we did not find data leakages with a trace larger than 8.



(a) Avg. run-times for various K (b) Avg. subsumptions for various K (c) Timeouts for various K

Fig. 5. Performance characteristics for varying K

6.3 Precision Evaluation

We evaluate the errors reported from our analysis as well as the the analysis precision. In particular we note the length of each error path and its impact on precision for bounded K values. Our analysis reported a total of 32 data leakage errors observed in 20 notebooks. Out of these, two notebooks

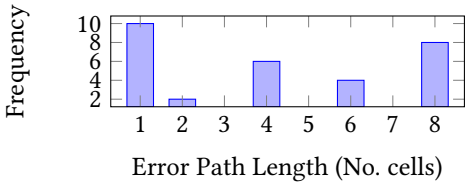


Fig. 6. Error path length frequency

reported 1 false positive execution each (for a total of 2). Thus we exhibit a precision of 94%. Our investigation into the reasons for the false positives revealed that they occurred for the same reason, namely, the case where different objects have the same function name. We believe that this can be overcome by introducing object sensitivity in our analysis.

Of the true positives, 60% were due to overlapping rows and 40% due to incorrect sequences of calls to tainted functions. The data leakages cell execution traces varied between $1-8$ cells in length. We summarise these findings in the histogram in Figure 6. These results seems to suggest that a $K = 8$ appears to be enough to detect all the encountered data leakages and $K = 4$ for the majority of them. Combined with the performance results (Figure 5a), timeout results (Figure 5c) it indicates that $K = 8$ may be a good compromise between performance and detection.

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.

Overall, with regards to RQ2, we conclude that data leakages indeed exist even in high-quality competition notebooks and that our analysis can be detected what we assume is a majority of them, with high precision.

7 RELATED WORK

Related Abstract Semantics. Our dependency abstract semantics generalizes that of Cousot [11] to multi-dimensional data frame variables. The implication of this generalization is that we also provide a multi-dimension generalization for all derivable semantics (cf. Section 7 in [11]).

Static Analysis for Data Science. Static analysis for data science is an emerging area in the program analysis community. A comprehensive state of the art is outlined in [27]. Some notable static analyses for data science scripts include an analyses for ensuring correct shape dimensions in TensorFlow [7] programs [18], an analysis for constraining inputs based on program constraints [28], provenance analysis [21]. In addition, static analysis have been proposed for data science notebooks [19, 26]. Due to notebooks having a unique execution semantics, static analysis for scripts cannot be directly employed on notebooks without naively assuming linear execution [1] or batch execution. In [26] a framework is proposed that can support a wide class of analyses for notebooks. We integrate our analysis in this framework (See Section 5) to support both notebooks and scripts. In regard to data leakage detection, our work is most similar to the demonstrative

analysis presented in [25, 26]. Our analysis is a superset of this analysis, corrects several sources of unsoundness, provides more precise abstract domain operators and provides a formally and rigorous semantic underpinning. Moreover, the implementation of our analysis is able to detect real data leakages which we demonstrate in this paper. Furthermore, to the best of our knowledge, we are the first to propose a formal definition of data leakage semantics which has been systematically derived and proven using the theory of abstract interpretation.

Data Leakage Detection and Avoidance. Several techniques exist to avoid and discover data leakages in data science code. Traditionally manual techniques [17] are employed to inspect data when a data leak is suspected. Other popular techniques employed are the use of data science piplelines [8] that stage the phases of sourcing, cleaning, splitting, normalization, and training to avoid performing a normalization step before splitting. This, also requires a manual effort and code modifications, and is not widely used among the millions of data scientists, especially in notebook environments. Data provenance and lineage techniques [21] can also aid in the discovery of data leakages by building a dependency graph. However, this tool is appropriate for a post mortem analysis and cannot detect insidious instances of data leakages. Tools such as [3, 9] are used to perform dynamic instrumentation to detect data leakages at execution-time with the cost of runtime-overhead and boilerplate code and some post mortem manual inspection. One interesting direction is to use such dynamic techniques to prune false positives.

8 CONCLUSION

We have presented a method for detecting data leakages statically. Our approach is comprehensive in that we provide a formal and rigorous derivation from a base trace semantics, via successive intermediate semantics to a final sound and computable static analysis definition. We then implement our analyzer in the NBLyzer static analysis framework including several operators for handling notebook out-of-order execution semantics. Finally, we have demonstrated that our implementation performs within the constraints required for an interactive notebook environment and is able to detect real data leakages in high quality competition notebooks. To the best of our knowledge, we are the first to formally and systematically derive, using abstract interpretation, a data leakage static analyzer for data science notebooks.

ACKNOWLEDGMENTS

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

REFERENCES

[1] 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.
[2] 2022. Kaggle. http://kaggle.com. Accessed: 2022-09-30.
[3] 2022. leak-detect. https://github.com/abhayspawar/leak-detect. Accessed: 2022-03-08.
[4] 2022. Pandas Library. https://pandas.pydata.org. Accessed: 2022-09-30.
[5] 2022. RAIL model. https://web.dev/rail/. Accessed: 2022-09-30.
[6] 2022. Scikit-learn Library. https://scikit-learn.org. Accessed: 2022-09-30.
[7] 2022. TensorFlow. https://www.tensorflow.org. Accessed: 2022-09-30.
[8] Sumon Biswas, Mohammad Wardat, and Hridesh Rajan. 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* (Pittsburgh, PA, USA).
[9] Shir Chorev, Philip Tannor, Dan Ben Israel, Noam Bressler, Itay Gabbay, Nir Hutnik, Jonatan Liberman, Matan Perlmutter, Yurii Romanyshyn, and Lior Rokach. 2022. Deepchecks: A Library for Testing and Validating Machine Learning Models and Data. https://doi.org/10.48550/ARXIV.2203.08491

[10] Patrick Cousot. 2002. Constructive Design of a Hierarchy of Semantics of a Transition System by Abstract Interpretation. *Electronic Notes in Theoretical Computer Science* 277, 1-2 (2002), 47–103.

[11] Patrick Cousot. 2019. Abstract Semantic Dependency. In *Proc. SAS*. 389–410.

[12] Patrick Cousot and Radhia Cousot. 1976. Static Determination of Dynamic Properties of Programs. In *Proceedings of the Second International Symposium on Programming*. 106–130.

[13] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. POPL*. 238–252.

[14] Patrick Cousot and Radhia Cousot. 1979. Systematic Design of Program Analysis Frameworks. In *Proc. POPL*. 269–282.

[15] Patrick Cousot and Radhia Cousot. 1994. Higher Order Abstract Interpretation (and Application to Comportment Analysis Generalizing Strictness, Termination, Projection, and PER Analysis. In *ICCL*. 95–112.

[16] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. 2019. Scaling Static Analyses at Facebook. *Commun. ACM* 62, 8 (July 2019), 62–70.

[17] Shachar Kaufman, Saharon Rosset, and Claudia Perlich. 2011. Leakage in Data Mining: Formulation, Detection, and Avoidance *(KDD '11)*. Association for Computing Machinery, New York, NY, USA, 556–563. https://doi.org/10.1145/2020408.2020496

[18] Sifis Lagouvardos, Julian Dolby, Neville Grech, Anastasios Antoniadis, and Yannis Smaragdakis. 2020. Static Analysis of Shape in TensorFlow Programs. In *34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference) (LIPIcs, Vol. 166)*, Robert Hirschfeld and Tobias Pape (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 15:1–15:29.

[19] Stephen Macke, Hongpu Gong, Doris Jung Lin Lee, Andrew Head, Doris Xin, and Aditya G. Parameswaran. 2020. Fine-Grained Lineage for Safer Notebook Interactions. *CoRR* abs/2012.06981 (2020). arXiv:2012.06981 https://arxiv.org/abs/2012.06981

[20] Antoine Miné. 2004. *Weakly Relational Numerical Abstract Domains*. Ph. D. Dissertation. École Polytechnique, Palaiseau, France. https://tel.archives-ouvertes.fr/tel-00136630

[21] Mohammad Hossein Namaki, Avrilia Floratou, Fotis Psallidas, Subru Krishnan, Ashvin Agrawal, Yinghui Wu, Yiwen Zhu, and Markus Weimer. 2020. Vamsa: Automated Provenance Tracking in Data Science Scripts. In *Proc. KDD*. 1542–1551.

[22] Panagiotis Papadimitriou and Hector Garcia-Molina. 2009. A Model for Data Leakage Detection. In *Proc. ICDE*. 1307–1310.

[23] Jeffrey Perkel. 2018. Why Jupyter is data scientists' computational notebook of choice. *Nature* 563 (11 2018), 145–146. https://doi.org/10.1038/d41586-018-07196-1

[24] Devin Petersohn, Stephen Macke, Doris Xin, William Ma, Doris Lee, Xiangxi Mo, Joseph E. Gonzalez, Joseph M. Hellerstein, Anthony D. Joseph, and Aditya Parameswaran. 2020. Towards Scalable Dataframe Systems. *Proc. VLDB Endow.* 13, 12 (jul 2020), 2033–2046. https://doi.org/10.14778/3407790.3407807

[25] Pavle Subotic, Uros Bojanic, and Milan Stojic. 2022. Statically detecting data leakages in data science code. In *SOAP '22: 11th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis, San Diego, CA, USA, 14 June 2022*, Laure Gonnord and Laura Titolo (Eds.). ACM, 16–22. https://doi.org/10.1145/3520313.3534657

[26] Pavle Subotić, Lazar Milikić, and Milan Stojić. 2022. A Static analysis Framework for Data Science Notebooks. In *ICSE'22: The 44th International Conference on Software Engineering* (Pittsburgh, PA, USA).

[27] Caterina Urban. 2019. Static Analysis of Data Science Software. In *Static Analysis - 26th International Symposium, SAS 2019, Porto, Portugal, October 8-11, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11822)*, Bor-Yuh Evan Chang (Ed.). Springer, 17–23. https://doi.org/10.1007/978-3-030-32304-2_2

[28] Caterina Urban and Peter Müller. 2018. An Abstract Interpretation Framework for Input Data Usage. In *Proc. ESOP*. 683–710.