TaintSQL: Dynamically Tracking Fine-Grained Implicit Flows for SQL Statements

Wei Lin*†, Lu Zhang*†, Haotian Zhang‡, Kailai Shao‡, Mingming Zhang§, and Tao Xie*†
*Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, China
†School of Computer Science, Peking University, Beijing, China
‡Ant Group Co., Ltd., Hangzhou, China
§Advanced Institute of Information Technology, Peking University, Hangzhou, China
linwei@stu.pku.edu.cn, {zhanglucs, taoxie}@pku.edu.cn, {jingyun.zht, kailai.skl}@antgroup.com, mmzhang@aiit.org.cn

***Abstract*—To address software engineering tasks such as security risk assessment, software change government, and access control in database applications, taint analysis approaches for SQL statements have been commonly adopted for tracking information flows in these applications. However, existing taint analysis approaches cannot track implicit flows (i.e., control dependencies between sources and sinks) for SQL statements, facing the challenges of native/unmanaged code and database management system (DBMS) complexity. To address these challenges, in this paper, we propose TaintSQL, a cell-level dynamic taint analysis (DTA) framework (maintaining a taint tag for each table cell) to track fine-grained implicit flows for SQL statements. Our TaintSQL framework includes two novel techniques, namely MutaIF and MockIF. MutaIF aims to track implicit flows with causal relationships, whereas MockIF aims to dynamically track implicit flows at runtime. We implement the two techniques of TaintSQL and evaluate them on a set of test subjects to assess their effectiveness and efficiency. The evaluation results show that both techniques effectively track fine-grained implicit flows for SQL statements with reasonable runtime overhead. The F1 scores of MutaIF and MockIF are 96.2% and 97.9%, respectively. We also conduct an industrial study of MutaIF in an international IT company (which serves over *1 billion* global users and *80 million* merchants). The positive feedback from the software engineers also demonstrates the practicability of the TaintSQL framework and the MutaIF technique in industrial settings.**

***Index Terms*—implicit flow, taint analysis, SQL statement**

I. Introduction

Nowadays there is an increasing demand for tracking information flows in database applications to address various software engineering tasks such as security risk assessment [1], software change government [2], and access control [3]. Addressing such tasks is of great significance for software engineers to ensure that database applications provide secure, expected, and reliable services to their users. To track information flows in database applications, taint analysis [4], [5] (also referred to as information flow tracking) marks all or some program inputs as tainted (i.e., sources), and then propagates taint tags to check whether they reach the target area (i.e., sinks).

In database applications, there are two critical types of information flows (i.e., *explicit flows* [6] and *implicit flows* [7]) for SQL statements. First, explicit flows (also known as data flows) occur due to assignments or arithmetic operations that directly pass information from sources to sinks. For example, there are explicit flows from each program input in the `VALUES` clause of an `INSERT` statement to the corresponding newly inserted table cells' values. There are also explicit flows from the fetched table cells' values in the database state to the corresponding values in the result set returned by a `SELECT` statement. Second, implicit flows are indirect passages of information between values typically as a result of condition branches or column names. Unlike an explicit flow, each sink's value in an implicit flow is not directly computed from the corresponding source's value. Instead, the sinks' values in implicit flows are control-dependent on the sources' values. For example, condition branches in the `WHERE` clause of a SQL statement affect whether the records should be fetched or the table cells' values should be updated. Recent studies [8], [7] have demonstrated the importance of tracking implicit flows. Tracking only explicit flows without implicit flows may result in critical data relationships being lost, causing under-tainting.

To track implicit flows for SQL statements, one indeed can conduct column-level static taint analysis (STA)[1] , which however generally generates over-tainted results due to two main factors. First, column-level STA does not execute SQL statements to obtain database states or returned results and thus cannot maintain a taint tag for each table cell. Column-level STA tracks implicit flows by statically scanning statement contents, so it is impossible to know which table cells' values in concrete database states are changed or which table cells' values are fetched for subsequent execution. In other words, column-level STA maintains taint tags at a coarse granularity (i.e., the column level) and is not able to track taint propagation for each table cell. Second, column-level STA cannot precisely capture on which specific condition branch sinks are control-dependent. Once the `WHERE` clause of a `SELECT` statement contains multiple condition branches connected by the `OR` operator, some records are fetched due to satisfaction of only one condition branch. Column-level STA can hardly find that

‡Haotian Zhang has resigned from Ant Group Co., Ltd., but this work was done during his employment with Ant Group Co., Ltd.

[1]STA detects whether taint tags are propagated from sources to sinks by static program analysis. Column-level STA refers to STA for SQL statements by maintaining a taint tag for each column.

there is no implicit flow from other condition branches to these records.

In contrast to column-level STA, dynamic taint analysis (DTA)[2] can in principle track fine-grained implicit flows for SQL statements (i.e., each table cell is attached a taint tag), but porting existing DTA approaches [9], [10] faces two major challenges. (1) **Native/unmanaged code challenge**. It is challenging to track implicit flows across runtime environments, which are often implemented in native or unmanaged code. Database interaction APIs implemented in native or unmanaged code [11] run outside of but interact with managed runtimes. General-purpose DTA approaches in managed runtimes handle such flows imprecisely. These approaches discard all taint tags from sources, or just propagate these tags to all sinks during execution of native or unmanaged code. (2) **DBMS complexity challenge**. It is challenging to track implicit flows in the underlying DBMS. Even when the DBMS and its corresponding APIs are implemented in managed code, its engine's implementation tends to be of high complexity [11]. Tracking implicit flows generally requires code instrumentation to obtain the whole program structure [12]. Once any character (even in a keyword such as `SELECT`) in a SQL statement gets tainted, the taint tags are widely propagated to all execution-reachable sinks due to the sophisticated branch statements in the underlying code, causing "taint explosion" [13].

To address these challenges, in this paper, we propose a cell-level DTA framework named TaintSQL, including two novel techniques (each of which can be independently selected and used), namely MutaIF and MockIF, to propagate taint tags during execution of SQL statements. MutaIF is suitable for analysis tasks where users are expected to capture fine-grained implicit flows (1) precisely (i.e., false positives are not tolerated) or (2) for statements containing complicated advanced features such as user-defined functions. MockIF is suitable for analysis tasks where users are expected to capture fine-grained implicit flows (1) comprehensively (i.e., false negatives are not tolerated) or (2) from all or many inputs at once. Neither technique requires any modifications to operating systems or language interpreters. TaintSQL users can choose one of MutaIF and MockIF to meet their needs.

We design MutaIF based on our key insight of *counterfactual mutation*. To address the native/unmanaged code challenge, MutaIF adopts a mutation-based alternative rather than code instrumentation. To address the DBMS complexity challenge, MutaIF regards the underlying DBMS as a "black box" and tracks implicit flows based on experimental observation. It infers taint propagation rules for implicit flows by mutating sources, so it captures causal relationships between values. In particular, in each implicit flow reported by MutaIF, the source actually determines the value of the sink. MutaIF makes use of the `NULL` value and the `NOT` operator to design effective mutations.

[2]DTA uses real-time monitoring of variables during program execution to determine whether taints are propagated from sources to sinks. Compared with STA, DTA effectively reduces the number of false positives.

We design MockIF based on our key insight of *taint twin*. To address the native/unmanaged code challenge, MockIF redirects SQL statements to interact with a mock database and tracks implicit flows in the mock database implementation. To address the DBMS complexity challenge, MockIF replicates the effect of operations over the back-end database by performing the same operations on the mock database (which has the same functionality but is much simpler and less efficient). Taint propagation rules for implicit flows through the mock database are hard-coded in mock operations. Therefore, MockIF can by construction track implicit flows for SQL statements at runtime.

We conduct evaluations on a set of test subjects (including 200 SQL statements from five real-world database applications and an open-source dataset) to assess the effectiveness and efficiency of the MutaIF and MockIF techniques. The evaluation results show that the two techniques effectively track fine-grained implicit flows for SQL statements with reasonable runtime overhead. The F1 scores of MutaIF and MockIF are 96.2% and 97.9%, respectively. We also conduct an industrial study of MutaIF in an international IT company (which serves over **1 billion** global users and **80 million** merchants) because MutaIF is lightweight and the company expects the reported implicit flows to be highly precise. The positive feedback from the software engineers working in this company also demonstrates the practicability of the TaintSQL framework and the MutaIF technique in industrial settings.

In summary, this paper makes the following main contributions:

- **TaintSQL**, the first cell-level DTA framework to track fine-grained implicit flows for SQL statements.
- **MutaIF and MockIF**, two novel techniques to propagate taint tags in implicit flows for SQL statements.
- **Evaluations** for demonstrating the effectiveness and efficiency of MutaIF and MockIF on a set of test subjects. The industrial study also demonstrates the practicability of the TaintSQL framework and the MutaIF technique in industrial settings.

II. Motivation

We are motivated to investigate the problem of tracking implicit flows for SQL statements based on our experience working in an international IT company (represented as *Company A* in the rest of this paper) which serves over 1 billion global users and 80 million merchants. We believe that the motivation is shared by many other companies with a similar demand for tracking such implicit flows to address critical software engineering tasks.

Company A maintains an online fund trading system (which is a database application). In this system, hundreds and thousands of services supporting the company's businesses are provided by interacting with the back-end database via SQL statements. Every day the software engineers struggle to address the following three typical types of software engineering tasks.

- **Security risk assessment.** Safeguarding funds is extremely important to the fund trading system. If there is an error during calculation of fund-related variables in upstream services, and these erroneous variables are used to form condition branches or column names in SQL statements, incorrect results will be fetched or database states will be modified. It may affect tens of downstream services, resulting in serious financial losses. Once such an error occurs, it is necessary to immediately check which sources in upstream services are untrusted and which sinks in downstream services are potentially affected so as to rate the security risk.
- **Software change government.** A large number of software change requests including feature adding, bug fixing, and code refactoring are submitted every day. If some change requests introduce new bugs that lead to unexpected implicit flows for SQL statements, functional correctness of some provided services will not be guaranteed, resulting in unpredictable consequences. It is necessary to check whether change requests are in line with requirements and will not affect the functional correctness of released services.
- **Access control**. In the fund trading system, confidential user information and crucial financial values are continuously fetched from or stored into its back-end database by various groups (e.g., fund data analysts and marketing teams) for different purposes. Some private values should be accessed by only a certain team and other teams may not be allowed to access them, otherwise causing privacy leakage. It is necessary to check which illegal inputs an unauthorized requester may use to form condition branches or column names in SQL statements to access the restricted data.

Tracking implicit flows for SQL statements is needed to address these daily tasks. In *Company A*, the current practice to address the software change government tasks is mainly based on manual code review, whereas the current practice to address the other two types of tasks is mainly based on column-level/table-level STA (i.e., tracking coarse-grained implicit flows for SQL statements by building control-dependency graphs among columns/tables). When tracking implicit flows for SQL statements, the current practice faces the limitation of over-tainted results because once a source input is marked as tainted, all columns/tables that are control-dependent on the source's column/table get tainted. In cases where coarse-grained implicit flows report a high percentage of false positives, software engineers must conduct manual code review to keep track of true implicit flows (e.g., manually determining whether each sink is control-dependent on the tainted sources), which typically takes hours or days. The current solutions are labor-intensive and lack of effectiveness. Therefore, an automated solution to track fine-grained implicit flows for SQL statements is highly desirable.

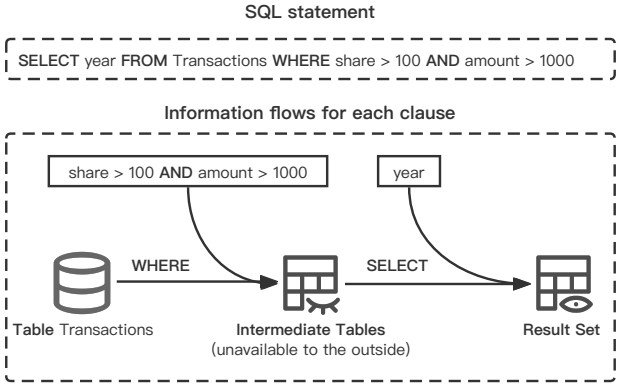

Fig. 1. Information flows for each clause in a `SELECT` statement.

III. Problem Formalization

Although in practice there is a growing demand for tracking fine-grained implicit flows for SQL statements, such implicit flows are ill-defined for the following two main reasons. First, unlike procedural programming language (e.g., C), SQL is a high-level non-procedural programming language. Each SQL statement conveys only what data to manipulate, not how to manipulate the data, so the fine-grained implicit flows are hidden behind each SQL statement during its execution. Second, the underlying code of DBMS is of high complexity, so using standard semantics for propagating taint tags results in severe over-tainting. It is impractical to define implicit flows for SQL statements in such a manner.

To propose an empirically reasonable definition of fine-grained implicit flows for SQL statements, our insight is based on the semantics of each clause. During execution of a SQL statement, its clauses are executed in a determined order [14], and they feed intermediate tables (which are unavailable to the outside) between each other. There are information flows for each clause from sources in its content and input intermediate table to sinks in its output intermediate table. For example, Figure 1 shows information flows for each clause in a `SELECT` statement. On the basis of our industrial experience, we first introduce the definition of fine-grained implicit flows for each clause in a SQL statement.

Definition 1 (fine-grained implicit flows for a clause): A fine-grained implicit flow for a clause is a fine-grained observable information flow from the source in this clause's inputs to the sink in this clause's outputs where the sink is control-dependent on the source. The details are as follows[3]:

- During execution of a `WHERE` or `HAVING` clause, the rows from the previous intermediate table where the condition branches hold are filtered into the next intermediate table. If the condition branches are formed with $\texttt{branch}_1$

[3]Although some sinks may also be control-dependent on the sources like table names in a `FROM` clause, these weak dependencies are ignored to avoid over-tainting.

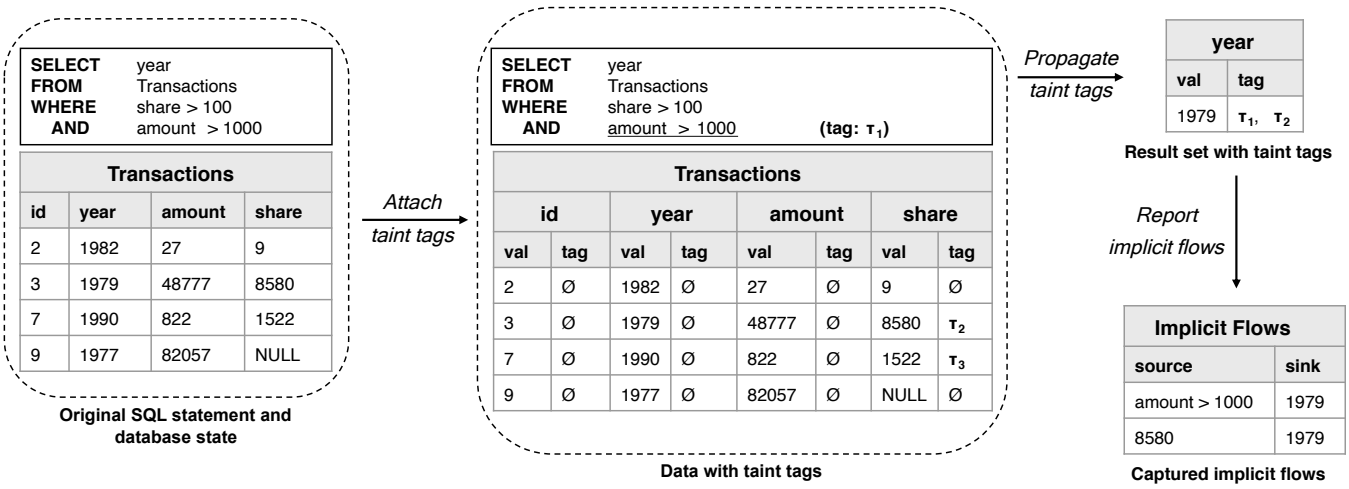

Fig. 2. TaintSQL workflow.

$\wedge \cdots \wedge$ $\texttt{branch}_n$, for each row that satisfies the condition branches, there are implicit flows from each condition branch to the values in the row, and there are also implicit flows from the values whose columns appear in the condition branches to the other values in the same row. If the condition branches are formed with $\texttt{branch}_1 \vee \cdots \vee \texttt{branch}_n$, for each row that satisfies $\texttt{branch}_i$ $(1 \leq i \leq n)$ but does not satisfies any other branch before $\texttt{branch}_i$, there are implicit flows from $\texttt{branch}_i$ to the values in the row, and there are also implicit flows from the values whose columns appear in $\texttt{branch}_i$ to the other values in the same row.

- During execution of a `SELECT` clause, the specific columns from the previous intermediate table are selected into the next intermediate table. There are implicit flows from each column name to the selected values in the corresponding column. Note that if the `SELECT` clause contains any aggregation function, there are explicit flows or implicit flows from the table cells in the previous intermediate table processed by this function to the aggregation results in the next intermediate table according to the semantics of this function.
- During execution of a `SET` or `INTO` clause, the specific columns are selected to be updated and a new database state is generated. There are implicit flows from each column name to the updated or inserted values in the corresponding column.

By nature an information flow for a SQL statement is determined by the connection of information flows for each clause[4] (i.e., the sinks of the previously executed clause are the sources of the next clause to be executed). We now introduce the definition of fine-grained implicit flows for a SQL statement.

Definition 2 (fine-grained implicit flows for a SQL statement): A fine-grained information flow for a SQL statement is an implicit flow *if and only if* there exists at least one fine-grained implicit flow in the connected information flows for each clause.

[4]The definition of fine-grained explicit flows for each clause in a SQL statement can be found in previous work [6], [15].

IV. TaintSQL Framework

We propose TaintSQL, a cell-level taint analysis framework to track fine-grained implicit flows for SQL statements. TaintSQL does not require any modifications to operating systems or language interpreters. The workflow of TaintSQL is shown in Figure 2. During the execution of a database application, TaintSQL takes each SQL statement and the database state as inputs (e.g., the `SELECT` statement and the `Transactions` table in Figure 2). To track implicit flows for SQL statements, TaintSQL receives the taint tags (e.g., τ_1, τ_2, and τ_3 in Figure 2) of sources (which are located in the SQL statement and/or the database state). These associated taint tags may be attached by TaintSQL users manually, or derived from the preceding taint propagation results. Then TaintSQL dynamically propagates taint tags to the corresponding sinks (which are located in the fetched result set or the new database state). According to the taint tag of each sink, we can obtain a complete picture of implicit flows for SQL statements (e.g., there are implicit flows from `amount > 1000` and `8580` to `1979` in Figure 2).

A. Attaching Taint Tags

After receiving the inputs of a SQL statement and a database state, TaintSQL first parses the SQL statement to analyze the involved database tables and columns. In practice, tracking implicit flows does not involve the whole database state at once, so a large amount of irrelevant data can be reduced in the test environment.

Then TaintSQL parses the schema of the SQL statement's back-end database and creates a new unit for each table cell to store its associated taint tag in the database state. Generally, there are two manners to maintain the mapping between each

value and its taint tags in the database state. First, each value and its associated taint tag are stored in the same table cell. The storage requires that the DBMS should support composite data types (e.g., PostgreSQL [16]). Second, each value and its associated taint tag are stored in different columns but in the same row. The storage requires that additional columns should be added to store taint tags.

The taint tag of each piece of data is initially empty, so TaintSQL requires the sources to be marked as tainted before taint tracking. These taint tags can be derived from the preceding taint propagation results or manual marking. Each taint tag not only reflects whether the associated data gets tainted, but also makes it clear where the implicit flow comes from. Compared with recording whether each table cell gets tainted by only a single bit (i.e., `false` indicates untainted and `true` indicates tainted), TaintSQL allows using multiple distinct taint tags to mark different sources at once, and thus can distinguish the taint propagation paths from different sources.

B. Propagating Taint Tags

After monitoring the mapping between each source and its associated taint tag, TaintSQL begins to use one of the two different techniques (namely MutaIF and MockIF) chosen by its users to propagate taint tags. MutaIF propagates taint tags via each source's mutants. It captures the causal relationships between sources and sinks existing in implicit flows. MockIF propagates taint tags in mock databases. It reduces database manipulations to array operations in memory. The details are elaborated in Section IV-C.

By observing the distribution of sinks' taint tags, TaintSQL users can know which sinks get tainted and which sources these tainted sinks are control-dependent on. For `SELECT` statements, TaintSQL retrieves pieces of data together with the associated taint tags. For other statements such as `UPDATE` and `INSERT`, TaintSQL updates database states with the associated taint tags. Based on the taint propagation results, TaintSQL users can intuitively detect the fine-grained implicit flows for SQL statements. These captured implicit flows are able to help software engineers address critical issues in time. TaintSQL users can also take the taint propagation results as inputs for subsequent taint analysis to track comprehensive information flows for application code (e.g., Java or C#).

C. MutaIF and MockIF

TaintSQL includes two techniques to propagate taints in implicit flows for SQL statements, namely MutaIF and MockIF. They use different strategies to address the challenges of taint propagation. MutaIF is suitable for analysis tasks where users are expected to capture fine-grained implicit flows (1) precisely (i.e., false positives are not tolerated) or (2) for statements containing complicated advanced features such as user-defined functions. MockIF is suitable for analysis tasks where users are expected to capture fine-grained implicit flows (1) comprehensively (i.e., false negatives are not tolerated) or (2) from all or many inputs at once. Neither technique requires modifying the underlying environments.

Algorithm 1 Propagating taint tags for a `SELECT` statement

Input: Q: a `SELECT` statement;
Input: D: the database state;
Input: T_N: each column name in Q and its taint tag;
Input: T_B: each condition branch in Q and its taint tag;
Input: T_D: each table cell's value in D and its taint tag;
Output: R: the fetched result set;
Output: T_R: each table cell's value in R and its taint tag.

```
1:  R ← Execute Q on D
2:  for each tainted column name n in Q do
3:      T_n ← Get n's taint tag from T_N
4:      R_d ← Get all values in the column n from R
5:      T_R ← Propagate T_n to the values in R_d
6:  end for
7:  for each tainted condition branch b in Q do
8:      T_b ← Get b's taint tag from T_B
9:      Q_m ← Mutate b to its negation
10:     R_m ← Execute Q_m on D
11:     R_d ← Compare R with R_m
12:     T_R ← Propagate T_b to the values in R_d
13: end for
14: for each tainted table cell's value v in D do
15:     T_v ← Get v's taint tag from T_D
16:     c ← Get the column name of v
17:     if c is in Q's condition branches then
18:         if v is not NULL then
19:             D_m ← Mutate v to NULL
20:         else
21:             D_m ← Mutate v to a random value
22:         end if
23:         R_m ← Execute Q on D_m
24:         R_d ← Compare R with R_m
25:         T_R ← Propagate T_v to the values in R_d
26:     end if
27: end for
28: return ⟨R, T_R⟩
```

1) MutaIF: MutaIF infers taint propagation paths from each source by mutating its value to observe which sinks get changed. We name this insight as *counterfactual mutation* because it is inspired by counterfactual inference [17], the theoretical basis of causal reasoning. Algorithm 1 gives details about how to propagate taint tags for a `SELECT` statement via MutaIF. Note that T_D, T_B, and T_N can be derived from the preceding taint propagation results or manual marking.

To infer taint propagation path for a `SELECT` statement, MutaIF first needs to execute the original statement to get the result set R (Line 1). To propagate taint tags from the column names, MutaIF directly assigns each column name's taint tag to all the values belonging to this column (Lines 3-5). To propagate taint tags from the condition branches, MutaIF negates each tainted condition branch by adding the `NOT` operator to this condition branch (Line 9). Then it executes

the mutated statement to retrieve the result set R_m (Line 10). If some rows in R do not exist in R_m, MutaIF propagates the condition branch's taint tag to the values in these rows (Lines 11-12).

To propagate taint tags from the table cells in the database state, MutaIF mutates each source's value to `NULL`[5] where the original value is not `NULL` (Line 19). But if the original value is `NULL`, MutaIF mutates it to a random value with the same data type (Line 21). MutaIF leverages the `NULL` value to mutate each tainted table cell because the processing of `NULL` is different from other values in common DBMSs. The comparison operators such as ">=" cannot be used to match `NULL`. Mutating the source to `NULL` can ensure that it will not satisfy the condition branches in common cases so that the implicit flows can be identified with only one mutant. Moreover, `NULL` is compatible with arbitrary data types. Then MutaIF re-executes the original statement to retrieve the result set R_m (Line 23). By comparing R_m with R, MutaIF determines which rows in R get changed (Line 24). Thus, MutaIF propagates the table cell's taint tag to the values in these rows (Line 25).

Now we illustrate how MutaIF propagates taint tags for the example in Figure 2. First, MutaIF executes the original statement and gets the year `1979`. Then it mutates the tainted condition branch to create a statement mutant. After executing this mutant, the year `1990` instead of `1979` is fetched, so the year `1979` is marked as τ_1. Next, MutaIF mutates the value `8580` to `NULL`, and re-executes the original statement. The empty result set indicates that the year `1979` gets tainted with τ_2. However, after mutating the value `1522` to `NULL`, the year `1979` is still fetched, so τ_3 should not be propagated to it. Finally, the taint propagation result is $\langle$`1979`, $\{\tau_1, \tau_2\}\rangle$.

MutaIF leverages similar strategies to propagate taint tags for other types of statements via counterfactual mutation. For `INSERT` and `UPDATE` statements, MutaIF propagates taint tags from column names in `INTO` and `SET` clauses to the corresponding table cells' values. For `UPDATE` statements, MutaIF constructs the corresponding `SELECT` statements to retrieve data to be updated. MutaIF also mutates each source in `WHERE` clauses and original database states to determine which table cells' values in the updated database states are affected by each source.

2) MockIF: MockIF propagates taint tags by using application code to simulate database states and operations, i.e., reducing the core functionalities in real-world DBMSs to basic operations on object arrays. It intercepts each SQL statement forwarded by database interaction APIs and executes mock operations on object arrays with equivalent semantics. The taint tags of sources are propagated along with data manipulation in memory. We name this insight as *taint twin* because it performs the same operations on a mock database as on a real-world database during execution of a SQL statement. For example, when executing an `INSERT` statement, MockIF parses the statement to extract the values to be inserted. Then it inserts these values along with their associated taint tags into the corresponding columns of a mock database.

To propagate taint tags during execution of a SQL statement, MockIF first mocks the original database state. It uses a two-dimensional array to represent a database table. Each column of the array corresponds to a column in the real-world database table (i.e., a field) or a column used to store taint tags of table cells in a specific field. Each row of the array is equivalent to a row in the real-world database table (i.e., a record).

Besides mocking database states, MockIF also mocks the core operations for clause execution. For each SQL statement, MockIF parses its clauses and maintains the mapping between each source and its taint tag. Then it leverages semantically equivalent but simplified operations to execute each clause on the mock database. Although such operations do not support advanced optimizers, the information flows for each clause keep consistent. More importantly, the mock operations are allowed to integrate hand-crafted taint propagation rules for each clause. Thus, MockIF is able to execute each clause along with taint propagation at runtime. For the operation used to filter rows in the object array, MockIF analyzes each condition branch recursively and gets intermediate tables for each condition branch. For each row in an intermediate table that satisfies a condition branch, MockIF propagates the taint tags of that condition branch and the table cells' values whose columns appear in that condition branch to all values in the row. If two condition branches are connected by `AND`, MockIF intersects the data and unions the taint tags from two intermediate tables. If two condition branches are connected by `OR`, MockIF unions the data from two intermediate tables. If a row satisfies both condition branches and is filtered into both intermediate tables, MockIF selects the taint tags of values in that row from the first intermediate table. For other operations used to manipulate specific columns in the object array, MockIF propagates the taint tags of each column name to the table cells' values in the corresponding column.

Now we illustrate how MutaIF propagates taint tags for the example in Figure 2. First, MockIF stores all table cells' values of the `Transactions` table in an array. For each condition branch, MockIF traverses each row in the mock database to determine whether it satisfies the condition branch. Then it generates an intermediate table for each condition branch and assigns the condition branch's taint tag to each value in the intermediate table. For each row, the filtered table cells are control-dependent on the value in the same row whose column appears in the condition branch (i.e., `amount` or `share`), so the value's tag should also be propagated to the other values in the same row. Therefore, the year `1979` is marked as $\{\tau_2\}$ in the first intermediate table, and is marked as $\{\tau_1\}$ in the second intermediate table. Since the first condition branch "`share > 100`" and the second condition branch "`amount > 1000`" are connected by `AND`, MockIF intersects the data and unions the taint tags from two intermediate tables. Finally, the taint propagation result is $\langle$`1979`, $\{\tau_1, \tau_2\}\rangle$.

[5]In the test environment, TaintSQL users can alter the database schema and remove some inessential non-null constraints in advance.

V. Evaluation

In our evaluations, we address the following three research questions (RQs):

- **RQ1:** How effectively does TaintSQL (MutaIF and MockIF, respectively) track implicit flows for SQL statements compared with the baseline?
- **RQ2:** How much is the runtime overhead of TaintSQL (MutaIF and MockIF, respectively)?
- **RQ3:** How practicably does TaintSQL perform in industrial settings?

A. Evaluation Setup

1) Subjects: To the best of our knowledge, there is no existing benchmark suite tailored for tracking fine-grained implicit flows for SQL statements. We therefore collect SQL statements from five real-world database applications and an open-source text-to-SQL dataset to evaluate the effectiveness and efficiency of TaintSQL. (1) iTRUST [18] is a class project created at North Carolina State University for teaching software engineering. It consists of functionalities that cater to patients and the medical staff. (2) RiskIt [19] is an insurance quote application that makes estimation based on users' personal information (e.g., zipcode and income). (3) UnixUsage [20] is an application to obtain statistics about how users interact with the Unix systems using different commands. (4) Odyssey [21] is an open source library that manages controls for WPF and ASP.NET. (5) JForum [22] is a complete, powerful, robust, and multi-threaded discussion board system. Its features include forums and messages, topic watching, email notification, advanced permission schema, and more. (6) Spider [23] is a semantic parsing and text-to-SQL dataset consisting of complex `SELECT` statements covering different domains. We choose these database applications and datasets because they have been widely used as evaluation subjects in previous work [24], [11], [6].

We finally construct a test subject suite including 200 randomly selected SQL statements in total because analyzing taint propagation results for all queries requires unacceptable manual efforts. These statements are grouped into five categories (i.e., *Single-Branch-Select*, *Single-Branch-Update*, *Multi-Branch-Select*, *Multi-Branch-Update*, and *Insert*), each of which is designed to evaluate a representative feature of SQL (e.g., different types of SQL statements and different numbers of branches)[6].

We assign an initial database state to each SQL statement in the test subject suite with the following criteria. If a SQL statement is derived from Odyssey (which does not provide the database state), we use a data generation tool named Fake [26] to randomly generate an initial database state for this SQL statement. Otherwise, we select the database state provided by each database application or dataset as the initial database state for each SQL statement.

2) Baseline: Note that there is no applicable baseline to which we can compare our work because no existing work focuses on tracking fine-grained implicit flows for SQL statements and TaintSQL is the first cell-level DTA framework for it. Therefore, We design a column-level STA approach as the baseline. Once a table cell's value in a column gets tainted, the column-level STA regards that all values in that column are tainted and share the same taint tag.

The code-instrumentation-based DTA approach is not applicable as a baseline due to the challenges of native/unmanaged code and DBMS complexity. Even if we use a state-of-the-art dynamic taint tracking tool for JVM named PHOSPHOR [9], [10] to track implicit flows through a simplified open-source DBMS [27] that does not implement optimizers or any other advanced feature, the taint propagation results still show that this approach is not applicable because as long as any character in a `SELECT` statement is tainted (and even any obtained token after the parsing stage is tainted), all fetched values get tainted.

3) Methodology: Because there is no existing labeled ground truth, we need to manually check all taint propagation results to evaluate our approaches. However, it is impractical to manually check fine-grained implicit flows from all sources to all sinks to identify the complete set of true implicit flows. Therefore, we limit the size of each database table to a maximum of 10 records. In addition, we randomly pick up to five table cells' values, up to five condition branches, and up to five column names as sources. We assign a distinct taint tag to each source.

To make our evaluation results convincing, we use the following cross-validation policy to check each reported implicit flow. We first collect all implicit flows reported by our proposed techniques and the baseline. We independently classify all reported implicit flows to distinguish true from false ones based on the definition of implicit flows for SQL statements. We then discuss the disagreements for inconsistent classifications, and finally all the implicit flows are classified consistently. Based on all classified implicit flows, we count the number of true positives (TP), false positives (FP), and false negatives (FN) for each approach, respectively.

4) Metrics: To measure the effectiveness and efficiency of TaintSQL, we calculate the following four types of metrics:

- The precision rate (denoted as P) is defined as the proportion of true implicit flows in all reported implicit flows: $P = \frac{TP}{TP+FP}$.
- The recall rate (denoted as R) is defined as the proportion of reported implicit flows in all true implicit flows: $R = \frac{TP}{TP+FN}$.
- The F1 score (denoted as F_1) is defined as the harmonic mean of the precision rate and the recall rate: $F_1 = \frac{2PR}{P+R}$.
- The increased runtime overhead (denoted as ΔT) is defined as the average cost per statement increased by taint propagation.

5) Implementations and Environments: We implement the MutaIF prototype in Java. It leverages JSqlParser [28] to

[6]Nested statements are not included because previous work [25] has demonstrated that most of them can be unnested into equivalent canonical statements via transformation. In addition, `DELETE` statements are not included because they delete the values along with their taint tags. Although the deletion behavior may affect the subsequent data processing results, such hidden implicit flows [8] are out of scope for TaintSQL.

TABLE I
TAINT PROPAGATION RESULTS FOR COLUMN-LEVEL STA, MUTAIF, AND MOCKIF

Test Subject Category		Column-Level STA			MutaIF				MockIF			
Name	**Source**	P	R	F_1	P	R	F_1	$\Delta T(s)$	P	R	F_1	$\Delta T(s)$
Single-Branch-Select	Table Cell	42.7%	100.0%	59.8%	100.0%	84.3%	91.5%	0.18	91.9%	100.0%	95.8%	0.09
	Condition Branch	93.8%	100.0%	96.8%	100.0%	90.2%	94.8%	0.09	93.8%	100.0%	96.8%	0.09
	Column Name	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	0.03	100.0%	100.0%	100.0%	0.09
Single-Branch-Update	Table Cell	24.1%	100.0%	38.9%	100.0%	100.0%	100.0%	0.28	100.0%	100.0%	100.0%	0.08
	Condition Branch	29.0%	100.0%	45.0%	100.0%	100.0%	100.0%	0.19	100.0%	100.0%	100.0%	0.08
	Column Name	28.7%	100.0%	44.5%	100.0%	100.0%	100.0%	0.05	100.0%	100.0%	100.0%	0.08
Multi-Branch-Select	Table Cell	38.3%	100.0%	55.4%	100.0%	75.2%	85.8%	0.23	79.5%	100.0%	88.6%	0.16
	Condition Branch	54.0%	100.0%	70.1%	100.0%	79.1%	88.3%	0.17	85.5%	100.0%	92.2%	0.16
	Column Name	100.0%	100.0%	100.0%	100.0%	100.0%	100.0%	0.04	100.0%	100.0%	100.0%	0.16
Multi-Branch-Update	Table Cell	16.3%	100.0%	28.0%	100.0%	88.2%	93.7%	0.29	100.0%	100.0%	100.0%	0.15
	Condition Branch	21.8%	100.0%	35.8%	100.0%	92.9%	96.3%	0.24	100.0%	100.0%	100.0%	0.15
	Column Name	28.8%	100.0%	44.8%	100.0%	100.0%	100.0%	0.06	100.0%	100.0%	100.0%	0.15
Insert	Column Name	13.6%	100.0%	24.0%	100.0%	100.0%	100.0%	0.04	100.0%	100.0%	100.0%	0.13
Average		45.5%	100.0%	57.2%	100.0%	93.1%	96.2%	0.15	96.2%	100.0%	97.9%	0.12

parse SQL statements. We implement the MockIF prototype in C#. It leverages ANTLR [29] to parse SQL statements. It mocks database states and operations in memory based on an automated test generation tool for database applications named MODA [24], which guarantees the semantic equivalence for common SQL features. The prototype code, industrial study cases, and supplementary materials such as supported SQL features are publicly available [30].

To answer RQ1 and RQ2, the evaluations are conducted on a Windows 11 laptop with Intel i7-8565U CPU and 16 GB of memory. To answer RQ3, the evaluations are conducted on a Linux server with 2.5 GHz Intel Xeon Platinum 8163 CPU and 4 GB of memory.

B. RQ1: Effectiveness

We design and conduct the following evaluations on our test subject suite to assess the effectiveness of TaintSQL.

1) Taint Propagation Results: We report the taint propagation results for our proposed techniques in TaintSQL and the baseline in Table I. Note that to mitigate experimental biases, we randomly pick 15% untainted sinks to check whether there are unreported but true implicit flows from tainted sources to these sinks. After such samplings are done twice and no more false negatives are found, we believe that the results are convincing.

The evaluation results in Table I show that the F1 scores of both MutaIF and MockIF are higher than column-level STA. Column-level STA achieves the lowest average precision rate (45.5%) and F1 score (57.2%) because it cannot distinguish taint tags of different values in the same column. MutaIF achieves the highest precision rate (100.0%) because it precisely captures causal dependencies between data. MockIF achieves the highest average recall rate (100.0%) and F1 score (97.9%) because it propagates taint tags along with mock operations, reflecting the equivalent semantics of clause execution. Therefore, both techniques in TaintSQL outperform the baseline.

2) False Negatives Reported by MutaIF: In the evaluations, we find that there are implicit flows that MutaIF does not capture. We analyze the in-depth root causes as follows.

- MutaIF mutates a tainted condition branch but the same results are returned because the records satisfy multiple condition branches connected by `OR` at the same time. Mutating the first condition branch will not affect the records satisfying other condition branches. The fact is that there are implicit flows from this condition branch to the values in the results, but MutaIF mistakenly infers that there is no implicit flow.
- MutaIF mutates a tainted condition branch but the same results are returned because exactly the same values from different table cells are fetched. Although the two results are numerically identical, they come from different table cells with different semantics. Besides the semantics, their original taint tags may also be different. The fact is that there are implicit flows from this condition branch to the values in the results, but MutaIF mistakenly infers that there is no implicit flow.

3) False Positives Reported by MockIF: In the evaluations, we find that there are false implicit flows reported by MockIF. We analyze the in-depth root causes as follows.

- MockIF guarantees the conservativeness of taint propagation for different aggregation functions with diverse semantics. In strict, some aggregation results are not dependent on the original table cell's values. For example, if a table cell's value is 0, it will not affect the aggregation result by the `SUM` function. But MockIF conservatively unions all taint tags and assigns these tags to the final results, and thus reports false implicit flows.
- The current prototype of MockIF has not yet supported some SQL features such as the `LIKE` operator and scalar functions. When analyzing SQL statements containing such features, MockIF leverages the same approach as column-level STA to propagate taint tags. Therefore, it may report some false implicit flows for these statements.

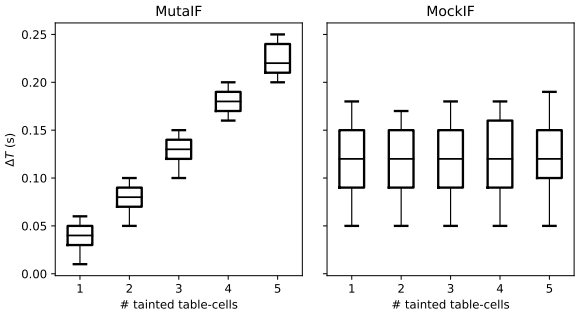

Fig. 3. Impact of different numbers of tainted table cells.

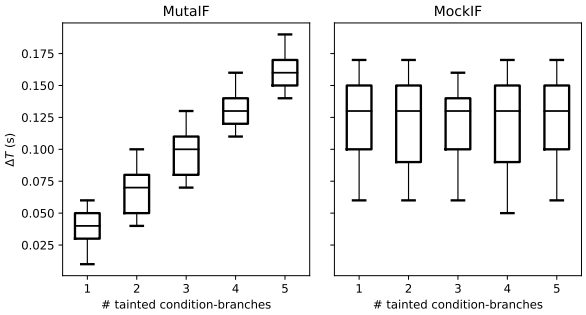

Fig. 4. Impact of different numbers of tainted condition branches.

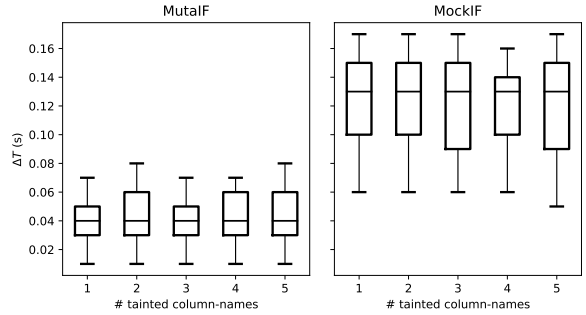

Fig. 5. Impact of different numbers of tainted column names.

C. RQ2: Efficiency

The evaluation results shown in Table I illustrate that MutaIF costs an average of 0.15 seconds for taint propagation, whereas MockIF costs an average of 0.12 seconds for taint propagation. In practice, only relevant columns/tables and records (which tend to be in the minority) are kept in the test environment, so the increased runtime overhead is reasonable.

In addition, we conduct empirical evaluations on MutaIF and MockIF to assess the impact of different numbers of sources on runtime overhead. We randomly choose 10 SQL statements in the test subject suite, and record the runtime overhead of propagating taint tags under different factors by MutaIF and MockIF, respectively.

1) Impact of Different Numbers of Tainted Table Cells: The time costs for different numbers of tainted table cells are shown in Figure 3. The results show that as the number of tainted table cells increases, the taint propagation time of MutaIF becomes longer, but the efficiency of MockIF is almost unaffected. The reason is that MutaIF mutates once for each tainted table cell, and then re-executes the SQL statement to track implicit flows. Assuming that there are n tainted table cells, the same SQL statement needs to be executed at least $n + 1$ times. MockIF executes the SQL statement only once, and all taint tags are propagated to sinks during execution of mock operations. Therefore, the number of tainted table cells has little impact on the cost of MockIF.

2) Impact of Different Numbers of Tainted Condition Branches: The time costs for different numbers of tainted condition branches are shown in Figure 4. The results show that the runtime overhead of MutaIF rises slowly with increase of the number of tainted condition branches. The reason is that although more statement mutants are executed during taint propagation by MutaIF, there is no need for `SELECT` statements to backup and restore database states. The runtime overhead of MockIF is basically not affected also because there is no need for MockIF to re-execute the SQL statements.

3) Impact of Different Numbers of Tainted Column Names: The time costs for different numbers of tainted column names are shown in Figure 5. The results show that different numbers of tainted column names hardly affect the runtime overhead of MutaIF and MockIF. The reason is that there is no need for both MutaIF and MockIF to re-execute the SQL statements when propagating taint tags from column names.

D. RQ3: Practicability

To assess the practicability of TaintSQL, we present an industrial study conducted in the test environment of an online fund trading system maintained by *Company A*. It adopts a micro-service architecture and uses an open-source distributed DBMS named OceanBase [31] as its underlying DBMS. All evaluation subjects covering multiple fund businesses are developed on an open-source financial-level distributed architecture named SOFAStack [32] and use a self-developed middleware to interact with back-end databases.

We choose the MutaIF technique to propagate taint tags in industrial settings because MutaIF is lightweight and the software engineers working in *Company A* expect the reported implicit flows to be highly precise. We reproduce historically captured implicit flows for SQL statements for addressing the software engineering tasks (introduced in Section II) in eight cases. After manual checking, the study results show that all fine-grained implicit flows in these cases are captured and all implicit flows reported by MutaIF are true positives (some of which are the key to triggering severe risks). Compared with the column-level/table-level STA approaches in current practice, using MutaIF can eliminate more than half of the implicit flows which are proved to be false positives.

Taking a `SELECT` statement shown in Figure 6 for example (with millions of daily users, some values are anonymized

```
SELECT SUM(apply_amount)
FROM fund_trade_order_01
WHERE user_id = <user_id>
  AND (scene_type = <scene_type_1>
    OR scene_type = <scene_type_2>)
  AND (order_type = <order_type_1>
    OR order_type = <order_type_2>)
  AND order_status = <order_status>
```

Fig. 6. Calculating the total amount of short-term investment in transit.

according to the company policy), it returns a wrong aggregation result which affects the downstream services that receive this result as an input. The current practice only conservatively infers that there are implicit flows from all condition branches to this result, making software engineers take non-trivial efforts to check. But MutaIF successfully detects that this result is control-dependent on the "`scene_type = <scene_type_2>`" condition branch instead of the "`scene_type = <scene_type_1>`" condition branch. According to the combination of the fine-grained implicit flows for SQL statements reported by MutaIF with the information flows for application code captured by general-purpose taint analysis approaches, the software engineers are able to conduct traceability and impact analysis of erroneous data and rate the security risk.

Based on the industrial study results, our TaintSQL framework and the MutaIF technique receive positive feedback from the software engineers working in *Company A*. They comment that MutaIF "*generates convincing results with acceptable overhead*" and TaintSQL "*is able to address pain points in their day-to-day work*".

E. Threats to Validity

Like any research of taint analysis, our evaluations are subject to threats to validity.

1) External Validity: We are aware that the evaluation results are preliminary and more extensive studies are needed in future work. First, a threat includes the degree to which the database states and SQL statements in our evaluations are representative of true practice. Our results may not generalize to arbitrary subjects. Second, our implementations perform taint analysis for only relational database statements. Advanced transaction processing and NoSQL statements are not yet supported. Third, our prototypes do not integrate taint analysis approaches for application code. We assume that TaintSQL users have the ability to use general-purpose taint analysis tools to track complete information flows in real-world services. Nevertheless, we believe that the benefits and costs of TaintSQL are promising based on our evaluation results.

2) Internal Validity: On internal validity, our evaluations may subject to researcher biases. Bugs in our prototypes, manual efforts for some not-yet-automated functionalities (e.g., replacing database interaction APIs), and the underlying integrated tools (e.g., JSqlParser [28] and MODA [24]) might cause such effects. To reduce these threats, we manually inspect sampled runtime traces of TaintSQL. The implicit flows for SQL statements tracked by TaintSQL in *Company A* have been confirmed by software engineers with years of working experience.

VI. Related Work

To the best of our knowledge, TaintSQL is the first cell-level DTA framework to track fine-grained implicit flows for SQL statements. There are two main categories of related work for TaintSQL.

A. Taint Analysis for Application Code

A large number of academic and industrial studies focus on tracking explicit flows or implicit flows for application code. PHOSPHOR [9], [10] and ClearTrack [33] support DTA of Java applications. TAINTART [34], TaintDroid [35], and MUTAFLOW [36] support DTA of Android applications. FLOWDROID [37], DroidSafe [38], and DroidInfer [39] support STA of Android applications. TASEL [40], NEUTAINT [41], DTA++ [42], TAINTINDUCE [43], and DYTAN [12] are feasible for propagating taint tags in C/C++ binaries. In recent years, a number of approaches have made an attempt to perform DTA in GraalVM [44], [45], [46] (a polyglot virtual machine in which multi-language applications can be run [47]) or data-intensive scalable computing (DISC) systems [48], [15]. In an industrial setting, ANTaint [49] developed by Alibaba supports STA for service-oriented architecture (SOA) applications. These preceding approaches treat SQL statements, which interact with back-end databases, as trivial embedded strings in the application code. There is no in-depth semantic information of SQL statements, so taint tags carried in these statements cannot be correctly propagated to the corresponding sinks. Different from these approaches, TaintSQL aims to track implicit flows for SQL statements instead of application code. However, due to the complexity of implicit flows, code instrumentation of underlying DBMSs or manually specifying complicated policies are impractical. Tracking fine-grained implicit flows for SQL statements faces more challenges than tracking explicit flows.

B. Taint Analysis for SQL Statements

DBTaint [6] supports cell-level DTA to track explicit flows for SQL statements. It modifies database interfaces and rewrites SQL statements so that both data and the associated taint tags can be stored in or retrieved from back-end databases at the same time. LABELFLOW [50] and SilverLine [51] support row-level DTA (maintaining a taint tag for each row) for PHP-based web applications. RESIN [52] allows developers to write application-specific code for the assertions that must hold for each table cell. It keeps track of assertions as explicit flows for SQL statements. Yang et al. [53] present faceted databases for supporting policy-agnostic SQL statements, and prove that the interoperation with faceted databases yields strong guarantees. Schütte and Brost [3] propose a domain-specific policy language in the model of controlling data usage

and track explicit flows to monitor the processing of individual data. In an industrial setting, MaxCompute [54] (also called ODPS), a big-data computing engine developed by Alibaba, supports column-level/table-level STA for SQL statements. These approaches lack TaintSQL's ability to track fine-grained implicit flows for SQL statements, and there is no need for TaintSQL users to manually specify complicated policies.

VII. Conclusion

In this paper, we have presented a cell-level DTA framework named TaintSQL, including two novel taint propagation techniques, namely MutaIF and MockIF, to track fine-grained implicit flows for SQL statements. We have implemented both techniques and evaluated them on a set of test subjects. The evaluation results show that the techniques effectively track fine-grained implicit flows for SQL statements with reasonable runtime overhead. The F1 scores of MutaIF and MockIF are 96.2% and 97.9%, respectively. We have also conducted an industrial study of MutaIF on an online fund trading system in an international IT company (which serves over **1 billion** global users and **80 million** merchants). The positive feedback from the software engineers also shows the practicability of the TaintSQL framework and the MutaIF technique in industrial settings.

Acknowledgment

We thank the anonymous reviewers for their insightful comments. This work was partially supported by National Natural Science Foundation of China (Grant No. 62161146003), Ant Group through Ant Research Program, and XPLORER PRIZE.

References

[1] P. Wang, W. J. Chao, K. Chao, and C. Lo, "Using taint analysis for threat risk of cloud applications," in *11th IEEE International Conference on e-Business Engineering, ICEBE 2014, Guangzhou, China, November 5-7, 2014*, Y. Li, X. Fei, K. Chao, and J. Chung, Eds. IEEE Computer Society, 2014, pp. 185–190. [Online]. Available: https://doi.org/10.1109/ICEBE.2014.40

[2] B. Li, X. Sun, H. Leung, and S. Zhang, "A survey of code-based change impact analysis techniques," *Softw. Test. Verification Reliab.*, vol. 23, no. 8, pp. 613–646, 2013. [Online]. Available: https://doi.org/10.1002/stvr.1475

[3] J. Schütte and G. S. Brost, "A data usage control system using dynamic taint tracking," in *30th IEEE International Conference on Advanced Information Networking and Applications, AINA 2016, Crans-Montana, Switzerland, 23-25 March, 2016*, L. Barolli, M. Takizawa, T. Enokido, A. J. Jara, and Y. Bocchi, Eds. IEEE Computer Society, 2016, pp. 909–916. [Online]. Available: https://doi.org/10.1109/AINA.2016.127

[4] F. Berner and J. Sametinger, "Dynamic taint-tracking: Directions for future research," in *Proceedings of the 16th International Joint Conference on e-Business and Telecommunications, ICETE 2019 - Volume 2: SECRYPT, Prague, Czech Republic, July 26-28, 2019*, M. S. Obaidat and P. Samarati, Eds. SciTePress, 2019, pp. 294–305. [Online]. Available: https://doi.org/10.5220/0008118502940305

[5] Y. Liu, L. Liao, and T. Song, "Static tainting extraction approach based on information flow graph for personally identifiable information," *Sci. China Inf. Sci.*, vol. 63, no. 3, 2020. [Online]. Available: https://doi.org/10.1007/s11432-018-9839-6

[6] B. Davis and H. Chen, "Dbtaint: Cross-application information flow tracking via databases," in *USENIX Conference on Web Application Development, WebApps'10, Boston, Massachusetts, USA, June 23-24, 2010*, J. K. Ousterhout, Ed. USENIX Association, 2010. [Online]. Available: https://www.usenix.org/conference/webapps-10/dbtaint-cross-application-information-flow-tracking-databases

[7] K. Hough and J. Bell, "A Practical Approach for Dynamic Taint Tracking with Control-flow Relationships," *ACM Transactions on Software Engineering and Methodology*, vol. 31, no. 2, pp. 1–43, 2022.

[8] C.-A. Staicu, D. Schoepe, M. Balliu, M. Pradel, and A. Sabelfeld, "An empirical study of information flows in real-world javascript," in *Proceedings of the 14th ACM SIGSAC Workshop on Programming Languages and Analysis for Security*, 2019, pp. 45–59.

[9] J. Bell and G. E. Kaiser, "Phosphor: illuminating dynamic data flow in commodity jvms," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, A. P. Black and T. D. Millstein, Eds. ACM, 2014, pp. 83–101. [Online]. Available: https://doi.org/10.1145/2660193.2660212

[10] ——, "Dynamic taint tracking for java with phosphor (demo)," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*, M. Young and T. Xie, Eds. ACM, 2015, pp. 409–413. [Online]. Available: https://doi.org/10.1145/2771783.2784768

[11] K. Pan, X. Wu, and T. Xie, "Guided test generation for database applications via synthesized database interactions," *ACM Trans. Softw. Eng. Methodol.*, vol. 23, no. 2, pp. 12:1–12:27, 2014. [Online]. Available: https://doi.org/10.1145/2491529

[12] J. A. Clause, W. Li, and A. Orso, "Dytan: a generic dynamic taint analysis framework," in *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2007, London, UK, July 9-12, 2007*, D. S. Rosenblum and S. G. Elbaum, Eds. ACM, 2007, pp. 196–206. [Online]. Available: https://doi.org/10.1145/1273463.1273490

[13] C. Harmon, B. Larsen, and E. A. Sultanik, "Toward automated grammar extraction via semantic labeling of parser implementations," in *2020 IEEE Security and Privacy Workshops, SP Workshops, San Francisco, CA, USA, May 21, 2020*. IEEE, 2020, pp. 276–283. [Online]. Available: https://doi.org/10.1109/SPW50608.2020.00061

[14] (2022) The sql-92 standard. [Online]. Available: http://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt

[15] C. Yang, Y. Li, M. Xu, Z. Chen, Y. Liu, G. Huang, and X. Liu, "Taintstream: fine-grained taint tracking for big data platforms through dynamic code translation," in *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, D. Spinellis, G. Gousios, M. Chechik, and M. D. Penta, Eds. ACM, 2021, pp. 806–817. [Online]. Available: https://doi.org/10.1145/3468264.3468532

[16] T. P. G. D. Group. (2022) Postgresql: The world's most advanced open source relational database. [Online]. Available: https://www.postgresql.org/

[17] J. Pearl, "Causal inference," in *Causality: Objectives and Assessment (NIPS 2008 Workshop), Whistler, Canada, December 12, 2008*, ser. JMLR Proceedings, I. Guyon, D. Janzing, and B. Schölkopf, Eds., vol. 6. JMLR.org, 2010, pp. 39–58. [Online]. Available: http://proceedings.mlr.press/v6/pearl10a.html

[18] (2022) itrust. [Online]. Available: https://sourceforge.net/projects/itrust/

[19] (2022) Riskit. [Online]. Available: https://sourceforge.net/projects/riskitinsurance/

[20] (2022) Unixusage. [Online]. Available: https://sourceforge.net/projects/se549unixusage/

[21] (2022) Odyssey. [Online]. Available: https://archive.codeplex.com/?p=odyssey

[22] (2022) Jforum. [Online]. Available: https://sourceforge.net/projects/jforum/

[23] T. Yu, R. Zhang, K. Yang, M. Yasunaga, D. Wang, Z. Li, J. Ma, I. Li, Q. Yao, S. Roman, Z. Zhang, and D. R. Radev, "Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task," in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, Brussels, Belgium, October 31 - November 4, 2018*, E. Riloff, D. Chiang, J. Hockenmaier, and J. Tsujii, Eds. Association for Computational Linguistics, 2018, pp. 3911–3921. [Online]. Available: https://doi.org/10.18653/v1/d18-1425

[24] K. Taneja, Y. Zhang, and T. Xie, "MODA: automated test generation for database applications via mock objects," in *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010*, C. Pecheur,

J. Andrews, and E. D. Nitto, Eds. ACM, 2010, pp. 289–292. [Online]. Available: https://doi.org/10.1145/1858996.1859053

[25] K. Pan, X. Wu, and T. Xie, "Program-input generation for testing database applications using existing database states," *Autom. Softw. Eng.*, vol. 22, no. 4, pp. 439–473, 2015. [Online]. Available: https://doi.org/10.1007/s10515-014-0158-y

[26] D. Faraglia. (2022) Welcome to faker's documentation! [Online]. Available: https://faker.readthedocs.io/en/master/

[27] (2022) A basic database management system written in java. [Online]. Available: https://github.com/gatesporter8/DBMS-SimpleDB-

[28] T. J. Authors. (2022) Jsqlparser. [Online]. Available: http://jsqlparser.sourceforge.net/

[29] T. A. Authors. (2022) Antlr. [Online]. Available: https://www.antlr.org/

[30] (2022) Taintsql: Dynamically tracking fine-grained implicit flows for sql statements. [Online]. Available: https://github.com/abc123000111/TaintSQL

[31] OceanBase. (2022) Oceanbase: Easier data management and use. [Online]. Available: https://www.oceanbase.com/en

[32] T. S. Authors. (2022) Sofastack: Financial-level distributed architecture. [Online]. Available: https://www.sofastack.tech/en/

[33] J. H. Perkins, J. Eikenberry, A. Coglio, and M. Rinard, "Comprehensive java metadata tracking for attack detection and repair," in *50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2020, Valencia, Spain, June 29 - July 2, 2020*. IEEE, 2020, pp. 39–51. [Online]. Available: https://doi.org/10.1109/DSN48063.2020.00024

[34] M. Sun, T. Wei, and J. C. S. Lui, "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, Vienna, Austria, October 24-28, 2016*, E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, Eds. ACM, 2016, pp. 331–342. [Online]. Available: https://doi.org/10.1145/2976749.2978343

[35] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B. Chun, L. P. Cox, J. Jung, P. D. McDaniel, and A. N. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Trans. Comput. Syst.*, vol. 32, no. 2, pp. 5:1–5:29, 2014. [Online]. Available: https://doi.org/10.1145/2619091

[36] B. Mathis, V. Avdiienko, E. O. Soremekun, M. Böhme, and A. Zeller, "Detecting information flow by mutating input data," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, G. Rosu, M. D. Penta, and T. N. Nguyen, Eds. IEEE Computer Society, 2017, pp. 263–273. [Online]. Available: https://doi.org/10.1109/ASE.2017.8115639

[37] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. D. McDaniel, "Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, M. F. P. O'Boyle and K. Pingali, Eds. ACM, 2014, pp. 259–269. [Online]. Available: https://doi.org/10.1145/2594291.2594299

[38] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard, "Information flow analysis of android applications in droidsafe," in *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*. The Internet Society, 2015. [Online]. Available: https://www.ndss-symposium.org/ndss2015/information-flow-analysis-android-applications-droidsafe

[39] W. Huang, Y. Dong, A. Milanova, and J. Dolby, "Scalable and precise taint analysis for android," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*, M. Young and T. Xie, Eds. ACM, 2015, pp. 106–117. [Online]. Available: https://doi.org/10.1145/2771783.2771803

[40] B. Kang, T. Kim, B. Kang, E. G. Im, and M. Ryu, "TASEL: dynamic taint analysis with selective control dependency," in *Proceedings of the 2014 Conference on Research in Adaptive and Convergent Systems, RACS 2014, Towson, Maryland, USA, October 5-8, 2014*, C. Lu, E. S. Nadimi, S. Kim, and W. Wang, Eds. ACM, 2014, pp. 272–277. [Online]. Available: https://doi.org/10.1145/2663761.2664219

[41] D. She, Y. Chen, A. Shah, B. Ray, and S. Jana, "Neutaint: Efficient dynamic taint analysis with neural networks," in *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 2020, pp. 1527–1543. [Online]. Available: https://doi.org/10.1109/SP40000.2020.00022

[42] M. G. Kang, S. McCamant, P. Poosankam, and D. Song, "DTA++: dynamic taint analysis with targeted control-flow propagation," in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011*. The Internet Society, 2011.

[43] Z. L. Chua, Y. Wang, T. Baluta, P. Saxena, Z. Liang, and P. Su, "One engine to serve 'em all: Inferring taint rules without architectural semantics," in *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society, 2019.

[44] J. Kreindl, D. Bonetta, L. Stadler, D. Leopoldseder, and H. Mössenböck, "Multi-language dynamic taint analysis in a polyglot virtual machine," in *MPLR '20: 17th International Conference on Managed Programming Languages and Runtimes, Virtual Event, UK, November 4-6, 2020*, S. Marr, Ed. ACM, 2020, pp. 15–29. [Online]. Available: https://doi.org/10.1145/3426182.3426184

[45] J. Kreindl, D. Bonetta, and H. Mössenböck, "Towards efficient, multi-language dynamic taint analysis," in *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes, MPLR 2019, Athens, Greece, October 21-22, 2019*, A. L. Hosking and I. Finocchi, Eds. ACM, 2019, pp. 85–94. [Online]. Available: https://doi.org/10.1145/3357390.3361028

[46] M. R. Azadmanesh, M. Hauswirth, and M. L. V. de Vanter, "Language-independent information flow tracking engine for program comprehension tools," in *Proceedings of the 25th International Conference on Program Comprehension, ICPC 2017, Buenos Aires, Argentina, May 22-23, 2017*, G. Scanniello, D. Lo, and A. Serebrenik, Eds. IEEE Computer Society, 2017, pp. 346–355. [Online]. Available: https://doi.org/10.1109/ICPC.2017.5

[47] M. L. V. de Vanter, C. Seaton, M. Haupt, C. Humer, and T. Würthinger, "Fast, flexible, polyglot instrumentation support for debuggers and other tools," *Art Sci. Eng. Program.*, vol. 2, no. 3, p. 14, 2018. [Online]. Available: https://doi.org/10.22152/programming-journal.org/2018/2/14

[48] J. Teoh, M. A. Gulzar, and M. Kim, "Influence-based provenance for dataflow applications with taint propagation," in *SoCC '20: ACM Symposium on Cloud Computing, Virtual Event, USA, October 19-21, 2020*, R. Fonseca, C. Delimitrou, and B. C. Ooi, Eds. ACM, 2020, pp. 372–386. [Online]. Available: https://doi.org/10.1145/3419111.3421292

[49] J. Wang, Y. Wu, G. Zhou, Y. Yu, Z. Guo, and Y. Xiong, "Scaling static taint analysis to industrial SOA applications: a case study at alibaba," in *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, P. Devanbu, M. B. Cohen, and T. Zimmermann, Eds. ACM, 2020, pp. 1477–1486. [Online]. Available: https://doi.org/10.1145/3368089.3417059

[50] G. Chinis, P. Pratikakis, S. Ioannidis, and E. Athanasopoulos, "Practical information flow for legacy web applications," in *Proceedings of the 8th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems, ICOOOLPS@ECOOP 2013, Montpellier, France, July 2, 2013*, O. Zendra, Ed. ACM, 2013, pp. 17–28. [Online]. Available: https://doi.org/10.1145/2491404.2491410

[51] Y. Mundada, A. Ramachandran, and N. Feamster, "Silverline: preventing data leaks from compromised web applications," in *Annual Computer Security Applications Conference, ACSAC '13, New Orleans, LA, USA, December 9-13, 2013*, C. N. P. Jr., Ed. ACM, 2013, pp. 329–338. [Online]. Available: https://doi.org/10.1145/2523649.2523663

[52] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek, "Improving application security with data flow assertions," in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, J. N. Matthews and T. E. Anderson, Eds. ACM, 2009, pp. 291–304. [Online]. Available: https://doi.org/10.1145/1629575.1629604

[53] J. Yang, T. Hance, T. H. Austin, A. Solar-Lezama, C. Flanagan, and S. Chong, "Precise, dynamic information flow for database-backed applications," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, C. Krintz and E. Berger, Eds. ACM, 2016, pp. 631–647. [Online]. Available: https://doi.org/10.1145/2908080.2908098

[54] A. Group. (2022) Lineage. [Online]. Available: https://www.alibabacloud.com/help/doc-detail/137497.htm