TaintStream: Fine-Grained Taint Tracking for Big Data Platforms through Dynamic Code Translation

Chengxu Yang*†
Key Lab of High Confidence Software Technologies (Peking University), MoE
Beijing, China
yangchengxu@pku.edu.cn

Yuanchun Li†‡
Microsoft Research
Beijing, China
Yuanchun.Li@microsoft.com

Mengwei Xu*
State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications
Beijing, China
mwx@bupt.edu.cn

Zhenpeng Chen
Key Lab of High Confidence Software Technologies (Peking University), MoE
Beijing, China
czp@pku.edu.cn

Yunxin Liu*
Institute for AI Industry Research (AIR), Tsinghua University
Beijing, China
liuyunxin@air.tsinghua.edu.cn

Gang Huang
Xuanzhe Liu‡
Key Lab of High Confidence Software Technologies (Peking University), MoE
Beijing, China
hg@pku.edu.cn
liuxuanzhe@pku.edu.cn

ABSTRACT

Big data has become valuable property for enterprises and enabled various intelligent applications. Today, it is common to host data in big data platforms (*e.g.*, Spark), where developers can submit scripts to process the original and intermediate data tables. Meanwhile, it is highly desirable to manage the data to comply with various privacy requirements. To enable flexible and automated privacy policy enforcement, we propose *TaintStream*, a fine-grained taint tracking framework for Spark-like big data platforms. *TaintStream* works by automatically injecting taint tracking logic into the data processing scripts, and the injected scripts are dynamically translated to maintain a taint tag for each cell during execution. The dynamic translation rules are carefully designed to guarantee non-interference in the original data operation. By defining different semantics of taint tags, *TaintStream* can enable various data management applications such as access control, data retention, and user data erasure. Our experiments on a self-crafted benchmark suite show that *TaintStream* is able to achieve accurate cell-level taint tracking with a precision of 93.0% and less than 15% overhead. We also demonstrate the usefulness of *TaintStream* through several real-world use cases of privacy policy enforcement.

*This work was done while Chengxu Yang, Mengwei Xu, and Yunxin Liu were working at Microsoft (as an intern, visiting scholar, and researcher, respectively). † Chengxu Yang and Yuanchun Li contributed equally. ‡ Correspondence goes to Yuanchun Li and Xuanzhe Liu.

ESEC/FSE '21, August 23–28, 2021, Athens, Greece

ACM ISBN 978-1-4503-8562-6/21/08…$15.00
https://doi.org/10.1145/3468264.3468532

CCS CONCEPTS

• **Security and privacy → Information flow control**; **Information accountability and usage control**.

KEYWORDS

Taint tracking, big data platform, privacy compliance, GDPR

ACM Reference Format:
Chengxu Yang, Yuanchun Li, Mengwei Xu, Zhenpeng Chen, Yunxin Liu, Gang Huang, and Xuanzhe Liu. 2021. TaintStream: Fine-Grained Taint Tracking for Big Data Platforms through Dynamic Code Translation. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3468264.3468532

1 INTRODUCTION

In the past decade, we have witnessed an explosion of data and rapid advances in data analysis techniques. To handle the massive amount of data and complicated data processing requests, several big data analytics engines [19, 37, 55] are proposed. Today, managing big data with a unified platform that stores data in distributed file systems (*e.g.*, HDFS) and offers data processing ability with Stream-like APIs (*e.g.*, Spark) has become a standard in industry [7].

There are typically three roles involved in a big data platform, including data providers, data analysts, and data managers. *Data providers* are typically the end-users or consumers that contribute the raw data to the platform. For example, each user of an email service would produce her email messages to the service maintainer. Data providers are usually subject to privacy protection, and thus the data provided by them must be managed properly to ensure privacy compliance. *Data analysts* are individuals or organizations that submit code to process the data. They may have different identities (from different teams, companies, or countries), with different permissions, and/or have different purposes (for advertising, product

development, scientific research, etc.). *Data managers* are usually the platform maintainers whose job is to ensure that the data is used legitimately. They should arrange the data, sweep the data periodically, and manage data access to avoid privacy violations. In this work, we stand in the role of data managers and try to provide a generic solution for privacy protection.

The current industrial practice of privacy protection for big data platforms is mainly based on manual code review and coarse-grained control. To meet different privacy requirements, various non-trivial mechanisms and rules for data storage, access, and processing are designed [24, 29]. Both the analysts and the managers are required to learn the rules, and any script submitted to the platform must go through a time-consuming review process to ensure compliance. Privacy policies such as data retention and access control are usually enforced in a per-dataset manner, which forces the data manager and analysts to make a difficult choice between convenience and data utilization ratio: if they want to fully utilize the data, they must manually organize the dataset in fragments, so that the expiration and sensitivity of one fragment do not affect the availability of others. Maintaining and processing the data fragments are also cumbersome for developers. As a result, a more automated, flexible, and fine-grained solution for privacy compliance is highly desirable.

Taint analysis [11, 13, 40, 42] is a promising technique for fine-grained data tracking and privacy protection. The goal of taint analysis is to detect whether the sensitive information from a source point (*e.g.*, original personal data) may be propagated to a target point (*e.g.*, unauthorized third parties). Taint analysis can be conducted statically by analyzing the reachability in data-flow graphs, or dynamically by tracking the variable definitions and assignments at runtime.

Porting existing taint analysis techniques to big data platforms involves several challenges: (1) Data processing scripts (*e.g.*, Spark scripts) are designed to describe high-level operations over the whole dataset (`select`, `filter`, `groupBy`, etc.) rather than direct information propagation (`mov`, `add`, etc.), which makes it difficult for static taint analysis methods to extract fine-grained data flow. (2) The data in the platforms is usually scattered and transferred across different computing and storage nodes, which makes it difficult to apply traditional (register or memory-based) dynamic taint tracking methods [13, 42]. (3) Big data platforms have a higher requirement for stability and maintainability of the runtime environment, so directly modifying the system to achieve tracking is usually unacceptable for its high risk and maintenance cost.

In this paper, we propose *TaintStream*, a fine-grained taint tracking framework for big data platforms. *TaintStream* solves the above challenges through dynamic code translation, which converts the original data processing script at runtime to additionally maintain a taint tag for each data cell (the minimum unit in a dataset). Such a method combines the advantages of static and dynamic analysis techniques: The code rewriting is performed statically before running the script so that the rewritten script can seamlessly be executed in the unmodified analytic engine. The data processing pipeline translation is completed dynamically at runtime when the necessary information for taint propagation logic injection is available. Since the injected taint propagation logic is flexible and customizable similar to classic data-flow analysis frameworks, *TaintStream* is able to fulfill various privacy protection requirements in fine granularity.

Specifically, given a data processing script pending execution, *TaintStream* first scans the code and wraps the data processing operations with a translation function ϕ. When the rewritten script is executed, ϕ is invoked to interpret the data processing operations. The interpretation follows a set of translation rules, which are carefully designed to ensure the conservativeness of taint propagation and the non-interference in the original data processing operations. The translated data processing pipeline will operate on an extended dataset where each cell has an accompanying taint tag field. The taint tags are maintained during data processing, and the output data will have taint tags as well. Based on the taint propagation mechanism, various privacy policies can be automatically enforced by formulating the policy in *TaintStream*'s format, *i.e.*, defining the tag type and initializing, merging, and sanitizing functions. For example, in data retention, each taint tag is a timestamp indicating the expiration date of the data. Such a method is generic across most modern big data platforms.

We implement the prototype based on Spark [55] that is widely used in practice. To ensure the robustness of the translated script, we add an exception handling mechanism, which will roll back the dataset to a safe and conservative state once the translation fails. We also include some simple rules to fuse data propagation operations to improve the efficiency of translated scripts.

We evaluate *TaintStream* with two sets of scripts. The first is a self-built benchmark that contains 33 scripts that cover most common data processing operations. The second one is a set of real-world scripts obtained from a production data platform in industry. The results show that *TaintStream* is able to accurately track taint tags in the cell-level granularity with a precision of 93.0% and a recall of 100.0%, which outperforms baselines that can track only column-level taint propagation with an accuracy lower than *TaintStream*. We also conduct a qualitative comparison between *TaintStream* and the current privacy enforcement methods in industry, which shows *TaintStream*'s benefits in reducing manual efforts and improving data utilization. The average time overhead and storage overhead of *TaintStream* are 12.7% and 2.06%, respectively, which are acceptable as compared with the extra manual efforts required by existing solutions.

We summarize our contributions as follows.

- We propose a fine-grained taint tracking framework [1] to flexibly support various privacy requirements on big data platforms while overcoming the limitations of existing static and dynamic taint analysis techniques.
- We introduce a benchmark for evaluating the performance of information flow tracking methods for big data platforms. Our method achieves a high precision of 93.0% and a 100% recall with respect to the benchmark.
- We demonstrate various privacy enforcement cases where *TaintStream* can be used to largely automate the process and improve protection granularity, as compared with existing solutions in practice.

[1] Our code is open-sourced at https://github.com/PrivacyStreams/TaintStream.

Table 1: Syntax of stream-like data processing APIs.

df	::=	$source$
	|	$df.transformation$
$transformation$	::=	$select(Col)$
	|	$drop(Col)$
	|	$orderBy(Col)$
	|	$filter(Col)$
	|	$join(df, on = Col)$
	|	$union(df)$
	|	$withColumn(str, Col)$
	|	$groupBy(Col_1).agg(F_{agg_1}(Col_2), F_{agg_2}(Col_3), ...)$
	|	$map(Func).reduce(Func)$
	|	...
Col	::=	$column\_name$
	|	$const$
	|	$Func(Col_1, Col_2, ...)$
	|	$F_{agg}(Col)$

2 BACKGROUND AND MOTIVATION

In this section, we first introduce some background, including the general syntax of stream-like data processing APIs (§2.1) and privacy compliance requirements in big data platforms (§2.2). We then introduce a simple but full-fledged example to show the necessity of fine-grained taint tracking.

2.1 Stream-like Data Processing API

Most big data processing engines including Spark [2], Hadoop [37], Kafka [27], etc. adopt a stream-like API for data processing. The core concept of stream-like APIs is to view the data as a sequence of elements and support sequential and parallel functional-style operations over the elements.

Table 1 shows the general syntax of the stream-like data processing APIs. The data source (*e.g.*, a table loaded from storage, a live data stream generated from users, etc.) is loaded as a dataframe (df), which can be viewed as a sequence of elements with named columns. The dataframe can be transformed into another dataframe with various operations, including *select*, *filter*, *map/reduce*, etc. For example, some operations are used to include, exclude, or create columns (*select*, *drop*, *withColumn*), some are to filter, convert, or group elements (*filter*, *map*, *groupBy*), and some are to merge the data from different streams (*join*, *union*). The parameters of these operations are typically column names, constant values, and functions ($Func$, F_{agg}, etc.).

Such a pipeline-style data processing API is similar to SQL. In fact, stream-like APIs and SQL APIs are interchangeable in some cases [39]. However, the stream-like APIs have become the most common interface in most data platforms [2] due to its simplicity, flexibility (seamless support of customized functions), parallelizability (support of efficient operations like *map/reduce*), and real-time processing ability (the data source can be real-time data streams). Our taint tracking method can be applied to traditional SQL databases as well.

2.2 Data Management and Taint Tracking

The data processed on big data platforms must be properly managed to ensure privacy compliance because it is usually collected or generated from users.

There are several laws and regulations related to user privacy protection all around the world, such as GPDR [50], HIPAA [51], COPPA [49], etc. These regulations apply to a broad range of enterprises that are processing the personal information of individuals under protection. In addition, many companies and data platform owners introduce more strict data access, processing, and sharing policies to further ensure data security and earn users' trust.

Below we highlight three typical examples of data management tasks that will be commonly used throughout the paper:

(1) **Data retention.** A data retention policy regulates how long the personal data can be retained by the data collector. The raw data and the data inferred from it must be deleted after a certain retention period (*e.g.*, three months). The retention period may vary for different types of data. Data retention requirements are common in many regulations including GDPR and HIPAA.
(2) **Access control.** Access control is needed when multiple parties are sharing the data platform. The access to certain data should be restricted if the requester is unauthorized. For example, in an international company with multiple teams, the teams in one country may not be allowed to access the data from another country, and the advertisement team may not be allowed to read children's data.
(3) **User data erasure.** GDPR also emphasizes the users' right to erasure (*i.e.*, the right to be forgotten). When a user requests erasure, the data collected from him/her and generated based on it must be deleted.

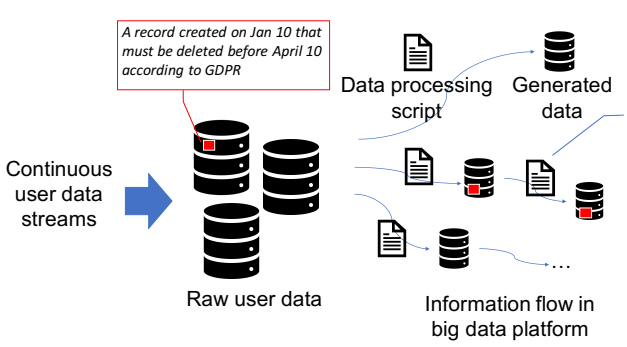
Taint tracking (aka. information flow tracking) is a common technique to achieve the goal of privacy-preserving data management. In a taint tracking system, the data records are tagged with marks indicating whether and how the record is tainted, and the tags may be propagated during computation to keep track of the information flow. A straightforward example is the access control mechanism in modern operating systems. Each file is typically tagged with its owner and/or access mode (readable, writable, executable, etc.), and the files computed from it will inherit the tags. Then the system can simply check the permission tags to decide whether an operation of a user is allowed on the files.

2.3 Motivating Example and Challenges

We are motivated to investigate the problem of taint tracking in big data platforms based on our experience working with an industrial data management team (represented as *Team A* in the rest of this paper) who operates a big data platform. We believe the motivation is shared by many other companies and data platforms.

In such data platforms, confidential user data is continuously generated from end-users and/or stored in a large database, and the raw user datasets are processed by various developers like data scientists and business analysts for different purposes, including supporting personalized services, mining user behavior, and gaining new business insights. Ensuring data security and privacy compliance is usually necessary for these platforms.

A motivating scenario is shown in Figure 1. *Team A* has a data platform hosting the data generated from client-side applications. The data is open to different teams through Spark APIs [2, 55] to facilitate data analytic and development of new features. We assume



A sample data processing script:

```
# user-defined function that constructs email reply pairs
func build_pairs(arg, …) {…}
# dataset "emails" contains emails in different conversations
# select conversations containing more than one email
convs = emails.groupBy($conversationId)
              .agg(count(*).alias($count),
                   collect_list($msg).alias($msgList))
              .filter($count > 1)
# construct email reply pairs
emails = emails.join(convs, on=$conversationId, "inner")
emails = emails.orderBy($date)
emails = emails.withColumn($replyPairs, build_pairs($msgList))
write_to_hdfs(emails)
```

Figure 1: A motivating scenario for fine-grained taint tracking. The expired records in the raw datasets and generated datasets must be deleted to fulfill GDPR data retention requirements.

the privacy policy to be enforced is a simplified version of data retention, which requires any personal data records created three months ago must be deleted, so do other data records computed from it. Deleting the expired data is challenging because the expiration date information may be missing in the datasets generated by the analytic scripts. An example script is shown on the right side of Figure 1, which is simplified from a real-world script that takes user messaging data as input and generates training data for a smart-reply machine learning model. Note that the scripts are chainable, *i.e.*, the output of a script can be the input of another script, making the data management even more complicated.

The current practice in *Team A* to ensure privacy compliance is mainly based on manual code review and coarse-grained dataset lineage analysis. For example, the datasets are partitioned and labeled with different data types and sensitivity levels that are propagated to the generated datasets. The privacy-related requirements are enforced based on these dataset labels in an all-or-nothing manner, *e.g.*, the whole dataset needs to be deleted if any record in it is expired. In cases where coarse-grained automated checking is insufficient, the developers must follow some coding rules to keep track of sensitive data (*e.g.*, manually maintain a user ID column for each record to handle user erasure requests) when writing the scripts and submit the scripts for expert review before actually executing them, which typically takes hours or days. The current solutions are time-consuming, inconvenient, and hard to maintain, since both developers and code reviewers are required to learn and obey complicated privacy requirements. An automated solution to fine-grained information flow tracking is highly desirable.

Taint analysis is a popular technique to achieve fine-grained data tracking in traditional programs. However, directly applying them to our problem involves the following technical challenges:

(1) **Missing traceable information in scripts.** Static data-flow analysis [4, 25, 46, 48] can extract data dependency between variables in a program. However, a script in big data platforms describes only the high-level operations (*e.g.*, `map`, `groupBy`, etc.) to be performed on the dataset, rather than the detailed assignments and computations (`mov`, `add`, etc.) between variables. Meanwhile, the schema and actual values of the datasets are missing in the scripts, making it difficult to extract fine-grained data-flow between data records from high-level operations.

(2) **Complicated low-level computation and storage.** Dynamic taint tracking techniques [6, 13, 35, 42] can track information propagation in traditional systems at the register, memory, and file level. However, applying these techniques to big data platforms is difficult because the storage and computation in big data platforms may be distributed, replicated, and out-of-order.

(3) **Diverse and flexible privacy policies.** Moreover, the privacy requirements may vary across different countries and organizations and evolve quickly. Supporting them in system level may require frequent system updating and rebooting. This would significantly harm the system stability and service quality, since many data processing jobs are long-running non-interruptable tasks.

To sum up, an ideal taint tracking mechanism for big data platforms should achieve three objectives: fine granularity (tracking values rather than datasets), lightweight (tracking at the semantic level rather than variable level), and ease of use (supporting flexible privacy policies).

3 OUR APPROACH: *TAINTSTREAM*

We introduce *TaintStream*, a fine-grained taint tracking technique for big data platforms. The core idea of *TaintStream* is to translate the data processing scripts during script execution to add taint propagation logic while retaining the fidelity of original operations.

3.1 Overview

The workflow of *TaintStream* is shown in Figure 2. For each data processing script submitted to the platform, we first instrument the script by wrapping the original stream pipeline with a dynamic translation function ϕ. The instrumented script is then executed in the original runtime (*e.g.*, Spark) where ϕ is invoked to interpret the stream pipeline.

The instrumented script operates on an extended version of the original dataset, in which each data cell is attached a (or multiple) taint tag(s). The taint tags for the raw data are generated upon dataset creation by the data managers according to the privacy

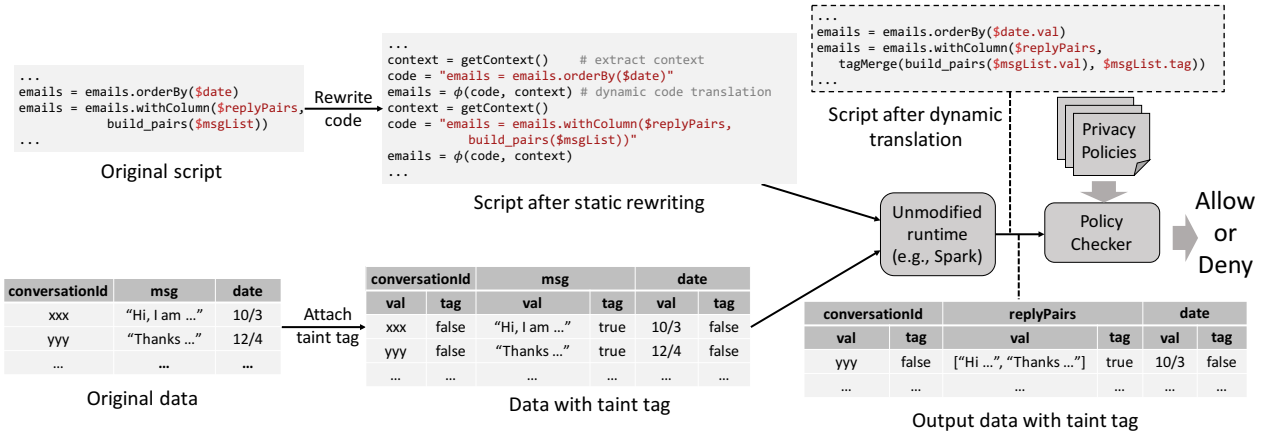


Figure 2: *TaintStream* workflow.

policies, and the extended versions of intermediate datasets are generated by our translated scripts. If the data cell has a hierarchical structure (*e.g.*, a $\langle Int, String \rangle$ pair), the taint tags are attached to the lowest-level values rather than the whole cell.

The type and semantic meaning of the taint tags are customizable based on the privacy policy to enforce. For example, in access control, the taint tag of each record can be set as the owner of the data, and anyone other than the owner can be denied to access the information. In date retention, each record can be tagged with its expiration date, so that data cleaning can be performed periodically to delete the expired records. For simplicity, the taint tags in Figure 2 are boolean values.

Based on how the data is processed in the original pipeline, the function ϕ determines how the taint tags should be propagated in each operation. The taint propagation logic is written as a normal stream operation and weaved into the original pipeline, so that the translated pipeline is also executable in the original runtime. For example, in *withColumn* operation, a new field (in this paper, we use field and column mutually) will be created for each element in the stream with a function that takes other fields as inputs. When being interpreted by ϕ, the *withColumn* operation will additionally aggregate the taint tags of input fields and pass the aggregated tag to the new field's tag. The aggregation function is also customizable based on the tracking objective. For example, when aggregating multiple values with different expiration dates, the earliest expiration date should be kept to ensure strict compliance. When aggregating two records with different authorized groups, the output record should be tagged with the intersection of the groups. Meanwhile, the aggregation function can also be customized based on the underlying data processing operation. For example, if the operation is to compute the average message length of users' messages, the taint tags can be cleared since the output value is no longer sensitive.

Finally, the translated script will produce output data with cell-wise taint tags. Privacy policy enforcement can be performed by checking the taint tags. For example, to enforce data retention, we can periodically scan the datasets and sweep the records that have reached the expiration date. To support fine-grained access control, we can make sure the records are only visible to the developers in authorized groups.

Here we explain how *TaintStream* addresses the challenges described in §2.3. First, our method combines the benefits of static analysis and dynamic analysis. The static code rewriting phase overwrites the original data processing pipeline with a modified (extended) pipeline, and the statically-unavailable information such as data schema is available during runtime dynamic translation. Second, our solution is light-weight, because the taint propagation logic is injected into the script through code rewriting rather than system modification, which hides the details about how the data is actually stored and processed at the low level. Last but not least, the injected taint propagation logic can be customized with code translation rules, and thus able to support flexible privacy requirements by defining different types of taint tags, aggregation functions, etc.

The following sections will introduce the core components of *TaintStream* in more detail.

3.2 Static Code Rewriting

In *TaintStream*, a script submitted to the data platform is first passed through a static code rewriting phase, in which the script is instrumented to redirect the control flow to the translation functions.

Specifically, we first scan the script to locate the data processing pipelines. This can be achieved by parsing the script and searching the parsed syntax tree for the stream APIs described in §2.1. For instance, in PySpark (Python interface for Spark) scripts, the data processing operations can be located by finding the transformation functions invoked on Dataframe instances.

In the rewritten script, each data processing pipeline is wrapped with a translation function ϕ. The function takes the original lines of code (*i.e.*, the code to be translated at runtime) as input together with a context variable. The context variable includes the current data schema, types, variables, etc., which are used to provide necessary information for the dynamic code translation. We introduce a `getContext` function to collect the context information at runtime. Both the `getContext` function and the code translation function ϕ are packed as a library and linked to the script during running.

Table 2: Examples of dynamic code translation rules. $\phi(code)$ represents applying translation rules to the code. $\langle value, tag \rangle$ is to pack a value field and a taint tag field into a structured value field (the new field name is set to the same as the value field name). V_{Col} and T_{Col} are used to get the value and tag of the column respectively. $tagMerge(tag_1, tag_2, ...)$ and $tagAgg(tag)$ are to merge multiple taint tags or aggregate a group of tags using a customized merging function. $\perp$ represents the default tag for const/insensitive values.

Index	Original	Translated
①	$\phi(source)$	$source_{tagged}$
②	$\phi(df.transformation)$	$\phi(df).\phi(transformation)$
③	$\phi(select(Col))$	$select(\phi(Col))$
④	$\phi(drop(Col))$	$drop(\phi(Col))$
⑤	$\phi(orderBy(Col))$	$orderBy(V_{\phi(Col)})$
⑥	$\phi(filter(Col))$	$filter(V_{\phi(Col)})$
⑦	$\phi(join(df_2, on = Col))$	$join(\phi(df_2), on = V_{\phi(Col)})$
⑧	$\phi(union(df_2))$	$union(\phi(df_2))$
⑨	$\phi(withColumn(str, col))$	$withColumn(str, \phi(col))$
⑩	$\phi(groupBy(Col_1).agg(F_{agg_1}(Col_2), ...))$	$groupBy(V_{\phi(Col_1)}).agg(tagAgg(T_{\phi(Col_1)}), \phi(F_{agg_1}(Col_2), ...)).map(x \rightarrow (\langle x_1, x_2 \rangle, x_3, x_4, ...))$
⑪	$\phi(map(x \rightarrow (F_i(x_{i_1}, x_{i_2}, ...), F_j(x_{j_1}, x_{j_2}, ...), ...)))$	$map(x \rightarrow (\langle F_i(V_{x_{i_1}}, V_{x_{i_2}}, ...), tagMerge(T_{x_{i_1}}, T_{x_{i_2}}, ...) \rangle, \langle F_j(V_{x_{j_1}}, V_{x_{j_2}}, ...), tagMerge(T_{x_{j_1}}, T_{x_{j_2}}, ...) \rangle, ...))$
⑫	$\phi(reduce(a, b \rightarrow (G_i(a_i, b_i), ...)))$	$reduce(a, b \rightarrow (\langle G_i(V_{a_i}, V_{b_i}), tagMerge(T_{a_i}, T_{b_i}) \rangle, \langle ... \rangle, ...)$
⑬	$\phi(column\_name))$	$column\_name$
⑭	$\phi(const)$	$\langle const, \perp \rangle$
⑮	$\phi(Func(Col_1, Col_2, ...))$	$\langle Func(V_{\phi(Col_1)}, V_{\phi(Col_2)}, ...), tagMerge(T_{\phi(Col_1)}, T_{\phi(Col_2)}, ...) \rangle$
⑯	$\phi(F_{agg}(Col))$	$\langle F_{agg}(V_{\phi(Col)}), tagAgg(T_{\phi(Col)}) \rangle$

3.3 Dynamic Translation Rules

The static code rewriting phase sets up a proxy between the script and the original code interpreter. The actual code logic translation is completed dynamically when the modified script is being executed.

The core of the dynamic translation phase is the function ϕ that converts the original data processing pipeline to add the effect of taint propagation. The translation is based on a set of carefully designed rules as shown in Table 2. The rules are applied recursively on the stream APIs in a top-down manner, which starts from the overall pipeline (*e.g.*, $df.transformation*$) to the individual transformation operations (*select*, *groupBy*, etc.) and the arguments to the operations (*Col*, *Func*, etc.). Below shows how the first statement in Figure 2 is translated:

$\phi(emails.orderBy(\$date))$
$\rightarrow \phi(emails).\phi(orderBy(\$date))$ ..... *Rule* ②
$\rightarrow emails_{tagged}.orderBy(V_{\phi(\$date)})$ ..... *Rule* ①, *Rule* ⑤
$\rightarrow emails_{tagged}.orderBy(V_{\$date})$ ..... *Rule* ⑬

The rules are designed with two principles: non-interference and conservativeness. Non-interference means that the translated data processing pipeline must have the identical effect as the original script on the original data, so that the developers' tasks are correctly executed by *TaintStream*. We validate the non-interference of our translation rules through formal proofs [2], in which we try to demonstrate that the dataframe produced by the translated operation after removing taint tags is equivalent to the dataframe produced by the original operation.

[2]Details in our repository.

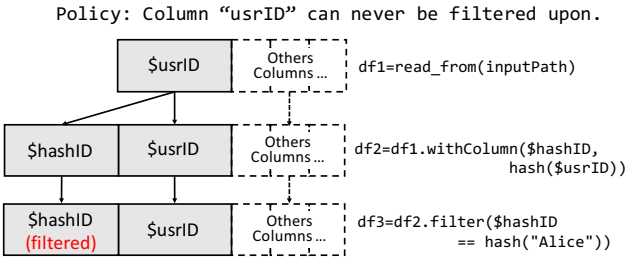


Figure 3: An example of the data flow graph built by *TaintStream*.

Second, we make sure that the taint tag of a cell is propagated to any cell that is computed from it. Similar to common data-flow analysis techniques, *TaintStream* is designed to ensure conservativeness, *i.e.*, any cell that carries sensitive information is marked as tainted, while there might be cells marked as tainted that are actually not sensitive (false positives). The conservativeness is guaranteed in the translation rules. First, similar to classic lattice-theoretic data-flow analysis approaches [26], the taint tag merging function is designed to be a conservative approximation to avoid false negatives produced during tag merging. Second, in each translation rule, all of the arguments that can possibly contribute to the output computation are passed into the tag merging function to produce the output tag. Specifically, for user-defined functions, we conservatively assume that the function return value is dependent on all arguments, no matter what the function actually does.

3.4 Handling Implicit Flow

Our dynamic translation rules are also to pass taint tags between the inputs and outputs of operations, while in some cases, sensitive information may be leaked to values that are not the direct output of the operation.

A typical example is the $filter$ operation, which scans the data and filters out the records that do not satisfy certain conditions. As a result, all the remaining records in the produced data would satisfy the specified conditions. Another example is the *sortBy* operation, which sorts the records in the dataframe according to certain key(s), and thus the first or last few records in the sorted dataframe would convey ranking information.

Such implicit information propagation operations are mostly performed on a whole column rather than on a value or a group of values. Thus, tracking such information flow in cell granularity may be inappropriate. Instead, we design a column-level data-flow graph (DFG) to track these operations.

Specifically, we maintain a graph for each dataframe to represent how each column in the dataframe is generated from raw data. As shown in Figure 3, the graph contains multiple layers, each of which represents a dataframe state. The last layer is the current dataframe and the first layer represents the original dataframe, and the middle layers are the intermediate dataframes. Each node in a layer represents a column and the edge between nodes represents

the dependency between cross-dataframe columns. For example, we can know that the $\$hashID$ column in Figure 3 is generated from the $\$usrID$ column in the original dataframe. If an implicit flow operation is performed, the corresponding column will be marked with the operation name (*e.g.*, `filter`, `sort`, etc.).

Various checking rules can be defined on such column-level DFGs to restrict implicit information flows. For example, in Figure 3, one can require that "the column user id can never be filtered upon", and the requirement can be checked by seeing whether the relevant columns are tagged with implicit operations. These checking rules are designed to prohibit hazardous data processing requests (*e.g.*, extracting information of a specific user as shown in Figure 3), which are independent from the taint tag-based privacy enforcement mechanisms that are designed to enforce general privacy policies.

4 IMPLEMENTATION

We implement the prototype of *TaintStream* on Spark [55] through the PySpark APIs. Both the Spark platform and its Python APIs are widely used in industry. We use *astroid* 2.4.2 [5], a library for python code static analysis, to parse and rewrite the scripts. The implementation contains ~4.9k lines of code. Other platforms can be easily supported by customizing the parser and operation translation rules accordingly. Here we further introduce our efforts on making *TaintStream* more robust and efficient.

Exception handling. *TaintStream* may fail in some cases, for example, when it encounters an unsupported operator or the translated code causes crashes (details in § 5.2). To ensure the completion of data processing operations, we handle the failures by sacrificing some precision. The high-level idea is to run the original operations on the untainted dataset and then re-taint the dataset conservatively. Specifically, when *TaintStream* catches a failure during executing a translated operation, it first calculates an upper-bound taint tag by merging all tags in the dataset before the operation, and untaints the dataset by removing all taint tags. Then we execute the original operation on the untainted dataset to get the untainted post-operation dataset. Finally, we recover the taint tags in the dataset using the upper-bound taint tag. With this mechanism, *TaintStream* can robustly deal with complicated and unseen scripts, although the precision may be sacrificed in rare cases. The overhead of exception handling is minimal ($O(n)$ where n is the number of tags) as compared with a typical data processing task.

Performance optimization. Inspired by compiler optimization techniques [1, 42], we implement two key operation fusion rules to reduce redundant computations. (1) When the translated code matches the pattern of $V_{\langle value,tag\rangle}$ or $T_{\langle value,tag\rangle}$, *TaintStream* simply translates the corresponding code snippets to *value* and *tag*. This optimization can reduce redundant operations to pack and get value/tag. (2) Another optimization is applied when *TaintStream* finds nested *tagMerge* operations in the translated code. We implement *tagMerge* as a function accepting arbitrary numbers of parameters, so nested calls can be simplified to a single call of *tagMerge*. For example, suppose a code snippet in the translated code is $tagMerge(Tag_1, tagMerge(Tag_2, Tag_3))$, *TaintStream* will optimize the code snippet to $tagMerge(Tag_1, Tag_2, Tag_3)$. These two optimizations effectively reduce 24% time overhead on average. More advanced optimization rules will be added in the future to further improve the performance.

5 EVALUATION

Our evaluation addresses the following research questions:

(1) What is the accuracy of *TaintStream* on taint tracking? How does it compare with baselines? (§5.2)
(2) What is the system overhead of *TaintStream* on running time and storage? (§5.3)
(3) How can *TaintStream* support real-world data management tasks in big data platforms? (§5.4)

5.1 Experiment Setup

We first introduce how we set up our experiments, including benchmarks, baselines, and the experimental environment.

Benchmarks. There are benchmark suites designed for measuring the performance of data processing [44] or evaluating the accuracy of taint analysis in Web/Android applications [3, 41]. However, to the best of our knowledge, there is no existing benchmark suite tailored for big data taint analysis tasks. Thus, we decide to create our own benchmark suite to evaluate the accuracy of *TaintStream*.

We construct two test suites based on a set of real-world data processing scripts obtained from an industry data platform (the platform managed by *Team A* as we mentioned before). The first one is called **CellBench**, which contains 33 hand-crafted data processing scripts that are abstracted from the real-world scripts by extracting the common operations and custom functions from the original scripts and combining them into executable data processing pipelines. The pipelines in CellBench are similar to real-world scripts, while the operations are simplified in order to run on fake data (fake data generated with Fake [17]). The scripts are grouped into six categories, each of which is designed to evaluate a representative class of operations, such as operations that group the data (*GroupBy*), operate on structured data (*Structured Data*), or use user-defined functions (*UDF*). The second one is **ProBench**, which contains 7 real-world scripts that can be executed on a subset of data that we have access to (the authors' personal data). The other scripts are not used because they rely on end-user datasets that we do not have access to. CellBench is used to evaluate the accuracy of taint analysis because we can automatically generate the ground truth by customizing the operations and datasets. ProBench is primarily used to measure the overhead of *TaintStream* since its scripts and datasets are more realistic.

We use a value-based approach to generate the ground truth in CellBench. Specifically, we randomly pick some cells from the input dataset and add a unique property to the cell values. For example, the string-typed cells are attached with a special string and the numerical cells are added a huge number, indicating the cells are tainted. The data processing operations in CellBench are designed to keep the taint properties when propagating information. For example, only string concatenation and number addition are allowed when generating new cells with existing cells. Thus the output dataset will automatically carry the ground truth taint information, *i.e.*, a cell should be tainted only if its value contains the predefined taint property. Figure 4 shows an example, in the

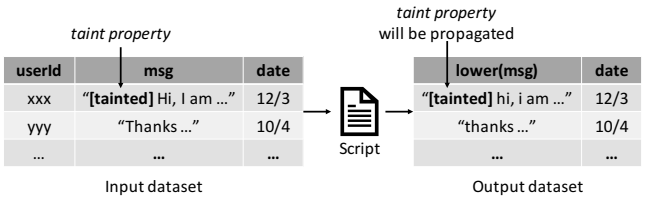


Figure 4: A simplified test case in CellBench. Each test case includes a script, an input dataset, and a ground-truth output dataset.

input dataset, the first msg field is tainted and its string value is inserted a "[tainted]" substring in the beginning. After processing, suppose the msg field goes through a lower operation that converts all letters to the lower case, the cells that contain the "[tainted]" substring should be identified as tainted cells.

Baselines. To evaluate the taint tracking accuracy, we choose two baselines to compare against. The first one is *PySa* [16], a generic static taint analysis framework for Python code developed by Facebook. *PySa* is designed to detect information flow between variables in code, thus it can only support dataset-level tracking, *i.e.*, analyzing a dataset is tainted or not. The second baseline is *PlanAnalyzer*, which builds a column-level data flow graph (DFG) based on the Spark logical execution plan [2]. The latter is generated by Spark at runtime to decide how to execute the data processing pipeline. The DFG created from *PlanAnalyzer* is similar to the one shown in Figure 3. An output column is considered tainted if there is a flow from a tainted input column to the output column. We do not compare with other existing approaches [12, 34, 43] mainly due to the unavailability of their source code [12, 34] or the difficulty in porting them to our benchmark [43].

In the case study (§5.4), the baselines are the current approaches of *Team A* to three common data management tasks. We qualitatively compare *TaintStream* with current approaches to show its usefulness.

Experimental environment. The experiments are conducted on a cluster with four worker nodes and each node is equipped with an Intel Xeon E5-2665 processor, 128GB memory, and 3TB storage. The server runs a 64-bit Ubuntu 18.04. The Spark version is 2.4.3 built on JAVA 1.8 and Scala 2.11. The Python version is 3.7.6.

5.2 Accuracy of Taint Tracking

We use the scripts and datasets in CellBench to measure the taint tracking accuracy of *TaintStream* and other baselines. To simplify the measurement, we consider the taint tags to be tracked are booleans indicating whether each cell is sensitive. 50% random cells in 20% random columns (*i.e.*, 10% of all cells) in the input datasets are marked as sensitive (carrying "True" tags), and the taint tracking accuracy is measured by examining whether the sensitive columns/cells in the output datasets are correctly labeled with the "True" tag. Specifically, the precision is measured by $\frac{\#TP}{\#TP+\#FP}$ and the recall is $\frac{\#TP}{\#TP+\#FN}$.

Since the baselines do not support cell-level tracking, we additionally design a column-tracking experiment to compare *TaintStream* with the baselines. A column is considered tainted if any of its cells is tainted. We measure how many columns in the output datasets are correctly labeled for each tracking method.

Table 3: The taint analysis accuracy of *TaintStream* and baselines on CellBench. The * sign represents that exception handling is triggered on the corresponding script.

Script Name	Column Level			Cell Level	
	PySa	*PlanAnalyzer*	*TaintStream*	*TaintStream*	
	correct/total			precision	recall
Basic					
orderBy_1	4/6	6/6	6/6	100.0%	100.0%
orderBy_2	1/3	3/3	3/3	100.0%	100.0%
select_1	3/6	6/6	6/6	100.0%	100.0%
select_2	1/2	2/2	2/2	100.0%	100.0%
withColumn_1	3/3	3/3	3/3	100.0%	100.0%
withColumn_2	3/5	5/5	5/5	100.0%	100.0%
withColumn_3	3/3	3/3	3/3	100.0%	100.0%
GroupBy					
count_1	1/3	3/3	3/3	100.0%	100.0%
count_2	2/3	3/3	3/3	100.0%	100.0%
count_3	2/5	5/5	5/5	100.0%	100.0%
count_4	2/4	4/4	4/4	100.0%	100.0%
statistics_1	4/5	5/5	5/5	100.0%	100.0%
statistics_2	3/5	5/5	5/5	100.0%	100.0%
Join					
inner_join_1	2/4	4/4	4/4	100.0%	100.0%
inner_join_2	3/5	5/5	5/5	100.0%	100.0%
inner_join_3	2/5	5/5	5/5	100.0%	100.0%
left_join	3/5	5/5	5/5	100.0%	100.0%
right_join	3/5	5/5	5/5	100.0%	100.0%
outer_join	3/5	5/5	5/5	100.0%	100.0%
Structured Data					
array_type_1 *	4/5	5/5	4/5	80.0%	100.0%
array_type_2	2/3	3/3	3/3	100.0%	100.0%
struct_type_1	2/5	1/5	4/5	100.0%	100.0%
struct_type_2	2/3	2/3	3/3	100.0%	100.0%
struct_type_3 *	3/4	2/4	3/4	66.7%	100.0%
UDF					
udf_1	1/4	4/4	4/4	100.0%	100.0%
udf_2	1/1	1/1	1/1	100.0%	100.0%
udf_3	0/1	0/1	0/1	0.0%	100.0%
class_udf_1	2/2	2/2	2/2	100.0%	100.0%
class_udf_2	1/1	1/1	1/1	100.0%	100.0%
Map Reduce					
map_reduce_1	0/2	0/2	2/2	100.0%	100.0%
map_reduce_2	1/4	0/4	4/4	100.0%	100.0%
map_reduce_3	1/4	0/4	4/4	100.0%	100.0%
map_reduce_4 *	1/4	0/4	1/4	21.9%	100.0%
Summary	69/125	103/125	118/125	93.0%	100.0%

The detailed results are shown in Table 3.

Column tracking results. We observe that *TaintStream* correctly labels all the columns in *Basic* (28/28), *GroupBy* (25/25), and *Join* (29/29) categories and gets more correct results on other categories compared to the baselines. It labels wrong tags on some columns mainly for two reasons, *i.e.*, exception handling mechanism and missing of the dataflow in UDFs. We will explain them in detail in the more strict cell tracking experiments.

PlanAnalyzer gains comparable results in *Basic*, *GroupBy*, *Join*, and *UDF* categories. However, its accuracy is worse in *Structured Data* category and poor in *Map Reduce* category. For the *Structured Data* category, *PlanAnalyzer* treats structured columns as a whole due to the non-trivial cost on achieving field-sensitive in the DFG. For the *Map Reduce* category, *PlanAnalyzer* cannot determine the column dependencies for the map/reduce operations because it cannot infer the output schema from the Spark execution plan.

PySa is worse than the other two tools in all categories. This is reasonable because, as a generic taint analysis framework, *PySa* is column-agnostic and supports only dataset-level tracking. Consequently, it gets many false positives results.

Cell tracking results. Given that baselines are worse than *TaintStream* in the column tracking experiment and they do not support a cell-level tracking, their results cannot be improved in the more strict cell tracking experiment. As a result, we report only *TaintStream*'s results.

As shown in Table 3, *TaintStream* achieves 100% recall on all scripts, this is because we carefully design the translation rules to ensure soundness (refer to §3.3). *TaintStream* achieves a high precision of 93.0% on average. The false-positive results come from two reasons, *i.e.*, the exception handling mechanism and the missing of data flow in UDFs. *TaintStream* successfully handles exceptions such as unsupported operators, missing taint tags, etc. For example, in script "struct_type_3", a "from_json" function that accepts a JSON string as the parameter is invoked in the pipeline, *TaintStream* cannot determine each field's taint tag and thus throws an exception. For the second reason, *TaintStream* conservatively merges all the parameters' tags without analyzing if there is a flow from UDF's parameter to the return value. For example, in script "udf_3", a UDF takes a tainted column as a parameter but never uses it.

Conclusion. From the two experiments, we can conclude that *TaintStream* can not only achieve more accurate results compared to baselines at column level but also track data at cell level with a precision of 93.0% and perfect recall. Handling data flow in UDFs and reducing exception triggered are two of the optimization directions.

5.3 System Overhead

The overhead of *TaintStream* is measured with real-world scripts in ProBench. Specifically, we run each script in ProBench with or without *TaintStream* enabled and compare their running time and storage space. Similar to the previous experiments, we consider the boolean taint tags to simplify the measurement (cases that use more complicated tags are also measured but reported in the case study for clearer organization). For time assessment, we measure the end-to-end running time, including reading, processing, and writing, and the results are shown in Figure 5. For storage assessment, we compare the tag size with the original output size and the results are shown in Table 4.

Running time. As shown in Figure 5, with the increase of the time spent by the original scripts, the running time of *TaintStream* increases proportionally. *TaintStream* is slower than the original scripts by 12.7% on average, with a maximum value of 17.0% (on the script "action provider"). The input data of this script contains many complicated structured columns. Propagating taint tags in these columns requires more operations (refer to §3) because *TaintStream* needs to first package a sub-column's value and tag and then package this sub-column to its parent column.

We further break down to investigate what are the main reasons for the overhead. The overhead can be divided into three parts, *i.e.*, static code rewriting, dynamic translation, and injected taint propagation logic. Among them, the former two can be precisely measured. The results show that *TaintStream* spends 0.28s on static code rewriting and 0.87s on dynamic translation on average. Although they are slightly influenced by the length of the script, the maximum value does not exceed three seconds given a script of around 300 lines of code. Given that the time spent on taint propagation could increase proportionally with the growth of the input data, we can conclude that *TaintStream*'s overhead mainly comes from the taint propagation and the time spent on static code rewriting and dynamic translation is negligible.

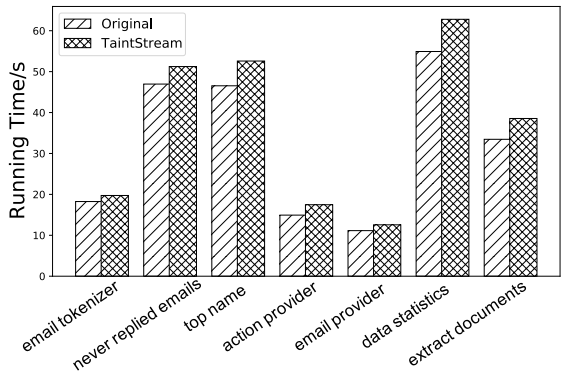


Figure 5: *TaintStream*'s overhead on running time.

Table 4: The storage overhead of *TaintStream*. We use x to represent the average size of a taint tag. The "overhead" column reports the overhead when the average tag size is 1 byte.

Script Name	Original Output Size (Byte)	Tag Size	Overhead
email tokenizer	2,697,017	19,952x	0.74%
never replied emails	246,129	5,000x	2.03%
top name	16,229	153x	0.94%
action provider	32,073	636x	1.98%
email provider	10,246,702	44,275x	0.43%
data statistics	633	25x	3.95%
extract documents	1,097	48x	4.38%

According to the results, *TaintStream* introduces a 12.7% overhead on the running time on average. Considering the time and manpower saved for various stakeholders, as we will show in §5.4, this cost is rather acceptable.

Storage. By comparing the tag size and the original output size in Table 4, we derive the following observations.

The size of the taint tags and the original output are positively correlated. This is reasonable because *TaintStream* tracks data at the cell level. When we set the taint tag to a boolean value, the storage overhead is 2.06% on average. The overhead would grow larger if more advanced taint tags are adopted. We believe that the overhead can be further decreased after compression, because taint tags may contain many duplicated values.

Conclusion. We test *TaintStream* on ProBench and compare the results with the original scripts in terms of running time and storage. The results show that the *TaintStream* introduces a 12.7% overhead on the running time and 2.06% overhead on the storage. Considering the manpower saved, this overhead is rather acceptable.

5.4 Case Study

Finally, we show how *TaintStream* can be used to enforce three common and important privacy policies, including data retention,

Table 5: Comparison between *TaintStream* and the current practice on three data management tasks.

Task	Approach	Storage Overhead	Running Time Overhead	Code Review	Extra efforts for data managers and data analysts	Granularity
Data Retention	Current practice	-	-	No	Data managers organize the raw data by date and maintain a dataset linage graph that records the relationship between datasets. Once the raw data expire, all related datasets will be deleted. No requirements for data analysts.	dataset level
	TaintStream	3.19%	10.47%	No	Data managers formulate the privacy policy in *TaintStream* format.	cell level
Access Control	Current practice	-	-	Yes	Data managers assign storage space with different access permissions to data analysts. Reading from or writing to an unauthorized space is prohibited through a process of code review.	dataset level
	TaintStream	3.97%	16.67%	No	Data managers formulate the privacy policy in *TaintStream* format.	cell level
User data Erasure	Current practice	-	-	Yes	Data managers maintain a data linage graph to track the data movement. Once a user opts to delete her data, all the records (rows) in related datasets will be deleted according to the user identifier. Data analysts are required to properly maintain and propagate the user identifier column. This will be enforced by a process of code review.	row level
	TaintStream	2.56%	15.69%	No	Data managers formulate the privacy policy in *TaintStream* format.	cell level

access control, and user data erasure. We qualitatively compare the solutions enabled by *TaintStream* with the current solutions in *Team A* to demonstrate its advantages.

Data retention. In current practice, data managers organize the raw data by date and the platform maintains a data linage graph that records the input/output relationship between datasets. Once some of the raw data expires, all related datasets will be deleted by a sweeper. For *TaintStream*, a feasible policy can be defined as follows.

> **Tag type**: Integer T. # timestamp of the expiration date
> **Tag init**: : Set T to the time when the data expires.
> **Tag merge**: $Minimum(T_1, T_2)$
> **Enforcement**: Periodically scan the datasets and delete any record x if the current timestamp is behind its timestamp T_x.

Access control. In current practice, data analysts are provided storage spaces in different confidentiality levels. They also have different access rights to these storage spaces. Any reading/writing access to an unauthorized space will be prohibited. They are also required to store confidential data in certain spaces only unless the data is sanitized, which is enforced by code review. The control is based on dataset level. *TaintStream* could achieve a more fine-grained control by defining the following policy.

> **Tag type**: Set of identifiers S. # set of authorized analysts
> **Tag init**: : $S_x \leftarrow$ the analysts with access to the raw data x.
> **Tag merge**: $Intersect(S_1, S_2)$
> **Enforcement**: A data record x is only visible to the analysts in its authorized analysts set S_x.

User data erasure. In current practice, data managers build a data linage graph same as the one in the data retention graph. Data analysts are required to keep the user ID of each record in the dataset, which will be enforced by the code review. Once a user opts to delete her data, all the records (rows) in related datasets will be deleted according to the user ID. With *TaintStream*, we can define the following policy so that data analysts no longer need to maintain the user ID in the datasets.

> **Tag type**: Set of identifiers S. # set of involved data providers
> **Tag init**: : $S_x \leftarrow$ the provider the raw data x
> **Tag merge**: $Union(S_1, S_2)$
> **Enforcement**: If a user u asks to be forgotten, delete any record x whose tag S_x contains u.

Since taint tags in the user data erasure task are sets of identifiers and the merge function is set union, the taint tag size may grow infinitely. To avoid this issue, we set the tag to "all" once the set size is larger than a threshold (*e.g.*, 100). Also, to save computational cost, the user erasure requests are processed in batches periodically.

Comparison. We qualitatively compare *TaintStream* with current practice and summarize the results in Table 5.

(1) **Additional code review**. In current practice, two out of three tasks need a code review, which takes hours or days before the script is approved to be executed in the platform. (2) **Extra manual efforts and restrictions**. To support these tasks, data managers take non-trivial efforts, including developing specialized features and organizing raw data properly. Data analysts are also restricted by many rules when writing scripts. (3) **Coarse granularity**. Current practice usually works in a coarse granularity. For example, in the data retention task, a dataset will be deleted once it contains an expiration record even if most of the records do not expire. These non-expiration data are falsely deleted, causing a waste of valuable computational resources.

By contrast, *TaintStream* realizes these tasks more automatically. No code review is required and no restrictions for data analysts. Data managers only need to formulate the policy in *TaintStream* format and a fine-grained result is reported. To realize the tasks, *TaintStream* brings a less than 4% overhead on the storage and an averaged 14.23% overhead on the running time. This is a rather acceptable proportion considering the convenience and saved efforts for various stakeholders as aforementioned.

6 RELATED WORK

Our work is close to two classes of research.

Taint analysis is a widely studied technique to track information flow in a program. It can be done statically or dynamically. The static approach is performed on the source code or byte code without executing the program, while the dynamic taint tracking operates during programming execution.

Taint analysis has been implemented for various platforms, including Android [4, 13, 23, 42, 52, 54], JavaScript [6, 10, 22], WebAssembly [20], etc. Among them, *TaintART* [42] adopted similar design ideas as *TaintStream*, which implemented an instrumented compiler that inserts code blocks to handle taint propagation between variables. As explained in §2.3, these approaches can hardly

be ported to big data platforms due to the lack of traceable information and the complicated storage and computation mechanisms.

There are also several taint tracking solutions proposed for data management systems. Schütte *et al.* [34] introduced a data usage control system, which models a query script as a column-level data flow graph similar to the one in Figure 3. *DBTaint* [12] was designed for dynamic taint tracking in SQL engines, which extended the data types in databases and modifies some SQL operations to maintain the extended data types. *FLOWDEBUG* [43] proposed to analyze fine-grained data provenance by extending the Scale type system and inserting custom data abstractions through source-to-source transformation. *TaintStream* is conceptually similar to these approaches (especially *FLOWDEBUG*), while having several differences: (1) *TaintStream* is a generic framework that can serve various privacy compliance tasks, while the prior approaches were designed for other purposes (*e.g.*, debugging) or specific scenarios (*e.g.*, access control). (2) *TaintStream* is able to perform non-intrusive taint tracking with no modification to the legacy systems like Spark runtime or Scala type system, while prior work needs to integrate extra modules to the systems. (3) *TaintStream* is tailored for Stream-like data processing engines that have more complicated and flexible operations than traditional SQL, such as *map*/*reduce*, customized functions, etc.

Privacy policy enforcement. There are many other approaches related to automated privacy policy enforcement.

Many researchers have attempted to formulate privacy policies in machine-readable formats. For example, Tschantz *et al.* [45] provided semantics of purpose restrictions to describe whether an action is for a purpose or not. Chowdhury *et al.* [9] presented a policy specification language which can capture the privacy requirements of HIPAA. Wang *et al.* [47] introduced a formal policy language that can govern how the data is processed. We also have a specific format for policy definition. These formats are designed for different purposes and checking mechanisms, and there isn't any policy format that is universally agreed upon today.

Based on the formalized privacy policies, privacy compliance can be examined by analyzing the program behavior and checking its consistency with the policies. There have been a great deal of methods in tracking or restricting information flows in programs, including language-based [18, 32, 33], role-based [8, 30, 31, 38], and purpose-based [14] approaches. To facilitate privacy risk assessment and privacy compliance checking, researchers have also proposed new programming languages and frameworks that can weave privacy policies into the code [21, 33, 53] or simplify the analysis of program behaviors [15, 28]. For big data platforms, existing privacy compliance checking approaches are either tailored for a specific type of privacy requirements such as access control [36], or based on a new framework or paradigm that requires non-trivial adaption efforts from developers [47]. *TaintStream* is able to support flexible and fine-grained policy enforcement without any manual modification to the analytic script and runtime engine.

7 CONCLUSION AND DISCUSSION

We have presented *TaintStream*, a light-weight fine-grained taint tracking framework for Spark-like big data platforms. By customizing the injected taint propagation logic, *TaintStream* is able to fulfill various privacy protection requirements such as data retention, access control, user data erasure, etc. Experiments on a self-built benchmark and several real-world cases show that *TaintStream* is able to achieve accurate cell-level tracking with 93.0% precision and 100% recall and is able to flexibly support various privacy requirements with acceptable overhead.

A potential obstacle to adopting *TaintStream* in practice is the over-conservative taint propagation, *i.e.*, taint tags may be propagated to many irrelevant cells, even the whole dataset, by insensitive operations (*e.g.*, `count`, `mean`, etc.). This may also lead to huge storage overhead if the taint tag size is large. A possible countermeasure is to define "exception" operations in which the taint tags should not be propagated. Another limitation of *TaintStream* is that the tracking mechanisms may be evaded by intentional data analysts, *e.g.*, by obfuscating the scripts or using complicated custom functions. Although in most cases the analysts are honest, we believe some mechanisms to regulate code style and/or detect abnormal code would be helpful in the future.

ACKNOWLEDGMENT

We thank all anonymous reviewers for the valuable feedback. This work was partly supported by the National Key Research and Development Program of China under the grant number 2020YFB2104100, the National Natural Science Foundation of China under the grant number 61725201, the Beijing Outstanding Young Scientist Program under the grant number BJJWZYJH01201910001004, and PKU-Baidu Fund Project under the grant number 2020BD007. Mengwei Xu was partly supported by National Industrial Internet Innovation and Development Project (No.TC190A3X1). For any questions about the code and data, please contact Yuanchun Li.

REFERENCES

[1] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. 1986. Compilers, principles, techniques. *Addison wesley* 7, 8 (1986), 9.

[2] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. 2015. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*. 1383–1394. https://doi.org/10.1145/2723372.2742797

[3] Steven Arzt. 2021. DroidBench 2.0. https://github.com/secure-software-engineering/DroidBench. Accessed February 4th, 2021.

[4] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices* 49, 6 (2014), 259–269. https://doi.org/10.1145/2594291.2594299

[5] Python Code Quality Authority. 2021. Astroid's documentation. http://pylint.pycqa.org/projects/astroid/en/latest/. Accessed February 25, 2021.

[6] Abhishek Bichhawat, Vineet Rajani, Deepak Garg, and Christian Hammer. 2014. Information flow control in WebKit's JavaScript bytecode. In *International Conference on Principles of Security and Trust*. Springer, 159–178.

[7] Muhammad Bilal, Lukumon O Oyedele, Junaid Qadir, Kamran Munir, Saheed O Ajayi, Olugbenga O Akinade, Hakeem A Owolabi, Hafiz A Alaka, and Maruf Pasha. 2016. Big Data in the construction industry: A review of present status, opportunities, and future trends. *Advanced engineering informatics* 30, 3 (2016), 500–521. https://doi.org/10.1016/j.aei.2016.07.001

[8] Niklas Broberg and David Sands. 2010. Paralocks: role-based information flow control and beyond. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on principles of programming languages*. 431–444.

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

[10] Ravi Chugh, Jeffrey A Meister, Ranjit Jhala, and Sorin Lerner. 2009. Staged information flow for JavaScript. In *Proceedings of the 30th ACM SIGPLAN conference on programming language design and implementation*. 50–62.

[11] James Clause, Wanchun Li, and Alessandro Orso. 2007. Dytan: a generic dynamic taint analysis framework. In *Proceedings of the 2007 international symposium on software testing and analysis*. 196–206.

[12] Benjamin Davis and Hao Chen. 2010. DBTaint: Cross-Application Information Flow Tracking via Databases. *WebApps* 10 (2010), 12.

[13] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. 2014. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)* 32, 2 (2014), 1–29.

[14] Tomoya Enokido and Makoto Takizawa. 2011. Purpose-based information flow control for cyber engineering. *IEEE Transactions on Industrial Electronics* 58, 6 (2011), 2216–2225.

[15] Michael D Ernst, René Just, Suzanne Millstein, Werner Dietl, Stuart Pernsteiner, Franziska Roesner, Karl Koscher, Paulo Barros Barros, Ravi Bhoraskar, Seungyeop Han, et al. 2014. Collaborative verification of information flow for a high-assurance app store. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. 1092–1104.

[16] Facebook. 2021. PySa Overview. https://pyre-check.org/docs/pysa-basics/. Accessed February 4th, 2021.

[17] Daniele Faraglia. 2021. Welcome to Faker's documentation! https://faker.readthedocs.io/en/master/. Accessed February 4th, 2021.

[18] Andrew Ferraiuolo, Rui Xu, Danfeng Zhang, Andrew C Myers, and G Edward Suh. 2017. Verification of a practical hardware security architecture through static information flow analysis. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. 555–568. https://doi.org/10.1145/3037697.3037739

[19] Apache Software Foundation. 2021. Apache Storm. https://storm.apache.org/. Accessed December 16, 2020.

[20] William Fu, Raymond Lin, and Daniel Inge. 2018. Taintassembly: Taint-based information flow control tracking for webassembly. *arXiv preprint arXiv:1802.01050* (2018).

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

[22] Salvatore Guarnieri, Marco Pistoia, Omer Tripp, Julian Dolby, Stephen Teilhet, and Ryan Berg. 2011. Saving the world wide web from vulnerable JavaScript. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. 177–187.

[23] Wei Huang, Yao Dong, Ana Milanova, and Julian Dolby. 2015. Scalable and precise taint analysis for android. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. 106–117. https://doi.org/10.1145/2771783.2771803

[24] Priyank Jain, Manasi Gyanchandani, and Nilay Khare. 2016. Big data privacy: a technological perspective and review. *Journal of Big Data* 3, 1 (2016), 1–25. https://doi.org/10.1186/s40537-016-0059-y

[25] Simon Holm Jensen, Magnus Madsen, and Anders Møller. 2011. Modeling the HTML DOM and browser API in static analysis of JavaScript web applications. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 59–69.

[26] Gary A Kildall. 1973. A unified approach to global program optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 194–206.

[27] Jay Kreps, Neha Narkhede, Jun Rao, et al. 2011. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, Vol. 11. 1–7.

[28] Yuanchun Li, Fanglin Chen, Toby Jia-Jun Li, Yao Guo, Gang Huang, Matthew Fredrikson, Yuvraj Agarwal, and Jason I. Hong. 2017. PrivacyStreams: Enabling Transparency in Personal Data Processing for Mobile Apps. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 1, 3, Article 76 (Sept. 2017), 26 pages. https://doi.org/10.1145/3130941

[29] Abid Mehmood, Iynkaran Natgunanathan, Yong Xiang, Guang Hua, and Song Guo. 2016. Protection of big data privacy. *IEEE access* 4 (2016), 1821–1834.

[30] Andrew C Myers and Barbara Liskov. 2000. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 9, 4 (2000), 410–442.

[31] Shigenari Nakamura, Dilewaer Doulikun, Ailixier Aikebaier, Tomoya Enokido, and Makoto Takizawa. 2014. Role-based information flow control models. In *2014 IEEE 28th International Conference on Advanced Information Networking and Applications*. IEEE, 1140–1147. https://doi.org/10.1109/AINA.2014.139

[32] François Pottier and Vincent Simonet. 2003. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 25, 1 (2003), 117–158.

[33] Andrei Sabelfeld and Andrew C Myers. 2003. Language-based information-flow security. *IEEE Journal on selected areas in communications* 21, 1 (2003), 5–19.

[34] Julian Schütte and Gerd Stefan Brost. 2016. A data usage control system using dynamic taint tracking. In *2016 IEEE 30th International Conference on Advanced Information Networking and Applications (AINA)*. IEEE, 909–916. https://doi.org/10.1109/AINA.2016.127

[35] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *2010 IEEE symposium on Security and privacy*. IEEE, 317–331.

[36] Shayak Sen, Saikat Guha, Anupam Datta, Sriram K Rajamani, Janice Tsai, and Jeannette M Wing. 2014. Bootstrapping privacy compliance in big data systems. In *2014 IEEE Symposium on Security and Privacy*. IEEE, 327–342. https://doi.org/10.1109/SP.2014.28

[37] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*. Ieee, 1–10.

[38] Xiaodan Song, Yun Chi, Koji Hino, and Belle L Tseng. 2007. Information flow modeling based on diffusion rate for prediction and ranking. In *Proceedings of the 16th international conference on World Wide Web*. 191–200.

[39] Apache Spark. 2021. Document of PySpark SQL module. http://spark.apache.org/docs/latest/api/python/pyspark.sql.html. Accessed February 21, 2021.

[40] Manu Sridharan, Shay Artzi, Marco Pistoia, Salvatore Guarnieri, Omer Tripp, and Ryan Berg. 2011. F4F: taint analysis of framework-based web applications. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*. 1053–1068.

[41] Stanford. 2021. Securibench Micro. https://github.com/too4words/securibench-micro. Accessed February 4th, 2021.

[42] Mingshen Sun, Tao Wei, and John CS Lui. 2016. Taintart: A practical multi-level information-flow tracking system for android runtime. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 331–342. https://doi.org/10.1145/2976749.2978343

[43] Jason Teoh, Muhammad Ali Gulzar, and Miryung Kim. 2020. Influence-based provenance for dataflow applications with taint propagation. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. 372–386.

[44] TPC. 2021. TPCx-BB is a Big Data Benchmark. http://www.tpc.org/tpcx-bb/. Accessed December 17, 2020.

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

[46] Kai Wang, Aftab Hussain, Zhiqiang Zuo, Guoqing Xu, and Ardalan Amiri Sani. 2017. Graspan: A single-machine disk-based graph system for interprocedural static analyses of large-scale systems code. *ACM SIGARCH Computer Architecture News* 45, 1 (2017), 389–404. https://doi.org/10.1145/3037697.3037744

[47] Lun Wang, Joseph P Near, Neel Somani, Peng Gao, Andrew Low, David Dao, and Dawn Song. 2019. Data capsule: A new paradigm for automatic compliance with data privacy regulations. In *Heterogeneous Data Management, Polystores, and Analytics for Healthcare*. Springer, 3–23. https://doi.org/10.1007/978-3-030-33752-0_1

[48] Fengguo Wei, Xingwei Lin, Xinming Ou, Ting Chen, and Xiaosong Zhang. 2018. Jn-saf: Precise and efficient ndk/jni-aware inter-language static analysis framework for security vetting of android applications with native code. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 1137–1150. https://doi.org/10.1145/3243734.3243835

[49] Wikipedia. 2021. Children's Online Privacy Protection Act. https://en.wikipedia.org/wiki/Children%27s_Online_Privacy_Protection_Act. Accessed February 13, 2021.

[50] Wikipedia. 2021. General Data Protection Regulation. https://en.wikipedia.org/wiki/General_Data_Protection_Regulation. Accessed February 13, 2021.

[51] Wikipedia. 2021. Health Insurance Portability and Accountability Act. https://en.wikipedia.org/wiki/Health_Insurance_Portability_and_Accountability_Act. Accessed February 13, 2021.

[52] Lok Kwong Yan and Heng Yin. 2012. Droidscope: Seamlessly reconstructing the OS and dalvik semantic views for dynamic android malware analysis. In *21st USENIX Security Symposium (USENIX Security 12)*. 569–584.

[53] Jean Yang, Kuat Yessenov, and Armando Solar-Lezama. 2012. A language for automatically enforcing privacy policies. *ACM SIGPLAN Notices* 47, 1 (2012), 85–96.

[54] Zhemin Yang and Min Yang. 2012. Leakminer: Detect information leakage on android with static taint analysis. In *2012 Third World Congress on Software Engineering*. IEEE, 101–104.

[55] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, Ion Stoica, et al. 2010. Spark: Cluster computing with working sets. *HotCloud* 10, 10-10 (2010), 95.