Statically Detecting Data Leakages in Data Science Code

Pavle Subotić
pavlesubotic@microsoft.com
Microsoft
Serbia

Uroš Bojanić
a-ubojanic@microsoft.com
Microsoft
Serbia

Milan Stojić
milan.stojic@microsoft.com
Microsoft
Serbia

Abstract

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 runtime methods. In this paper, we develop a static data leakage analysis to detect several instances of data leakages during development time. Our analysis is constructed to be lightweight so that it can be performed within interactive data science notebooks. We have integrated our analysis into the NBLyzer static analyzer framework and show its utility on real world benchmarks. To the best of our knowledge, we propose the first static detection of data science data leakages.

CCS Concepts: • **Theory of computation** → **Program analysis**; • **Software and its engineering** → **Integrated and visual development environments**.

Keywords: static analysis, data science, data leakage

ACM Reference Format:
Pavle Subotić, Uroš Bojanić, and Milan Stojić. 2022. Statically Detecting Data Leakages in Data Science Code. In *Proceedings of the 11th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP '22), June 14, 2022, San Diego, CA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3520313.3534657

1 Introduction

Data science software is increasingly ubiquitous for everyday applications. Consequently, the correctness of data science software is vital considering its impact on society. While a plethora of state-of-the-art static analyzers exist that can target various programming bugs, data science programs contain domain specific bugs that are not supported by standard static analyzers. A notable data science specific bug is a *data leakage* [6]. Data leakages arise when external data (e.g., from the test data set) is used to train a model. Data leakages can be particularly insidious. For example, normalized input data may be split into training and testing data. However, the process of normalization transforms the data based on the entire data set (e.g., taking the average) and thus any splitting of the data cannot guarantee independence and may result in a seemingly accurate model that does not perform well in practice.

Example 1.1 (Motivating Example)**.** Consider the program below. In the program, a CSV file is read into a data frame (line 8). From the data frame columns are selected to represent x and y coordinates (lines 14, 17). Each coordinate is normalized (lines 15) and split by selecting ranges of rows from the data frame (lines 20, 21, 23, 24). The training data is then used to train (line 30) the model and it is then tested (line 33) to evaluate its accuracy (line 36).

```
 1 import pandas as pd
 2 import numpy as np
 3 from sklearn.preprocessing import MinMaxScaler
 4 from sklearn.metrics import accuracy_score
 5 from sklearn.naive_bayes import GaussianNB
 6
 7 # load dataframe
 8 df = np.genfromtxt("data.csv", delimiter=',', dtype=None)
 9
10 # preprocessing tools
11 min_max_scaler = MinMaxScaler()
12
13 # feature/tabel selection
14 X = df[['col1', 'col2']]
15 X_selected = min_max_scaler.fit_transform(X)
16
17 y = df['col3']
18
19 # train/test split
20 X_train = X_selected.iloc[:3]
21 y_train = y.iloc[:3]
22
23 X_test = X_selected.iloc[4:6]
24 y_test = y.iloc[4:6]
25
26 # initiate model
27 clf = GaussianNB()
28
29 # train model
30 clf.fit(X_train, y_train)
31
32 # predict labels
33 pred = clf.predict(X_test)
34
35 # measure score
36 acc = accuracy_score(y_test, pred)
37 print("accuracy: {}".format(acc))
```

The code above highlights the ease of inducing a data leakage. Even though the training and testing data is seemingly

SOAP '22, June 14, 2022, San Diego, CA, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9274-7/22/06…$15.00
https://doi.org/10.1145/3520313.3534657

disjoint, the fact that the normalization function is called *before* splitting means that a data leakage is possible.

Typically, a data scientist will detect data leakage postmortem [3]. Given a suspicious result, data analyses methods are used to identify data dependencies. These methods are powerful to explore the relationships between data, however many data leakages can be detected in the code *statically* and thus a static analysis be can employed as an early detection mechanism to complement postmortem data analyses techniques.

In this paper we propose a static analysis for detecting such data leakages. We present an abstract domain that tracks the origin of data frame cells to determines if two data frames originate from overlapping or tainted data sources (i.e., it has been processed by an external library function that could change the data in a way that can introduce a data leak). When a variable is an argument to a function that trains or tests a model, we assert that the variable is *disjoint* and untainted. For instance, in the example above, our analysis would 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 have implemented our analysis in the NBLyzer [12] static analysis framework for data science notebooks and evaluated its performance using 2211 real-world competition data science notebooks. The evaluation shows that our analysis performs within the time required by the use case for the vast majority of notebooks. We summarize our contributions below:

- We define a novel static analysis which detects data leakages in data science code
- We implement our analysis in the NBLyzer static analysis framework for data science notebooks
- We evaluate our analysis on real-world data science code

2 Background

2.1 Abstract State Computation

Abstract Interpretation executes the program in a soundly over-approximating semantics that ensure termination at the price of false positives. Given a sequence of statements $\overrightarrow{st}$, we construct a control flow graph (CFG), a directed graph that encodes the control flow of the statements. We define a CFG as $\langle L, E \rangle$ where an edge $(l, st, l') \in E$ reflects the semantics of statement st associated with the CFG edge from locations l to l'. The set of variables in all the statements is denoted by V and the set of non-variable symbols by S. We assume the variables in the statements are organized in single static assignment (SSA) form [11].

A sound over-approximation $\sigma^\sharp$ of a state σ is computed by iteratively solving the semantic fixed point equation $\sigma^\sharp = \sigma_0^\sharp \sqcup [\![\overrightarrow{st}]\!]^\sharp(\sigma^\sharp)$ using the abstract semantics $[\![\overrightarrow{st}]\!]^\sharp$ for a sequence of statements $\overrightarrow{st}$ and the initial abstract state $(\sigma_0^\sharp)$.

2.2 Data Frames

A data frame is a table or a two-dimensional array-like structure in which each column contains values of one variable and each row contains one set of values from each column. Data frames have non-empty column names, with unique row names/indexes. The data stored in a data frame can be of numeric, factor or character type. Each column should contain the same number of data items. Thus semantically, a data frame has four components, namely, a data frame variable or label y, a set of rows $\bar{r} \subseteq R$, a set of columns labels $\bar{c} \subseteq C$ and the contents of the data frame. Since we are not concerned with the contents of a data frame in this work, we ignore this component. We denote a data frame as $y_{\bar{r}}^{\bar{c}}$.

3 Data Leakage Analysis

3.1 Simplified Syntax

We define a simplified syntax for succinctly describing our analysis abstract semantics for a data frame inspired language. We define five classes of statements that are used to define the abstract semantics for our analysis.

1. **source:**
$$y = \text{read}(name)$$
where $name \in S$
2. **select-project**:
$$y = x.sel[\bar{r}][\bar{c}]$$
where $\bar{r} \subseteq R, \bar{c} \subseteq C$
3. **operations**:
$$y = op(x_1, x_2)$$
where $op \in \{union, merge, diff\}$ and $x_1, x_2 \in V$
4. **functions**:
$$y = f(\bar{x})$$
where $f \in \{norm, other\}$ and $\bar{x} \subseteq V$
5. **sink**:
$$f(\bar{x})$$
where $f \in \{test, train\}$ and $\bar{x} \subseteq V$

The **source** statement reads from a source (indicated by a label *name*) and stores a data frame into a variable y. For example, this statement corresponds to pandas statements such as `read_csv`, `read_excel`, `read__fwf` etc.

The **select-project** statement, creates a data frame in variable y that holds a subset of data in x, based on a set of rows $\bar{r}$ and columns $\bar{c}$. It loosely corresponds to statements such as `iloc, loc etc.` commonly found in libraries like pandas and to select and project operations in relational algebra [9].

The **operations** class of statements transform data frames. Here we assume binary operations. Depending on the function, these have different semantics. The *union* statement performs a set union of two data frames. This maps to pandas statements such as `concat etc`. The *merge* statement combines two data frames by element. This maps to pandas statements such as `join` etc. The *diff* performs a set difference of two data frames. This maps to pandas statements such as `subtract` etc.

The **functions** statement class models external library functions that are not atomic operations. We distinguish between two types of functions. The *norm* functions and other functions. By norm, we mean a function that can invoke arbitrary dependencies between cells e.g., normalization function etc. and this cannot guarantee disjointness when split.

A **sink** statement represents a usage of data frames (as an argument) to either train or test a model.

3.2 Abstract Domain

In this subsection we describe our abstract domain for tracking the origin of data frames in variables.

3.2.1 Abstract Data Frame.

Columns: We first describe an abstract domain to track columns in a data frame. Note, since we do not assume a schema, we do not know *a priori* which columns are in a given data source. We have observed that since columns labels are almost always strings, it is very rare to specify inclusion or removal as numerical indexes. For this reason we propose an abstraction that is set based. Given a set of column labels C, we also introduce a set of *negative* dual labels in a set $\tilde{C}$. For a positive element $c \in C$ we denote the negative dual element as $\tilde{c} \in \tilde{C}$. As is the case with negation $\tilde{\tilde{c}} = c$, which also holds for sets i.e., $\tilde{\tilde{C}} = C$. In other words, the dual of the dual label is the original label. We define a abstract domain $\langle Col, \sqsubseteq_{Col} \rangle$ for tracking the columns in a data frames, namely,

$$Col = \{C' \mid C' \in \wp(C \cup \tilde{C}) \wedge \forall c_1 \in C'.\neg\exists c_2 \in C'.c_2 = \tilde{c_1}\}$$

We say $C' \sqsubseteq_{Col} C''$ iff $C'' \subseteq C'$ (note the inverse relationship), that is an element with more information about included or excluded columns is more precise than a column sets with less information. The empty set is the least precise i.e., has no information on what columns may exist and a set of all columns (represented as positive or (exclusively) negatives) is the most precise. We also define an operator Red_{Col} that reduces a non-canonical set to a canonical set by applying the rule $\bar{c} \cup \{x, \bar{x}\} = \bar{c}$. We define $\sqcup_{Col}$ operator as $Red_{Col} \circ \cup$ and $\sqcap_{Col}$ operator as $\cap$.

Example 3.1 (Column Domain). Consider the column set $C = \{id, name, city, country, zip\}$. The abstract column set elements $C_1 = \{id, name, \tilde{city}\}$, $C_2 = \{id, country, city\}$. Here C_1 asserts that the columns are *id* and *name* columns. It does not have a *city* column and it may have a *country* and *zip* column. C_2 asserts that the columns are *id* and *country* and *city*. It may have all the other columns. $C_1 \sqcup_{Col} C_2 = \{id, name, country\}$. Note that the intermediate result is reduced to not contain *city*. The join, $C_1 \sqcap_{Col} C_2 = \{id\}$.

Rows: We model the selection of rows with an interval domain of natural numbers. Rows are not named, and a data frame can have a large number of rows. Often ranges of rows are added or removed. Therefore, we find the interval domain as adequate for this task, we denote the interval domain as Int_+ [5]. We define two functions $lw(\bar{r})$ which takes the lowest value of $\bar{r}$, and $up(\bar{r})$ which takes the highest value.

Data Frames: We now bring together the abstract domains for columns and rows and define an abstract domain for data frames. We define the abstract data frame as a triple:

$$\langle v, [l, u], \bar{c} \rangle \in L = (V \cup S) \times Int_+ \times Col$$

For succinctness we use the notation $v^{\bar{c}}_{[l,u]}$ for an abstract data frame. We define join ($\sqcup_L$) and meet ($\sqcap_L$) operators as follows:

$$x^{\bar{c}}_{[l_1,u_1]} \sqcup_L x^{\bar{c}'}_{[l_2,u_2]} = x^{\bar{c} \sqcup_{Col} \bar{c}'}_{[l_1,u_1] \sqcup_{Int_+} [l_2,u_2]}$$
$$x^{\bar{c}}_{[l_1,u_1]} \sqcap_L x^{\bar{c}'}_{[l_2,u_2]} = x^{\bar{c} \sqcap_{Col} \bar{c}'}_{[l_1,u_1] \sqcap_{Int_+} [l_2,u_2]}$$

Note, joins and meets are performed on data frames from the same sources. We also define an overlap (ol) predicate that asserts if two data frames from the same source overlap.

$$\begin{aligned} ol(x^{\bar{c}}_{[l_1,u_1]}, x^{\bar{c}'}_{[l_2,u_2]}) =& [l_1, u_1] \sqcap_{Int_+} [l_2, u_2] \neq \bot \\ & \wedge (\bar{c} \sqcap_{Col} \bar{c}' \neq \emptyset) \end{aligned}$$

Moreover, we define a constraint function $\downarrow^{\bar{c}}_{[l,u]}$ that constrains data frames from an interval and abstract column set.

$$\downarrow^{\bar{c}'}_{[l',u']} (v^{\bar{c}}_{[l,u]}) = v^{\bar{c} \sqcap_{Col} \bar{c}'}_{[l,u] \sqcap_{Int_+} [l',u']}$$

We also define a minus operator on data frames, namely $\ominus$:

$$x^{\bar{c}}_{[l_1,u_1]} \ominus_L x^{\bar{c}'}_{[l_2,u_2]} = \begin{cases} x^{\bar{c}-\bar{c}'}_{[l_1,l_2-1]} & l_1 < l_2 \leq u_1 \leq u_2 \\ x^{\bar{c}-\bar{c}'}_{[u_2+1,u_1]} & l_2 \leq l_1 \leq u_2 < u_1 \\ x^{\bar{c}-\bar{c}'}_{\bot} & l_2 \leq l_1 \leq u_1 \leq u_2 \\ x^{\bar{c}-\bar{c}'}_{[l_1,l_2-1]}, x^{\bar{c}-\bar{c}'}_{[u_2+1,u_1]} & l_1 < l_2 < u_2 < u_1 \end{cases}$$

Example 3.2 (Abstract Data Frame). A data frame with known columns $\{id, city\}$ and rows $\{10, 12, 13, 14\}$ from a source *file1* can be abstracted as $file1^{\{id,city\}}_{[10,14]}$. The join and meet perform a component wise join on the same source. The data frame $file1^{\{id\}}_{[13,14]}$ overlaps but the data frame $file1^{\{country\}}_{[13,14]}$ does not. $\downarrow^{\{id\}}_{[12,15]} (file1^{\{id,city\}}_{[10,14]})$ results in $file1^{\{id\}}_{[12,14]}$. Finally, $file1^{\{id,city\}}_{[10,14]} \ominus_L file1^{\{id\}}_{[13,14]}$ yields $file1^{\{id\}}_{[10,12]}$.

Data Frame Sets: We lift the abstract data frame domain to sets that is we define $\langle \bar{L}, \sqsubseteq_{\bar{L}} \rangle$ where $\bar{L} = \wp(L)$ ordered by $\sqsubseteq_{\bar{L}}$ define as follows $\subseteq$ and where

$$\sqcup_{\bar{L}} = Red_{\bar{L}}^{\sqcup_L} \circ \cup$$
$$\sqcap_{\bar{L}} = Red_{\bar{L}}^{\sqcap_L} \circ \cap$$

We define a reduction operator $Red_{\bar{L}}^{op}$ as:

$$Red_{\bar{L}}^{op} = \begin{cases} \{l_1 \; op \; l_2 \mid \forall l_1, l_2 \in \bar{L}.ol(l1, l2)\} \cup \\ \{l_1 \mid \forall l_1 \in \bar{L}.\neg\exists l_2 \in \bar{L}.ol(l1, l2)\} \end{cases}$$

Essentially, after a join or meet is performed (op) the set of data frames may overlap and thus the set needs to be put into a canonical form. We also lift the $\downarrow$, and $\ominus$ functions to sets where the set version perform the operations for each element in the set. Since $\ominus$ returns a set, a *flattening* is performed.

Example 3.3 (Data Frame Sets)**.** Consider the data frame sets $\{file1_{[1,10]}^{\{id\}}, file2_{[0,100]}^{\{name\}}\}$ and $\{file1_{[9,12]}^{\{id\}}, file3_{[0,100]}^{\{zip\}}\}$. If we naively perform a join, we can get elements which can be joined into a single data frame i.e.,

$$\{file1_{[1,10]}^{\{id\}}, file1_{[9,12]}^{\{id\}}, file2_{[0,100]}^{\{name\}}, file3_{[0,100]}^{\{zip\}}\}$$

The reduction makes the set canonical i.e.,

$$\{file1_{[1,12]}^{\{id\}}, file2_{[0,100]}^{\{name\}}, file3_{[0,100]}^{\{zip\}}\}$$

Note, if we do not perform this reduction, it potentially hinders performance and violation detection.

3.2.2 Abstract State.

We define the *abstract state* as a mapping $V \longrightarrow D$ between variables in V and an *abstract domain* where

$$D = \langle \bar{L} \times \mathbb{B} \times \wp(\{tr.ts\}), \sqsubseteq \rangle$$

where we define $\sqsubseteq = \sqsubseteq_{\bar{L}} \times \sqsubseteq_{\mathbb{B}} \times \subseteq$ and $\sqcup = \sqcup_{\bar{L}} \times \sqcup_{\mathbb{B}} \times \cup$ and $\sqcap = \sqcap_{\bar{L}} \times \sqcap_{\mathbb{B}} \times \cap$.

Intuitively, for variable we keep track of (1) the set of data frames it is dependent on, (2) if those data frames are tainted i.e., have data that cannot be safely decomposed and (3) if any data frames have been used for training or testing. We differentiate between which tuple element in the product domain is access by $\sigma^{\sharp 1}$ (first tuple element) and $\sigma^{\sharp 2}$ (second tuple element) and $\sigma^{\sharp 3}$ (third tuple element).

Since data frames can have infinite ascending chains w.r.t the row interval abstraction and column abstraction, a simple widening operator can be applied for rows [5] and similarly, for columns the set can be set to $\emptyset$ (the top element) if the set of columns are detected to be reducing inside a loop.

3.2.3 Knowledge Base.

We also assume 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.

3.3 Abstract Semantics

We define an abstract semantics for our data frame language. The abstract semantics for each category of statements is described below.

1. **source:**

$$\lambda\sigma^{\sharp}.[\![y = \text{read}(name)]\!] = \sigma^{\sharp}[y \mapsto (\{name_{[0,\infty]}^{\emptyset}\}, F, \emptyset)]$$

where $read \in KB_{source}$

2. **select-project:**

$$\lambda\sigma^{\sharp}.[\![y = x.sel[\bar{r}][\bar{c}]]\!] =$$
$$\sigma^{\sharp}[y \mapsto (\sigma^{\sharp 1}(x) \downarrow_{[lw(\bar{r}), up(\bar{r})]}^{\bar{c}}, \sigma^{\sharp 2}(x), \sigma^{\sharp 3}(x))]$$

3. **operations:**

$$\lambda\sigma^{\sharp}.[\![y = op(x_1, x_2)]\!] =$$

a. op = union:

$$\sigma^{\sharp}[y \mapsto \sigma^{\sharp}(y) \sqcup (\sigma^{\sharp}(x_1) \sqcup \sigma^{\sharp}(x_2))]$$

b. op = merge:

$$\begin{cases} \sigma^{\sharp}[y & \mapsto \sigma^{\sharp}(y) \sqcup (\\ & \sigma^{\sharp 1}(x_1) \sqcap_{\bar{L}} \sigma^{\sharp 1}(x_2), \\ & \sigma^{\sharp 2}(x_1) \sqcup_{\mathbb{B}} \sigma^{\sharp 2}(x_2), \\ & \sigma^{\sharp 3}(x_1) \cup \sigma^{\sharp 3}(x_2), \\ &)] \end{cases}$$

c. op = diff:

$$\begin{cases} \sigma^{\sharp}[y & \mapsto \sigma^{\sharp}(y) \sqcup (\\ & \sigma^{\sharp 1}(x_1) \ominus_{\bar{L}} \sigma^{\sharp 1}(x_2), \\ & \sigma^{\sharp 2}(x_1) \sqcup_{\mathbb{B}} \sigma^{\sharp 2}(x_2), \\ & \sigma^{\sharp 3}(x_1) \cup \sigma^{\sharp 3}(x_2), \\ &)] \end{cases}$$

4. **functions:**

$$\lambda\sigma^{\sharp}.[\![y = f(\bar{x})]\!] =$$

a. $f \in KB_{taint}$:

$$\begin{cases} \forall x & \in \bar{x}.\sigma^{\sharp}[y \mapsto \\ & (\sigma^{\sharp 1}(y) \sqcup_{\bar{L}} rn(\sigma^{\sharp 1}(x), y), \\ & T, \sigma^{\sharp 3}(y) \cup \sigma^{\sharp 3}(x))] \end{cases}$$

b. otherwise:

$$\sigma^{\sharp}[y \mapsto \sigma^{\sharp}(y) \sqcup \bigsqcup_{x \in \bar{x}} \sigma^{\sharp}(x)]$$

5. **sink**:

$$\lambda\sigma^{\sharp}.[\![f(\bar{x})]\!] = \begin{cases} \forall x \in \bar{x}.\sigma^{\sharp}[x \mapsto (\sigma^{\sharp 1}(x), \sigma^{\sharp 2}(x) \sqcup \{tr\})] \\ \text{iff } f \in KB_{\text{train}} \\ \forall x \in \bar{x}.\sigma^{\sharp}[x \mapsto (\sigma^{\sharp 1}(x), \sigma^{\sharp 2}(x) \sqcup \{ts\})] \\ \text{iff } f \in KB_{\text{test}} \end{cases}$$

In the above abstract semantics the read statement produces a mapping between the data frame variable y and the filename. Because we do not statically know any information about the file data, we simply do not constrain columns (empty set) or rows (unbounded interval).

For the union, merge and diff statement, we perform a corresponding data frame operation on the two data frames x_1 and x_2, namely, join, meet and minus and add it to the previous state of y.

For the norm function, we *reset* the mapping, by setting the outgoing variable as the source mapping using the rename (rn) function which simply renames all source variables in the first argument to the variable in the second argument. We omit the definition of rn as it is intuitive. We also set the flag to true (T). If the function is not a norm function, we simply propagate the state information.

Finally in the case of a sink statement we mark each argument with a {tr} element if it is an argument to a train function, and {ts} if it is an argument to a test function.

Example 3.4 (Motivating Example (Cont.))**.** Consider our motivating example again. In the program a CSV file is read into a data frame (line 8). This creates a mapping of

$$df \mapsto (\{data^{\emptyset}_{[0,\infty]}\}, F, \emptyset)$$

From the data frame columns are selected to represent x and y co-ordinates (lines 14, 17). This adds constraints so that

$$x \mapsto (\{data^{\{col1,col2\}}_{[0,\infty]}\}, F, \emptyset)$$

$$y \mapsto (\{data^{\{col3\}}_{[0,\infty]}\}, F, \emptyset)$$

When each coordinate is normalized (lines 15) we flip the second tuple in the domain to T and *reset* the source by setting the source to the variable that holds the normalized data. When we split by selecting ranges of rows from the data frame (lines 20, 21, 23, 24) we perform a selection and get mappings

$$X\_train \mapsto (\{x\_selected^{\{col1,col2\}}_{[0,2]}\}, T, \emptyset)$$

$$X\_test \mapsto (\{x\_selected^{\{col1,col2\}}_{[4,5]}\}, T, \emptyset)$$

The training data is then used to train (line 30) the model and it is then tested (line 33) to evaluate its accuracy (line 36). At the end we obtain:

$$X\_train \mapsto (\{x\_selected^{\{col1,col2\}}_{[0,2]}\}, T, \{tr\})$$

$$X\_test \mapsto (\{x\_selected^{\{col1,col2\}}_{[4,5]}\}, T, \{ts\})$$

which violates the error conditions because they have the same source, and despite being disjoint, both passed through a function which tainted them. Consider if normalization was called after splitting. X_train and X_test would have different sources and be disjoint. We would then assert that no data leak occurred. Also consider there is no tainted normalizing function and no splitting. Without the flag, we would have to assert false positives (any two variables with same source can't be trained and tested).

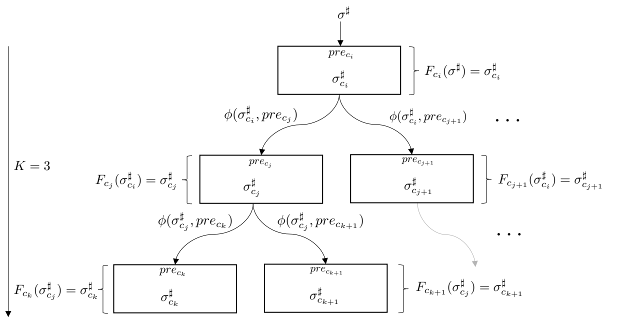
$\sigma^\sharp$
pre_{c_i}
$\sigma^\sharp_{c_i}$
$F_{c_i}(\sigma^\sharp) = \sigma^\sharp_{c_i}$
$\phi(\sigma^\sharp_{c_i}, pre_{c_j})$
$\phi(\sigma^\sharp_{c_i}, pre_{c_{j+1}})$
. . .
$K = 3$
$F_{c_j}(\sigma^\sharp_{c_i}) = \sigma^\sharp_{c_j}$
pre_{c_j}
$\sigma^\sharp_{c_j}$
$pre_{c_{j+1}}$
$\sigma^\sharp_{c_{j+1}}$
$F_{c_{j+1}}(\sigma^\sharp_{c_i}) = \sigma^\sharp_{c_{j+1}}$
$\phi(\sigma^\sharp_{c_j}, pre_{c_k})$
$\phi(\sigma^\sharp_{c_j}, pre_{c_{k+1}})$
. . .
$F_{c_k}(\sigma^\sharp_{c_j}) = \sigma^\sharp_{c_k}$
pre_{c_k}
$\sigma^\sharp_{c_k}$
$pre_{c_{k+1}}$
$\sigma^\sharp_{c_{k+1}}$
$F_{c_{k+1}}(\sigma^\sharp_{c_j}) = \sigma^\sharp_{c_{k+1}}$

Figure 1. Inter Cell Analysis

4 Integration into NBLyzer

We have integrated our analysis into NBLyzer, a static analysis framework for data science notebooks as described in [12]. The analysis technique is depicted in Figure 1.

The analyzer ingests the statements as a CFG as defined in Subsection 2.1. The analysis operates by performing standard static analysis intra cell. At the cell level this computation is defined as F_{c_i} which we refer to a *abstract cell transformer*. F_{c_i} takes an abstract state and computes a fix-point solution [10, 13] in the abstract domain. At each execution, a cell transformer F_{c_i} for a cell c_i is applied with the current state, returning an updated new state i.e., $F_{c_i}(\sigma^\sharp) = \sigma^{\sharp\prime}$.

For inter cell analysis, the abstract state needs to be propagated from one cell to another for a fixed depth of K or until a notebook wide fixed point is reached ($K = \infty$). It relies on an *inter cell dependency graph* which is defined by a predicate ϕ. Each analysis needs to define ϕ along with its abstract semantics. Moreover, each cell has preconditions pre_{c_j}, typically the set of unbound variables. If ϕ holds, the abstract state is propagated to the dependent cells, for which the incoming abstract state is treated as a initial state. For each cell the abstract state is checked for correctness criteria, if an error is found a report is updated which serves as instruction for notebook clients to alert the user to the consequences of the event (e.g., by cell highlighting etc.).

To integrate into the NBLyzer framework we specify the additional ϕ condition for inter-cell propagation as follows:

$$\phi(\sigma^\sharp_{c_i}, pre_{c_j}) = pre_{c_j} \subseteq \{v : (v \mapsto x^{\bar{c}}_{[l,u]}) \in \sigma^\sharp_{c_i} \wedge [l,u] \neq \bot\} \\ \wedge pre_{c_j} \neq \emptyset$$

This rule stipulates the condition by which a successor cell should be analyzed. That is, it states that if variables with rows (do not map to $\bot$ interval) in the abstract state of the current notebook cell, are also unbound in the successor notebook cell (pre_{c_j}), then we proceed to propagate the abstract state to that successor cell.

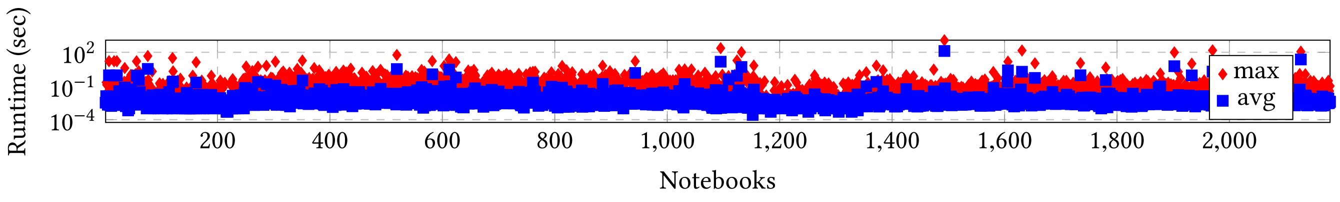

Figure 2. Data Leakage Analysis Avg. and Max. Analysis Times

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

5 Experimental Evaluation

Experimental Setup: All experiments were performed in an Intel(R) Xeon W-2265 CPU @ 3.50 GHz with 64 GB RAM running a 64 bit Windows 10 operating system. Python 3.8.8 was used to execute NBLyzer. We evaluate the execution-time of our static analyses running within NBLyzer on the full set of Kaggle notebooks.

We use a benchmark suite consisting for 2211 executable real-world notebooks from the Kaggle competition[1] that has previously been used to evaluate data science static analyzers [8]. The benchmark characteristics are summarized in Table 1. We emphasise that this is a fair reflection of typical notebook code.

We see 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. We note that these characteristics of low amount of branching, functions and classes are typically advantageous for static analysis precision. We found that every second notebook had a cell that could not be parsed and analyzed due to syntax errors in it. Overall, this affected 4% of cells in the benchmarks. 2.1 of variables were unbound, from an average of 8.2 variables per cell.

Performance Evaluation: We evaluate the performance of the data leakage analysis. This analysis is run on the NBLyzer setting $K = \infty$ (propagate till fixed point). In Figure 2 the average and maximum data leakage analysis for executions are shown. The results how that the average data leakage analysis on a notebook takes 41.45 milliseconds. The average maximum analysis per notebook take 880.9 milliseconds, with a global maximum of 233 seconds. Since this analysis is run as a *what-if* analysis in a notebook, we require fast response times (ideally < 1000ms) to retain the interactive notebook experience [2]. **The analysis time for average case is well under the threshold for users to notice any delay** and does not degrade the user experience. **The average maximum recorded analysis time is above the immediate fell threshold, but below the threshold for the task feeling out of flow** (1000ms). The global maximum does cause considerable delay and user degradation. Moreover, only 4% of all analyses execute for more than 1000ms and only 1% for more than 5000ms.

6 Related Work

Static Analysis for Data Science. Static analysis for data science is an emerging area in the program analysis communities. A comprehensive state of the art is outlined in [14]. In [12] is a static analysis framework for data science notebooks. We implement our technique inside this framework. Our analysis can benefit from analyses such as the one presented in [7] and [15].

Data Leakage Detection and Avoidance. Several techniques exist to avoid and discover data leakages in data science code. For example, a popular technique is the use of data science piplelines [4] that stage the phases of sourcing, cleaning, splitting, normalization, and training to avoid performing a normalization step before splitting. This, however, requires manual effort and is not widely used among the millions of data scientists. For this reason tools such as [3] exist that perform dynamic instrumentation to detect a data leakage.

7 Conclusion

We have presented a method for detecting data science data leakages statically. Our technique has been integrated into the NBLyzer static analyzer. We believe this is the first static analysis for detecting data science data leakages. In our current prototype we have not implemented all operations but have reasonable coverage of pandas data frames. Our analysis does not model several dynamic langauge constructs. However, we believe in principle given an appropriate (lightweight) alias analysis this can be achieved. We also note that data science code is simple compared to mainstream programs, and rarely employs dynamic features.

Acknowledgments

We thank our colleagues at Microsoft Azure Data Labs and Gray Systems Lab for all their feedback and support.

References

[1] 2021. Kaggle. http://kaggle.com. Accessed: 2021-09-30.

[2] 2021. RAIL model. https://web.dev/rail/. Accessed: 2021-09-30.

[3] 2022. leak-detect. https://github.com/abhayspawar/leak-detect. Accessed: 2022-03-08.

[4] 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). https://doi.org/10.1145/3510003.3510057

[5] 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. https://doi.org/10.1145/512950.512973

[6] Shachar Kaufman, Saharon Rosset, and Claudia Perlich. 2011. Leakage in Data Mining: Formulation, Detection, and Avoidance. *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* 6, 556–563. https://doi.org/10.1145/2020408.2020496

[7] 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. https://doi.org/10.4230/LIPIcs.ECOOP.2020.15

[8] 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. https://doi.org/10.1145/1122445.1122456

[9] 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), 2033âĂŞ2046. https://doi.org/10.14778/3407790.3407807

[10] Peter Schrammel and Pavle Subotic. 2013. Logico-Numerical Max-Strategy Iteration. In *Proc. VMCAI*. 414–433. https://doi.org/10.1007/978-3-642-35873-9_25

[11] Vugranam C. Sreedhar and Guang R. Gao. 1995. A Linear Time Algorithm for Placing ÏĘ-Nodes. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '95)*. Association for Computing Machinery, New York, NY, USA, 62âĂŞ73. https://doi.org/10.1145/199448.199464

[12] 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). https://doi.org/10.1145/3510457.3513032

[13] Alfred Tarski. 1955. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math.* 5, 2 (1955), 285–309.

[14] 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

[15] Caterina Urban and Peter Müller. 2018. An Abstract Interpretation Framework for Input Data Usage. In *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10801)*, Amal Ahmed (Ed.). Springer, 683–710. https://doi.org/10.1007/978-3-319-89884-1_24