Ensuring Authorized Updates in Multi-user Database-Backed Applications

Kevin Eykholt, Atul Prakash, and Barzan Mozafari, *University of Michigan Ann Arbor*

https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/eykholt

This paper is included in the Proceedings of the
26th USENIX Security Symposium
August 16–18, 2017 • Vancouver, BC, Canada
ISBN 978-1-931971-40-9

Open access to the Proceedings of the
26th USENIX Security Symposium
is sponsored by USENIX

Ensuring Authorized Updates in Multi-user Database-Backed Applications

Kevin Eykholt
University of Michigan Ann Arbor

Atul Prakash
University of Michigan Ann Arbor

Barzan Mozafari
University of Michigan Ann Arbor

Abstract

Database-backed applications rely on access control policies based on views to protect sensitive data from unauthorized parties. Current techniques assume that the application's database tables contain a column that enables mapping a user to rows in the table. This assumption allows database views or similar mechanisms to enforce per-user access controls. However, not all database tables contain sufficient information to map a user to rows in the table, as a result of database normalization, and thus, require the joining of multiple tables. In a survey of 10 popular open-source web applications, on average, 21% of the database tables require a join. This means that current techniques cannot enforce security policies on all update queries for these applications, due to a well-known view update problem.

In this paper, we propose phantom extraction, a technique, which enforces per user access control policies on all database update queries. Phantom extraction does not make the same assumptions as previous work, and, more importantly, does not use database views as a core enforcement mechanism. Therefore, it does not fall victim to the view update problem. We have created SafeD as a practical access control solution, which uses our phantom extraction technique. SafeD uses a declarative language for defining security policies, while retaining the simplicity of database views. We evaluated our system on two popular databases for open source web applications, MySQL and Postgres. On MySQL, which has no built-in access control, we observe a 6% increase in transaction latency. On Postgres, SafeD outperforms the built-in access control by an order of magnitude when security policies involved joins.

1 Introduction

Stateful (server-side) applications often rely on a backend database to manage their state. When sensitive data is involved, these databases become prime targets for attackers. Web applications, especially, are subject to attacks due the large number of users and easy access through the Internet. To protect the sensitive data these web application store in thee database, proper access control is required. Unfortunately, securing web applications has remained a challenge, mainly for three reasons: (i) the incompatibility of modern web architecture and the security mechanisms of database systems, (ii) limitations of the automated techniques for enforcing a security policy, and (iii) failure to write secure, non-buggy code when implementing access contol logic within the application.

1. Architectural Incompatibility — Some database systems provide vendor-specific syntax for fine-grained access control [13, 16, 18, 23, 24] with support for security policies that involve joins. However, use of a specific database's access control mechanism makes the application DBMS-specific. A larger problem is that existing the web application architecutre is incompatible with the database access control architecture. Most modern web applications use an over-privileged database account with the authority to access and modify any information stored in the database [19, 21]. This setup is popular because it avoids the performance overhead of creating (and tearing down) new database connections on-the-fly for possibly millions of end users. Using an overprivileged account, the web application can simply maintain a pool of active database connections that can execute queries on behalf of any end user.

To use a DBMS' mechnisms, (1) each application user must be assigned a unique database account, and (2) a separate database connection (using the assigned account) must be used for processing each user's requests.[1] Making such changes to web applications would prevent

[1] This is because most databases, for security reasons, disallow [31] or limit [20, 23, 41] a connection's ability to switch its user context once it is created. Databases that allow but limit context switching for existing connections are still vulnerable to (1) application bugs in switching users, and (2) SQL injection whereby malicious users manipulate the functionality to switch to previous user contexts.

them from using a connection pool, and result in performance degradation [33, 48].

2. Limitations of Existing Techniques — The incompatability of DBMS access control with modern web application has resulted in numerous access control solutions, which exist as a security layer between the application and the database. These solutions restrict each application user to a portion of the database [39, 43, 36]. Before issuing a query, the application rewrites the query to use the restricted portion of the datbase based on the authenticated user. Often, *database views* are the central mechnism these systems rely on [39, 43]. Although current techniques can fully restrict database reads [43, 45], they do not support database updates (i.e., INSERT, DELETE, and UPDATE queries) due to the *view update problem* [27]. The view update probem states that write queries cannot execute on a view when there is not a "one-to-one relationship between the rows in the view and those in the underlying table" [22]. Such a problem can occur when a view definition contains a join query.

Consider OsCommerce, an open-source e-commerce web application, which allows customers to leave reviews on products. The metadata for reviews is stored in the *reviews* table. In OsCommerce, customers can only review products they have purchased. The following query represents the allowable set of rows that conform to this access control policy:

```
SELECT R.*
FROM review R,orders_products OP,
   orders O
WHERE   O.customer_id=current_id
   AND
        O.orders_id=OP.orders_ID
           AND
        OP.product_id=R.product_ID
           AND
        R.customer_ID=current_id;
```

The first three conditions in the WHERE clause obtain the set of products a customer has ordered, and the last condition ensures that the customer and the current user are the same. Although this view definition correctly captures the intended access control policy, it cannot be enforced with existing query re-writing techniques, as such a view is not updatable. This is because there is no one-to-one mapping between the rows in the view and those in the base tables, e.g., a user can purchase a product multiple times across different orders.

Previous work has largely ignored the view update problem by assuming that any table on which a security policy is defined contains the user id, thus joins are not required to map a user to rows in the table [36, 39, 43]. Unfortunately, as our survey of popular open-source web applications in this paper reveals, on average, 21% (and up to 50%) of the tables do not contain sufficient information to map a user to rows in the table due to a lack of a user id field or similar, thus a join query is required. In other words, existing access control solutions would not be able to fully support database updates for any of these popular applications.

3. Unsecure and Buggy Code — In today's web application architecture, developers cannot rely on databases to enforce access control policies due to the reliance of the applications on a pool of persistent connections. They cannot use existing access control solutions either, due to the lack of support for write queries when tables that do not contain user information. As a result, developers often implement their own access control login within the application and such implementations must be secure [47]. In theory, the application will only issues queries in accordance with the access control rules for the authenticated user.

In practice, however, most implementations have security flaws. According to a five-year study of 396 open-source web applications, over 200 security vulnerabilities were discovered [26]. Likewise, a study of vulnerabilities in open-source Java projects [12] found 8.61 defects for every 100,000 lines of code. Unsecure or buggy code leave web applications vulnerable to numerous access control bypass attacks, such as SQL injection [8, 25, 28, 34, 35, 38, 42] and insecure direct object reference attacks [14, 44]. These issues allow attackers to cause the application to issue unauthorized queries with respect to the current user and leak sensitive data. For example, a vulnerability in Symantec's web-based management console allowed authenticated low-privileged users to execute arbitrary SQL queries on the backend database, and change their account privileges to administrator level [25]. Data leaks have also occurred in mobile apps, such as Uber and SwiftKey, that use a database-backed web service [15, 30].

Such vulernerabilities occur because there is not a declarative way to define an access control policy within the application. Rather, developers have an idea of what the security policy should be and attempt to implement in code whenever database queries are issued. A proper access control solution should exist between the application and the database and allow a devloper to declaratively define an access control policy in a centralized location. Furthermore, the solution should meet the following key criteria:

C1. *Generality*: The access control policy can be enforced for all read (SELECT) and write (UPDATE, INSERT, and DELETE) queries on any table (whether it contains the user id as a column or not);

C2. *Correctness*: The application user should only be able to access and modify authorized information as defined by the developer's policy;

C3. *Database Independence*: The mechanism should not rely on vendor-specific features of the backend database; and

C4. *Connection Sharing*: For compatibility with existing web applications, the solution should allow for reusing a set of persistent and over-privileged database connections to serve requests of multiple end users.

Our Approach — We introduce the **phantom extraction** technique for enforcing access control on write queries, while being robust to policies that involve joins. Before executing a write query, we copy the rows from the target table that the user is authorized to modify into a temporary table. The query is then copied and modified to operate on the temporary table. We refer to modified copy as the query's **phantom**. Once SafeD deems the phantom's modifications on the temporary table to be safe, the changes are copied over to the original table. The view update problem does not apply to phantom extractions because database views are not used in any part of the process. The correctness of phantom extraction is established with a formal notion of *query safety* that guarantees a query is compliant with a given security policy (see Section 5). We present necessary and sufficient conditions to achieve that guarantee.

With phantom extraction, we created SafeD (Safe Driver), a pratical access control solution that supports policy enforcement for both read and write queries. SafeD extends existing database drivers, such as JDBC and ODBC, and transparently enforces an application's access control policy. Policies are defined by a set of declarative statements which use a syntax similar to datbase views. Since the access control is evaluated at the driver level, SafeD does not require a new database connection to establish a new user context, nor is it tied to a particular database backend. The user context is established when users authenticate themselves to the application, and SafeD enforces the access control policy for all database connections in the application's connection pool.

Contributions —

1. We surveyed 10 popular open-source web applications and show that complex row-level access control policies with joins queries are required for, on average, 21% of the tables to define per-user policies (Section 3).

2. We establish a formal notion of *query safety* and prove the necessary and sufficient conditions for the safety of all database operations, i.e., `SELECT`, `DELETE`, `INSERT`, and `UPDATE` (Section 5).

3. We present a new technique, **phantom extraction**, which ensures the safety of database updates with full generality (Section 6).

4. We present SafeD as a practical solution for enforcing per-user access control policies within the database. On MySQL (which lacks built-in support for row-level access control for read/write queries), a 6% increase in transaction latency is observed (Section 8.1). On Postgres, SafeD provides comparable performance to Postgres' access control for simple policies, but outperforms it by an order of magnitude for row-level access control policies with joins in terms of transaction latency and throughput (Section 8.2).

2 Related Work

The related work on access control can be categorized into application-centered versus database-centered approaches.

2.1 Application Access Control

CLAMP[43] and Nemesis [36] have similarities to SafeD in that each defines a per-user access policy in terms of views on the underlying tables. However, both works assume the underlying tables contain a column, such as a user id, which enables mapping a user to rows in the table. If the underlying table does contain a column, such as a user id [column], a join with one or more additional tables is necessary. For example, in OsCommerce, the security policy for *reviews* requires joining *reviews* with *orders* and *orders_products* to map a user to the set of reviews they can update (see Section 1). A view defined by a join query can result in the *view update problem* [27]. A database view is **updatable** only when there is a one-to-one mapping of rows in the view to rows in the underlying table. Therefore, CLAMP does not support per-user access control for write queries when the database view is not updatable. SafeD does not use database views to define per-user access control policies. Rather, SafeD rewrites queries to conform to the defined access control policy and executes the modified queries on tables in the database, thus avoiding the view update problem entirely.

In addition to assuming a table contain a column that enables mapping a user to rows in the table, Nemesis [36] also assumes that `INSERT` statements do not read existing rows in the database. However, this may not hold in many cases (e.g., consider `INSERT INTO` T1 `AS SELECT * FROM` T2). This query reads information from table *T2* and copies it into *T1*. In contrast, SafeD makes no such assumption and can handle both blind and nested `INSERT` queries.

Diesel [39] implements module-based access control, whereby an application is broken into a series of code

Solution	Generality	Correctness	Database Independent	Connection Sharing
Diesel [39]	x	x	✓	✓
CLAMP [43]	x	✓	✓	✓
Nemesis [36]	x	✓	✓	✓
Oracle [13]	✓	✓	x	x
Postgres [23]	✓	✓	x	x
SafeD	✓	✓	✓	✓

Table 1: Comparison of SafeD to existing solutions. (See Section 1 for criteria definitions.)

modules, each restricted to only a portion of the database needed to complete its task. While the authors remark that Diesel can be extended to user-based access control (e.g., by duplicating all the modules for each connected user), they also acknowledge that their solution would not scale [39] and suggest using database access control in conjunction. SafeD does not require database access control and is thus compatible with today's web architecture.

Table 1 summarizes SafeD's differences with prior work.

2.2 Database Access Control

Stonebraker and Wong presented the first database access control through query rewriting in INGRES [46], which supported read queries, but not write queries. INGRES' treatment of read access restrictions as predicates has been adopted by modern databases. For instance, Oracle's VPD allows administrators to define a series of functions for each relation based on the mode of access. These functions append a predicate to the query to enforce access rules based on the user context [29]. Defining these functions via procedural code offers flexibility, but is also more error-prone compared to simply writing declarative policy statements as in SafeD.

More recently, Postgres 9.5 has added support for fine-grained access control, whereby administrators define two policy conditions for each table and each role. The first condition is evaluated against `SELECT` and `DELETE` queries, while the second condition is evaluated for `INSERT` queries. (`UPDATE` queries are treated as a `DELETE` followed by an `INSERT`.) Postgres's design assumes that if users can view information, they should also be able to delete it. SafeD does not make this assumption, decoupling a user's read and write permissions.

As mentioned in Section 1, the key advantage of SafeD over access control features of database systems is that the former is compatible with today's web application architecture. Both Oracle and Postgres rely on the database connection to obtain user context. Web applications have avoided this approach due to performance implications of creating new database connections [33]. In contrast, SafeD allows applications to share connections across users. SafeD also provides database-independence, and offers a simple syntax for defining a security policy compared to database solutions. SafeD only requires an understanding of `SELECT` queries. Oracle and Postgres each use a different syntax, and require developers to understand more complicated concepts, such as creating system contexts, system context triggers, and policy functions. (See Table 1.)

3 Survey of Modern Web Applications

Modern web applications currently define and enforces access control policies within the code. MediaWiki, for example, stores user groups and the associated access control rules within a PHP config file [4], and the access control rules are enforced within the PHP functions. As evidenced by numerous attacks on web applications, the current apporach is flawed [8, 25, 28, 34, 35, 38, 42]. Thus, prior work has proposed alternatives that decouple access control logic from the application, but all existing work cannot handle write queries when a declarative policy definition requires a join. These types of policies, which we call **join policies**, occur when a database table does not contain a field corresponding to a user, such as user id, which enables a mapping of rows in the table to a user. To determine the prevalence of join policies in modern web applications, we surveyed 10 open-source Java web applications of varying size and complexity.

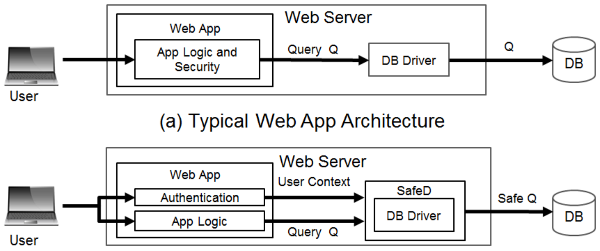
Before determining which tables require a join policy, we first must identify the user information table. Typically, the user information table contains a unique user ID that is used in other tables to map a row to a user. Given two tables, A and B, we say that table *A* is the **parent** of table *B* if *B* has a column that refers to the primary key of *A*. Similarly, given two tables, A and B, we say table *A* is the **grandparent** of table *B* if *B* has a column which refers to the primary key of a child of *A*. Often, these relationships are represented as a foreign key reference, but some of the applications we surveyed did not contain any such declarations. The lack of explicit foreign key declarations required us to infer the implied parent and grandparent relationships based on the database schema and structure.

In general, any table that has the user information table as its grandparent requires a join policy to define a per-user access control policy. Additional tables are included in our evaluation if accompanying documentation indicates a relationship between a user and a table despite no parent or grandparent relationship with the user information table. For example, in MediaWiki, pages can be semi-protected so only confirmed and autoconfirmed users[2] can modify them. In MediaWiki 1.10 and later, this information is stored in the *page_restriction* table. In

[2] Users whose account is at least four days old and has at least ten edits to Wikipedia

Web App	Total Tables	**Tables Requiring Join Policy**
Wordpress [10]	12	**4 (33%)**
hotCRP [40]	24	**6 (25%)**
LimeSurvey [3]	36	**18 (50%)**
osCommerce [7]	40	**4 (10%)**
MediaWiki [4]	48	**10 (21%)**
WeBid [9]	55	**5 (9%)**
Drupal [2]	60	**12 (20%)**
myBB [5]	75	**8 (11%)**
ZenCart [11]	96	**18 (19%)**
Cyclos [1]	185	**24 (13%)**
Average Percent		**21%**

Table 2: Summary of the number of tables in 9 web applications that require a join query to define a per-user policy declaratively.



(a) Typical Web App Architecture

Figure 1: Two web application architectures. A trusted authentication component within the app or on the server provides SafeD with the correct user context.

order to define a policy for the *page* table, a join with *page_restriction* is necessary to determine which pages a user can edit.

For each web application, we recorded the total number of tables in the database and the fraction of those tables that require a join query in a declarative policy definition to enforce a per-user policy.

Our survey results, shown in table 2, indicate that an average of 21% of an application's database tables require a join query to define a per-user policy declaratively. Web applications with a large amount of normalization tend to have a higher number of tables requiring a join query in their policy. LimeSurvey, which has the highest percentage of such tables, contains a user table with only a few children, but the children are heavily normalized resulting in numerous grandchildren. Zen Cart and Cyclos, which contain user tables with only a few children but multiple grandchildren, show similar trends.

4 Overview

Figure 1 shows the deployment architecture of SafeD. SafeD extends an existing database driver (e.g., JDBC or ODBC) to add a security layer that ensures all queries issued by the application are compliant with the defined declarative security policy (see Figure 2).

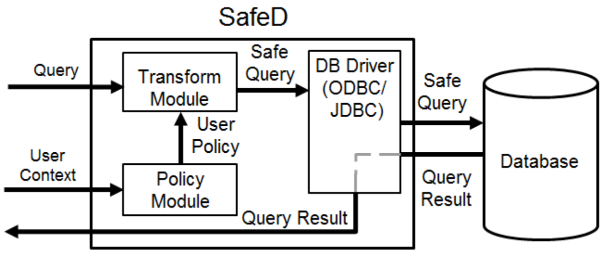


Figure 2: Given a query and a user context, SafeD obtains the user's security policy and creates a safe version of the query, which is executed on the database.

The application developer (or the system administrator) specifies the desired security policy via a set of declarative rules. These rules define the read and write permissions of each application user in the database (Section 5). At run-time, SafeD automatically transforms each query into a read-safe or write-safe query (i.e., one that is compliant with the read and write policies). SafeD provides Truman model semantics, i.e., the transformed query provides the same results as if the original query executed on a restricted view of the database that is accessible to the user. SafeD supports arbitrary read and write queries. SafeD's query transformation module (Section 6) ensures necessary and sufficient conditions for query safety. A developer can also use SafeD in an experimental debugging mode of operation, in which a query is tested for policy-compliance first. In this mode (a.k.a. non-Truman model [45]), a query is executed only if it is compliant, and is rejected otherwise (see Section 6). We implement our prototype by extending a JDBC driver (Section 7), and evaluate it using the TPC-C benchmark. Our results show that SafeD can protect practical database-backed applications at a negligible cost (Section 8).

4.1 Threat Model

We assume that a database app (e.g., web app) is benign, but buggy. We assume a remote attacker who attempts to exploit the web app, but cannot authenticate as another user. This is a reasonable assumption since most existing web frameworks, such as Django or Tomcat, have standardized support for authenticating users. In other words, the web application is assumed to reliably verify the end-user's identity and make it available to SafeD along with the issued query, but the query itself can be arbitrarily over-privileged, due to bugs or remote exploits.

Note that there are two causes of data leakage in a web application: incorrect policy definitions and incorrect policy enforcement. SafeD focuses on the latter, ensuring that all queries obey the developer-defined policy. However, if the policy is incorrectly defined, SafeD cannot prevent the undesirable actions of authorized users. One benefit of this decoupling is that developers are forced to explicitly define their security policies. These explicit definitions are often easier to debug than their implementation code.

5 Formal Results

In this section, we first describe the notion of per-user, row-level security policies. We then formally define the notion of safety for read and write queries. Finally, we derive necessary and sufficient conditions to achieve safety. These conditions are subsequently used to show correctness of our algorithms that render queries safe with respect to a given policy. Appendix B provides several examples of how safety can be enforced based on the results and definitions presented in this section.

5.1 Security Policy Definition

In SafeD, a security policy is composed of two sets of access rules: the *read policy* and the *write policy*. Given a user, the read policy identifies the tuples in the database that the user can read. Likewise, the write policy identifies the tuples in the database that the user can modify, remove, or add. These policies are specified as a *read set* and a *write set* for each table of the database. For each user and each table, a *read (write) set* identifies the set of tuples the user can read from (modify, remove from, or add to) that table. On a given table, the write set of a user must be a subset of his/her read set (i.e., users can read tuples that they can modify).

SafeD assumes that the authentication component of the web application provides the user's identity and, optionally, a 'role' assigned to the user. This role is only relevant to SafeD for selecting a policy and is not related to database roles. The user identity would usually be based on his/her authentication cookie and, possibly, the web request being made. SafeD takes as input a policy file comprised of a set of policy statements defining the read sets and write sets for each user and each role. The following are examples of policy statements for the customer and manager roles [3] :

```
DEFINE WRITESET FOR
    ROLE customer USER $i
     ON TABLE cust_info
          AS SELECT * FROM cust_info
               WHERE cid=$i
```

Listing 1: Customer's write set for the *cust_info* table.

```
DEFINE WRITESET FOR ROLE manager USER $i
     ON TABLE ordertable
          AS SELECT * FROM ordertable
```

Listing 2: Manager's write set for the *ordertable* table.

Here, $i is a wildcard that is replaced at runtime with the attribute(s) identifying the current user.[4] Read sets are defined similarly, except that the READSET keyword is used instead of WRITESET.

The definition of the read and write sets for a table may involve nested queries or joins with other tables. For example, suppose there are two additional tables in the database: *carts* and *cart_info*. The *carts* table maps a customer id to a cart id (cart_id), while the *cart_info* table contains the items in each cart. Since *cart_info* only contains cart_id, a join with *carts* is necessary to retrieve the cid. Listing 3 shows the *cart_info* table's read set for different customers.

```
DEFINE READSET ON ROLE customer USER $i
     ON TABLE cart_info
     AS SELECT * FROM cart_info x,
         carts y
         WHERE x.cart_id = y.cart_id
           AND y.cid=$i;
```

Listing 3: The read set for *cart_info* involving a join.

Next, we define read and write sets formally. In our discussion, an operation can be a SELECT, INSERT, DELETE, or UPDATE statement. (We use query and operation interchangeably.) We denote the set S as the **tuple space**, representing the (infinite) universe of all possible tuples. A **relational database** consists of a collection of tables. A **table** T is a finite subset of S. Since each element of a set is unique, we allow for duplicate entries by taking the Cartesian product of S with the natural numbers, $\mathbb{N} \times S$, and use that as our new tuple space. Duplicate entries will have a unique number in S. We also denote the number of tuples in a table T as $|T|$. Furthermore, given any subset $s \subseteq S$, we denote its complement as $s^c = S \setminus s$.

Given a set of **users** U, we define a **security policy** as a pair of two functions (p_r, p_w), where p_r is the **read policy** and p_w is the **write policy**, defined below.

Definition 1 (Read/Write Policy)**.** Given a user u, the read policy $p_r(u)$ is the subset of S that u is allowed to read. Likewise, the write policy $p_w(u)$ is the subset of S that u is allowed to add or modify.

A modification can be addition, removal, or update of a tuple. Based on the definition of a security policy, we now define the **read set** and **write set** for a user u.

Definition 2. (Read/Write Set) Given a user $u \in U$ and a table $T \subseteq S$,

- The read set of T, $V_r(T,u) = T \cap p_r(u)$, represents the set of tuples in T that user u can read.

[3] For simplicity, we define policies at the granularity of entire rows, but SafeD can be extended to finer granularities, e.g., at the attribute level.

[4] Currently, we assume the mapping of the current user to their role and identifying attribute is performed by the application.

- The write set of T, $V_w(T,u) = T \cap p_w(u)$, represents the set of tuples in T that user u can modify[5].

Note that the READSET and WRITESET statements introduced earlier correspond to these formal notions of read and write sets, given a table and user information, by simply instantiating the user identifier and applying their SELECT statements to T.

We also define the negated read set of T as $NV_r(T,u) = T \setminus V_r(T,u)$, which is the set of all tuples in T the user cannot read. Similarly, negated write set of T $NV_w(T,u) = T \setminus V_w(T,u)$, which is the set of all tuples in T the user cannot modify. It is trivial to show $NV_r(T,u) = T \setminus p_r(u)$ and $NV_w(T,u) = T \setminus p_w(u)$.

5.2 Safe Reads and Safe Writes

Now that we have formally defined a security policy and the read/write sets, we can formally define safe operations. We first consider read queries, which correspond to SELECT queries in SQL, and then write queries, which correspond to UPDATE, INSERT, and DELETE queries in SQL.

Definition 3 (Read-safety)**.** A query $\mathscr{R}$ by a user u with read policy p_r is **read-safe** if the query would return the same result when executed on the subset of tuples in the accessed tables that are readable to the user, namely $p_r(u)$.

In other words, a read-safe SELECT query should return the same result whether executed on the original tables or on the read sets of those tables. Note that $p_r(u)$ can, in general, be a set of tuples from multiple tables for queries with joins.

Corollary 1. *A query that only accesses tables whose tuples are all in $p_r(u)$ is read-safe.*

The above corollary implies the following: if in a query $\mathscr{R}$, each table T_i of the database accessed in the FROM clause is replaced by a table T_i' where $T_i' = T \cap p_r(u)$, then the resulting query $\mathscr{R}'$ will be read-safe. Such an approach has been proposed by previous systems [43, 46], and is also taken by SafeD. (As we will discuss shortly, enforcing safety for write queries is more challenging.) SafeD automatically transforms any SELECT queries (including those nested within other queries) by appending additional tables and conditions to the operation's FROM and WHERE clauses, respectively, that are implied by the READSET policy rules. We refer to this process as **read policy intersection**. Also, note that checking whether a query is read-safe can be more expensive than transforming it to be safe, since checking may require executing the query twice.

We next define the notion of read-safety and write-safety for a write query. As in SQL, we assume that a write query can read any set of tables (via nested SELECT statements), but modify only a single table and return, as its result, the modified table. Intuitively, a write query by a user u that updates a table T is **write-safe** if 1) it does not modify anything outside table T's write set, and 2) any nested SELECTs within it are also read-safe (so that it does not leak data via the writes). Formally,

Definition 4. A write query $\mathscr{W}$ by a user u that modifies table T is **read-safe** if all of its nested queries (which must be SELECTs) are read-safe. Furthermore, it is **write-safe** if it does not modify the *set* of tuples that are outside its write set for table T, i.e., $NV_w(T,u) = NV_w(\mathscr{W}(T),u)$, where $\mathscr{W}(T)$ represents the tuples in table T after executing $\mathscr{W}$.

Let $A = \mathscr{W}(T) \setminus T$ represent the new entries added by $\mathscr{W}$ to T, and let $D = T \setminus \mathscr{W}(T)$ represent the entries removed from T. It trivially follows that $\mathscr{W}(T) = (T \cup A) \setminus D$. For INSERT queries, D will be an empty set and for DELETE queries, A will be an empty set. For UPDATE queries, both A and D could be non-empty.

We denote $\langle \mathscr{W}(T) \rangle$ to be the sum of the cardinality of A and the cardinality of D for tuples added or deleted from T as a result of executing $\mathscr{W}$. It can be formally shown that the definition of write-safety does not require comparing $\mathscr{W}(T)$ with T, but only examining cardinality of changes. In particular, the following theorem can be shown to hold:

Theorem 1. *Given a user $u \in U$, a tuple space S, a set of tuples $s \subseteq S$, a table $T \subseteq S$, a write operation $\mathscr{W}$ that is read-safe, the write policy p_w, and the write set $V_w(T,u)$, the following conditions,*

$$\begin{aligned} &(1) \quad V_w(\mathscr{W}(T),u) = \mathscr{W}(V_w(T,u)) \\ &(2) \quad \langle \mathscr{W}(T) \rangle = \langle \mathscr{W}(V_w(T,u)) \rangle \end{aligned}$$

are necessary and sufficient to ensure $\mathscr{W}$ is write-safe, i.e.,

$$NV_w(T,u) = NV_w(\mathscr{W}(T),u).$$

Intuitively, condition (1) states that the resulting table from a write-safe write query should be the same whether the write is done on the original table T or on the write set $V_w(T,u)$. Condition (2) states that the total count of tuples added/deleted in T from executing $\mathscr{W}(T)$ should be identical to the count of tuples added/deleted if $\mathscr{W}$ was instead executed on $V_w(T,u)$. This ensures that $\mathscr{W}$ does not cause any tuples to be moved outside its write set as a result of changes. We defer the proof of Theorem 1 to the Appendix.

We now can define the notion of a query being *safe* in terms of read-safety and write-safety for the four types of queries addressed in this paper.

[5] Since an INSERT query adds tuples not in T, the write set is evaluated after the new tuples are added (See Section 5.2)

Definition 5 (Safe Query)**.** Given a policy (p_r, p_w), we consider a SELECT query for a user u to be *safe* if it is read-safe. We consider an `INSERT`, `DELETE`, or `UPDATE` query to be *safe* if it is read-safe and write-safe.

Corollary 2 (Safety of `INSERT`)**.** `INSERT` **queries:** *If all created tuples by an `INSERT` query are within the write set of the user, then the query is write-safe.*

Proof outline: In this case, the same tuple(s) will be added, irrespective of whether they are added to T or $V_w(T,u)$. Thus, conditions (1) and (2) in Theorem 1 hold.

Corollary 3 (Safety of `DELETE`)**.** `DELETE` **queries:** *A `DELETE` query that only deletes from $V_w(T,u)$ is write-safe.*

Proof outline: The query only deletes tuples in the write set so tuples in $NV_w(T)$ are not changed. Therefore, it trivially satisfies Definition 4. of Theorem 1 and takes advantage of the properties of `DELETE`.

Corollary 4 (Safety of `UPDATE`)**.** `UPDATE` **queries:** *An `UPDATE` query $\mathscr{W}$ that only updates tuples in $V_w(T,u)$ is write-safe if $\mathscr{W}(V_w(T,u)) \setminus V_w(T,u)$ is within the user's write set.*

Proof outline: An `UPDATE` can be thought of as a `DELETE` of the old tuples followed by an `INSERT` of the new tuples. From Corollary 3, we know the `DELETE` operation of the `UPDATE` is safe. If $\mathscr{W}(V_w(T,u)) \setminus V_w(T,u)$ is in the user's write set, then the `INSERT` operation of the `UPDATE` is write-safe due to Corollary 2.

The new value of an updated tuple has to be within the write set. If T is replaced by $V_w(T,u)$ in Theorem 1, it can be shown that the condition in the corollary implies both conditions of the theorem.

6 SafeD Design and Algorithms

SafeD operates as a modified JDBC driver that is transparent to the application. It transforms a submitted query into a safe query and returns the corresponding result. To do that, SafeD applies the Truman model [32] semantics for read queries, in which a query only sees tuples in its read sets. For write queries, SafeD uses a novel technique, called *phantom extraction*, to ensure only the portion of the table within the write set is updated (Section 6.1).

SafeD also provides an experimental (i.e., debugging) mode in which a read/write operation is carried out only if it is safe in the first place (a.k.a. "non-Truman" model [45]). Unfortunately, with the current state of the art, providing such a semantics is expensive. Consider a SELECT query. To know whether the query is safe, one needs to run the query on the original tables as well as their read sets and compare the results. Truman semantics avoid the need to execute the query twice since execution of the query on the original tables is not required. We prototyped this strategy and tested it with the TPC-C benchmark and found it to thrash at low transaction rates. We thus focus on the strategy of transforming queries to make them safe in rest of the paper.

6.1 Phantom Extraction

We say that a write query's **phantom** is a read-safe copy of the query, which only updates rows in the write set. In SafeD, write queries issued by the application are never executed on the database. Instead, each query's phantom is extracted and evaluated for write-safety. Phantom extraction involves 3 steps. First, transform the original query into a read-safe query using read policy intersection (Section 5.2). Then, modify the read-safe query so it only updates rows in the write set. This modified query is the *phantom*. Finally, determine if the phantom is write-safe. If the query's phantom is write-safe, the phantom's changes to the database are made permanent. Otherwise a permission violation error is returned and the changes are rolled back. In 6.2, we present two algorithms for phantom extraction.

6.2 Query Transformation Strategy

The transformation module automatically transforms a query Q into a safe query that is guaranteed to satisfy the two conditions in Theorem 1 for read-safety and write-safety, while providing the illusion that the query operates on the view of tables that are in the user's read set and write set. The algorithm we present can, at times, require issuing multiple queries to the database to check write-safety. Appendix B provides several illustrating examples of query transformation using the algorithms presented in this section.

Algorithm 1 shows the general transformation logic to transform a query Q issued by a user u. The queries currently handled by SafeD include `SELECT`, `INSERT`, `UPDATE`, and `DELETE` queries, which require row-level access controls. The transformation algorithm is a two-step process. First, SafeD must ensure that the transformed query is read-safe, i.e., $s \subseteq p_r(u)$ is true, where s is the set of tuples read by the query and gives the illusion that the query is running against the read set of accessed tables in the database (lines 2-5). Given a user u and a query Q, SafeD uses read policy intersection (Section 5.2) to create a read-safe query rsQ. Read policy intersection automatically transforms a `SELECT` query in Q into one that is read-safe by appending additional tables and conditions to each query based on the read policy.

Nested Queries (read-safety) — `SELECT` queries can be nested within other queries, including write queries. SafeD transforms them recursively to make them read-safe. Starting from the deepest sub-query, SafeD concatenates the associated read view predicates to the

WHERE clause of the sub-query.

Write Queries (write-safety) — Given *rsQ*, a read-safe transformation of *Q*, SafeD next executes the PhantomExtract function that results in a write-safe execution of the query using the phantom extraction technique (line 8). An input to PhantomExtract is the WRITESET definition that applies to this query (which is essentially a SELECT statement – see Section 5.1).

Algorithm 1 General Safe Execution Algorithm

```
1: function SafeExecute(User $u$, Query $Q$)
2:     $readpolicy \leftarrow$ GetReadPolicy($u$)
3:     $rsQ \leftarrow$ IntersectReadPolicy($Q, readpolicy$)
4:     if ($Q$ is a Select query) then
5:       return Execute($rsQ$)
6:     T $\leftarrow$ GetWriteTable($rsQ$)
7:     $writesetdef \leftarrow$ GetWriteSetDef($u$, T)
8: return PhantomExtract($rsQ$, T, $writesetdef$)
```

SafeD uses one of two strategies for implementing the PhantomExtract function: V-Copy or No-Copy. Both algorithms will result in only allowing permissible writes on the database. We present V-Copy strategy first.

The V-Copy strategy is shown in Algorithm 2. Instead of modifying T directly, an empty temporary table with the same schema as T is created in the database and the corresponding reference (i.e. the table name), tempT, is returned. The algorithm uses Corollary 2 to check the safety of INSERT (line 3), which means all inserts are performed on an empty table, *tempT*. For UPDATE or DELETE, the write set of T is added to *tempT* during initialization (lines 5-6). After initilizing *tempT*, *rsQ* is modified to execute on it, thus creating *phantom*, the phantom of the original query. After executing *phantom* on the database (line 8) , either (1) new tuples are inserted into *tempT*; (2) tuples are deleted from *tempT*; or (3) tuples are updated in *tempT*;. The check on Line 10 holds if the query's phantom is write-safe. The remaining lines of Algorithm 2 ensure that inserted or updated tuples have not gone outside the user's write set for table T (if they have, an exception is raised). Finally, T is modified based on the state of *addTup* and *rmTup*

Algorithm 2 V-Copy PhantomExtract

```
1: function PhantomExtract(Query $rsQ$, String T, WriteSet $writesetdef$)
2:     if $rsQ$ is an Insert Query then
3:         tempT $\leftarrow$ CreateTemp(T, null)
4:     else
5:         $authTup \leftarrow$ GetAuth(T, $writesetdef$)
6:         tempT $\leftarrow$ CreateTemp(T, $authTuples$)
7:     $phantom \leftarrow$ ChangeWriteTable($rsQ$, tempT)
8:     Execute($phantom$)
9:     $curTup \leftarrow$ GetAll(tempT)
10:    $authTup \leftarrow$ GetAuth(tempT, $writesetdef$)
11:    $rmTup \leftarrow \varnothing$;    $addTup \leftarrow \varnothing$
12:    if $curTup$ == $authTup$ then
13:        $authTup \leftarrow$ GetAuth(T, $writesetdef$)
14:        $addTup \leftarrow$ SetMinus($curTup, authTup$)
15:        if $rsQ$ is not an Insert Query then
16:            $rmTup \leftarrow$ SetMinus($authTup, curTup$)
17:    else   Raise permission exception
18:    Insert(T, $addTup$)
19:    Delete(T, $rmTup$)
```

An alternate strategy, No-Copy (Algorithm 3), can sometimes reduce the amount of work performed by the database and evaluates the conditions of Theorem 1 locally when possible. If the write set does not contain a join, No-Copy parses non-nested queries and determines if the query would result in tuples outside of the write set. This parsing can be always done for blind INSERT queries, which contain the new values for a tuple in the VALUE clause. Sometimes, the parsing can be done for UPDATE queries as well. If the UPDATE's SET clause does not assign values based on a computation, i.e., "attribute_name = function()", parsing can be performed. No-Copy creates a list of the attributes modified by the query and checks if any of the attribute are part of the write set's definition, i.e., contained in the WHERE clause of the write set. If so, then the value assigned to the attribute must satisfy the conditions defined in the write set. If the conditions are not satisfied, then the query will always result in tuples outside of the write set and is therefore not write-safe.

For DELETE queries, due to the Corollary 3, No-Copy executes the DELETE query on the subset of T that is within its write set, ensuring that only writable tuples are deleted.

Write set intersection is also used to transform *rsQ* into *phantom* if *rsQ* is an UPDATE. Since *rsQ* does not add tuples outside of the write set, *phantom* will not either, which means condition (1) of Theorem 1 is satisfied. Condition (2) requires that the number of modifications made to a table is equal to the number of modifications made in the write set of the table. Since the query's phantom only modifies tuples in the write set by definition, the number of changes made by *phantom* on T is equal to $\langle \mathscr{W}(V_w(T,u)) \rangle$ where $\mathscr{W}$ is the write operation representing *phantom*.

7 Implementation

We have implemented a prototype of SafeD by extending the JDBC driver. As previously shown in Figure 2, SafeD is comprised of two key modules: a transformation module and a policy one. The transformation module requires 317 lines of code in V-Copy and 452 lines in

Algorithm 3 No-Copy PhantomExtract

```
function QUERY $rsQ$, STRING T, WRITESET $writesetdef$)
    $phantom \leftarrow NullQuery$
    if $rsQ$ is not an Insert then
        $phantom \leftarrow$ IntersectWriteSet($rsQ, writeset$)
    else
        $phantom \leftarrow rsQ$
    if $phantom$ is a Delete then
      return Execute($phantom$)
    if ($phantom$ is a nested query) OR
        ($writeset$ contains a join) then
        Use Algorithm 2
    $attList \leftarrow$ GetAttributes($phantom$)
    if not(CanEvaluateLocal($modifiedList, writeset$))
then
        Use Algorithm 2
    $condList \leftarrow GetWhereConditions(writeset)$
    for all $a \in modifiedList$ do
        if $condList$.contains(a.name) then
            $pass \leftarrow$IsValidValue($condList$, a.value)
            if $pass == false$ then
                return Execute($NullQuery$)
return Execute($phantom$)
```

No-Copy. The policy module requires 119 lines.

Our policy module stores the read and write policies defined by the developer for each role, as well as a mapping between users and roles. Upon establishing a database connection, this module creates a connection state object that contains the security policy. When a user is identified, the module uses the supplied user context to initialize the read and write sets for the user. Given a SQL query and a user context, SafeD either verifies the compliance of the query before sending it for execution (in debug mode), or transforms it into a compliant query (in run-time mode).

8 Evaluation

Our experiments seek to answer the following questions:

1. What is SafeD's performance overhead for a database without built-in support for access control? (Section 8.1)

2. How does SafeD's performance compare to that of a built-in mechanism in a database that does support row-level access control? (Section 8.2)

3. How does SafeD's performance vary with the ratio of unsafe queries in the workload? (Section 8.3)

When studying SafeD's performance, we compare the V-Copy and No-Copy strategies. We experiment with both MySQL and Postgres. MySQL is perhaps the most popular open source database used by web applications, including several high-volume web sites, such as Facebook and Zappos [6]. However, given MySQL's lack of built-in support for row-level access control, web applications implement their own security policies. Postgres, on the other hand, offers row-level access control and thus provides a comparison point between a database-enforced access control with the costs of SafeD's approach. Postgres is also popular for small to medium-sized web applications [17].

Setup — In all experiments, we used two machines running Ubuntu 12.04 with 32GB of memory, configured as a client and a database server. The server had 8 CPUs (2.40 GHz each), while the client had 12 Xenon CPUs (2.67 GHz each). The client machine was used to send TPC-C queries to the database sever using the OLTP-Bench suite [37]. For TPC-C, we used its standard mixture of transactions (43% payment, 4% order status, 4% delivery, 4% stock-level, and 4% new order) and a scale factor of 20. For our database, we used MySQL 5.7 and Postgres 9.5.

Security Policies — Based on the semantics of the TPC-C benchmark, we used two different security policies. For both policies, there existed an administrator role with full read and write access to every table. In Policy 1, we defined two non-admin roles: a *manager* role and a *customer* role. A manager's user context contains two attributes: the warehouse id (WID) and the district id (DID). A customer's user context contains three attributes: the warehouse id (WID), the district id (DID), and the customer id (CID). Most of the database tables contain attributes that can be mapped directly to values in the user context. In these cases, a user is given read or write access (or both) to tuples where the user context matches the associated attributes in the tuples. When the target tables do not contain the necessary attributes to map the current user to tuples in the table, i.e, the *New_Order* and *Order_Line* tables, a join between the target table(s) and the *OOrder* table is necessary to obtain the set of order ids (O_ID) that the current user can access. The access rules for each role in Policy 1 are summarized in Table 3. We assigned each transaction type in TPC-C to one of the roles. Customers execute the new order, order status, and payment transactions, while managers execute the delivery and stock-level transactions.

Our second policy, Policy 2, tests the sensitivity of the performance results of SafeD and Postgres's row-level access control by adding restrictions to Policy 1. We modified the manager policy for the *OOrder* and *New_Order* tables as follows. First, a manager can read or modify tuples in the *OOrder* table only when the $O\_C\_ID \geq 0$. Second, a manager can only read and

Table Name	Customer	Manager
Customer(C_ID, C_D_ID, C_W_ID)	=(CID,DID,WID)	Full Access
District(D_ID, D_W_ID)	=(DID,WID)	=(DID,WID)
Warehouse(W_ID)	=(WID)	Full Access
OOrder(O_C_ID, O_D_ID, O_W_ID)	=(CID,DID,WID)	=(DID,WID)
New_Order(NO_O_ID)	Full Access	Full Access
Order_Line(OL_O_ID)	Full Access	Full Access
History(H_C_ID, H_D_ID, H_W_ID)	=(CID,DID,WID)	Full Access
Item	Full Access	Full Access
Stock	No Access	Full Access

Table 3: Policy 1 access rules for users. The user context is compared to the attributes in the table to determine if the user can read/write a tuple. Here, the read and write permissions are identical.

modify tuples in the *New_Order* table that correspond to authorized tuples in the *OOrder* table. Note that Policy 2 is still semantically equivalent to Policy 1 for the benchmark application since *O_C_ID* ≥ 0 is always true. However, the purpose of these constraints is to introduce artificial join constraints in the manager policy, and thereby assess their impact on SafeD's performance. Table 4 summarizes the change and shows how it alters the database account's privileges.

Table Name	Manager
OOrder(O_C_ID)	*O_C_ID* ≥ 0
New_Order(NO_O_ID)	Contain (OID) in OOrder

Table 4: Changes to Policy 1 to get Policy 2 and the new privileges for a database account. The changes result from modifications made to the manager role.

8.1 Performance Overhead of MySQL + SafeD

Since MySQL does not natively support row-based access control, we evaluated the overhead of adding access control to it using SafeD. Figure 3 shows the latency overhead on MySQL when SafeD verifies and enforces Policy 1 for varying transaction rates. The results show that SafeD can enforce a fine-grained security policy at a negligible cost to latency compared to having no protection (6.1% for No-Copy and 5.9% for No-Copy strategy).

8.2 Postgres + SafeD versus Postgres's Built-in Access Control

Unlike MySQL, some databases such as Postgres come with their own built-in row-level access control. The main advantage of SafeD over such built-in mechanisms is its compatibility with the common architecture of existing web applications (See Section 1). Nonetheless, we also wanted to compare the performance of the two approaches. We thus compared the costs of enforcing Policies 1 and 2 in Postgres using its internal access control versus using SafeD.

For Policy 1, to allow for reusing the same connections, we created a single role in Postgres for the benchmark application, and granted it the union of the privileges of all users so that the application can execute transactions on behalf of both customers or managers. The results are shown in Figure 4a, where NULL represents the baseline at which no access control was enforced. All three access control strategies (built-in, V-Copy, and No-Copy) had a maximum throughput of 350 to 400 transactions per second. Overall, the average latencies of all three strategies were also comparable (i.e., within 5% of one another). However, note that these results represent best-case scenarios for Postgres's built-in mechanism, since the benchmark application had full access to every table.

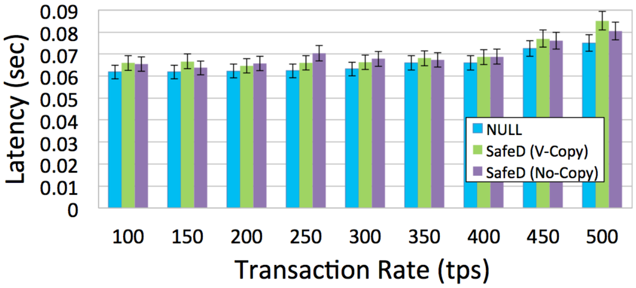
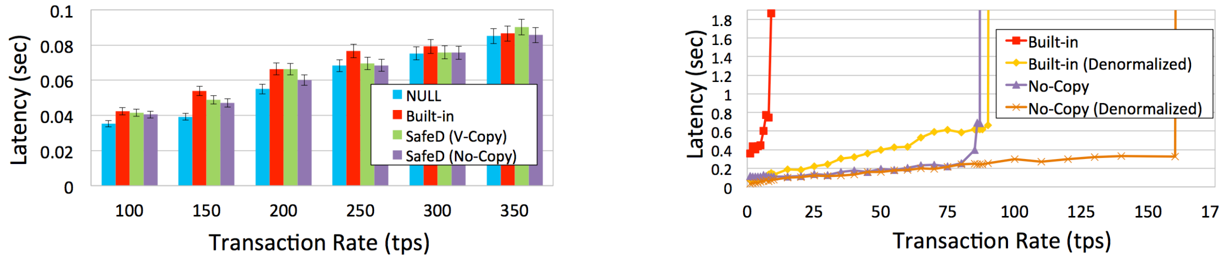


Figure 3: The performance overhead of different access control strategies compared to no access control (NULL) for TPC-C and Policy 1 on MySQL.

We conducted a second set of experiments using Policy 2 in order to artificially force both strategies (SafeD and built-in) to perform joins during their access control checks.

The results are shown in Figure 4b. In Policy 2, the write set for the *New_Order* table was defined as a join between the *New_Order* and *OOrder* tables. This considerably lowered the performance of V-Copy, due to its creation of temporary tables and copying of the write sets. Since V-Copy resulted in database thrashing and was unable to sustain any transaction rates, it is omitted from Figure 4b. The Postgres's throughput also dropped significantly with its built-in access control, down to only 9 tps (transactions per second). The throughput with No-Copy remained an order of magnitude higher, namely 85 tps. As reported in Table 5, even at 9 tps, Postgres's built-in mechanism is 387 times slower than SafeD when processing delivery transactions. The delivery transaction executes a large number of `SELECT` and `UPDATE` queries on the *New_Order* table. These results indicate that when a user's write set contains joins, SafeD using No-Copy significantly outperforms Postgres's built-in access control.

While SafeD outperforms the built-in access control, the performance of both strategies could be improved. In particular, we identified two sources of overhead when



(a) Postgres: Policy 1

(b) Postgres: Policy 2

Figure 4: The performance overhead of different access control strategies compared to no access control (NULL), using TPC-C on Postgres for Policy 1 (a) and Policy 2 (b). All numbers are average latencies.

Transaction	Null(s)	Built-in(s)	No-Copy(s)	Speedup
Delivery	0.05247	41.03159	0.10597	x387

Table 5: Transaction latency at 9 tps. Speedup is SafeD's performance compared to the built-in access control.

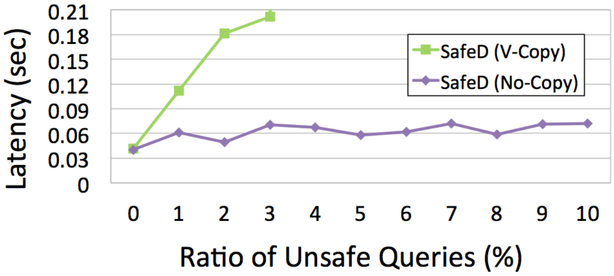


Figure 5: Average latency in SafeD for varying ratio of unsafe queries in the workload, at 100 tps.

enforcing Policy 2: (1) when the write set contains a join, a join query is issued to the database to create a copy of the write set; and (2) when the transformed query introduces a join or a nested sub-query. Thus, to reduce the performance overhead, we repeated the experiment with a denormalized database, i.e, we added a new column, NO_C_ID, to the *New_Order* table. As shown in Figure 4b, while the performance of both strategies improved significantly, SafeD remained the superior strategy.

8.3 Impact of Unsafe Queries on Performance

Unsafe queries are those that attempt to view or modify unauthorized tuples. In previous experiments, we measured SafeD's overhead when all queries in the workload were safe. To measure the impact of having unsafe queries on SafeD's performance, we modified the TPC-C workload by adding additional queries that are unsafe under Policy 1. We varied the ratio of such queries between 1% to 10% of the overall workload. The results for this experiment are shown in Figures 5 and 6.

As the number of unsafe queries increases, V-Copy's latency greatly increases, whereas No-Copy's latency overhead is relatively constant. This is because V-Copy creates temporary tables and executes additional queries to verify write-safety. Consequently, when 3% of the workload is unsafe, V-Copy thrashes. Figure 6 shows a similar trend for throughput. In conclusion, when a large number of unsafe queries are expected, No-Copy is a superior choice in terms of performance.

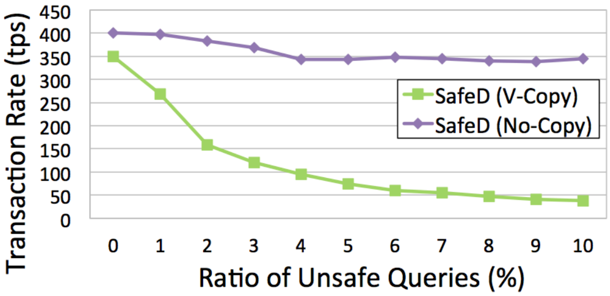


Figure 6: Achievable throughput in SafeD for varying ratio of unsafe queries in the workload.

8.4 Developer Effort

The primary manual effort required by developers when using SafeD is the defining of desired security policies. SafeD's policies are relatively compact. For example, Table 6 reports the number of lines of code needed to define Policy 1 in SafeD, Oracle (which also offers row-level access control), and Postgres. For SafeD, we count each read or write set declaration as one line of code. For Oracle, the count includes all of the procedural code necessary to establish the user context and enforce the read and write policies. For Postgres, we count each policy declaration and each ALTER TABLE command as one line. We also count the lines of code required to create an administrator role and a default role with no access.

We observe that Oracle requires the most lines of code, while Postgres and SafeD both require considerably fewer lines. Furthermore, Oracle requires the developer to understand how to define policy functions, policies, and system context triggers. Postgres requires

Access Control Mechanism	LOC
SafeD	36
Postgres's Built-in Access Control	54
Oracle's Built-in Access Control (a.k.a. VPD)	544

Table 6: Lines of code required to define a policy using three different syntaxes.

developers to work with DBAs to define policies and manage end-user roles. SafeD requires an understanding of SELECT statements to define policies. Thus, overall, defining security policies in SafeD seems to be relatively straightforward.

SafeD also simplifies developer effort when ensuring the application issues safe queries. To enforce a desired access control policy, developers add multiple security-oriented checks within the code to protect the database. For example, in WordPress 4.6.1, we identified about 515 lines of security-oriented checks in the code base. Each check is required to ensure no sensitive data is leaked, but there may be more checks necessary to fully protect the database [8], especially if the application's code is updated. SafeD reduces the effort required to protect the database because the security policies are declared explicitly within SafeD, thus they exist separately from the application and persist despite changes made to the application's code.

9 Conclusion

Database-backed application developers often implement their access control policies procedurally in code because the access control mechnisms of database systems are not adequate for enforcing access control for multi-user applications. Implementing access control procedurally in applicaton logic is both cumbersome and error-prone. Previous work has examined access control solutions for such situations, often using database views as the main mechanism for enforcing per-user access control. However, due to the view update problem, database views are not updatable when the view definition involves a join query. As our survey of 10 popular open-source web applications showed, on average, 21% of the tables require a join query to define a security policy. Therefore, previous work cannot enforce access control rules on write queries.

We proposed a new technique, phantom extraction that, given a write query, extracts a similar write query (known as the query's phantom), which only modifies permitted tuples in the database. Phantom extraction does not use database views, thus avoiding the view update problem. The correctness of the technique is established by a formal notion of *query safety*. We incorporated this technique into a system, SafeD, and provided a simple syntax for defining per-user (or per-role) access control policies declaratively. We also provide two possible design strategies, V-Copy and No-Copy, for performing query extraction.

References

[1] Cyclos: Online & mobile banking software. http://www.cyclos.org/.

[2] Drupal. https://www.drupal.org/.

[3] Limesurvey. https://www.limesurvey.org/.

[4] Mediawiki. https://www.mediawiki.org/wiki/MediaWiki.

[5] Mybb. https://mybb.com/.

[6] MySQL customers. https://www.mysql.com/customers/.

[7] oscommerce. https://www.oscommerce.com/.

[8] Sql injection vulnerability in ninja forms. http://tinyurl.com/z277h9f.

[9] Webid. http://www.webidsupport.com/.

[10] Wordpress. https://wordpress.com/.

[11] Zen cart. https://www.zen-cart.com/.

[12] Coverity scan open source report 2014. Technical report, 2014.

[13] Oracle database online documentation 12.1. http://tinyurl.com/jjgzavq, 2014.

[14] Snapchat - gibsec full disclosure. http://tinyurl.com/h6yk3za, 2014.

[15] Bug in Uber app leaks driver information. http://tinyurl.com/gtj5t54, 2015.

[16] Elements of row level security. http://tinyurl.com/jctpcll, 2015.

[17] PostgreSQL powers all new apps for 77% of the database's users. http://tinyurl.com/zlhnfuf, 2015.

[18] Row and column access control (rcac) overview. http://tinyurl.com/zavtmtx, 2015.

[19] Creating MySQL database and user. http://tinyurl.com/huv7uh7, 2016.

[20] Execute as (transact-sql). http://tinyurl.com/jjkd2fjFDB, 2016.

[21] Manual:security. http://tinyurl.com/qhylfza, 2016.

[22] MySQL internals manual. http://tinyurl.com/j8sou7y, 2016.

[23] PostgreSQL 9.5.0 documentation. http://tinyurl.com/hnjf7u6, 2016.

[24] Row-level security. http://tinyurl.com/jq7q2p2, 2016.

[25] Symantec patches high risk vulnerabilities in endpoint protection. http://tinyurl.com/zlgpfsg, 2016.

[26] R. Abela. Infographic: Statistics about the security scans of 396 open source web applications. http://tinyurl.com/zur7yfj, 2016.

[27] F. Bancilhon and N. Spyratos. Update semantics of relational views. *ACM Trans. Database Syst.*, 1981.

[28] D. Bisson. The talktalk breach: Timeline of a hack. http://tinyurl.com/jpp9epx, 2015.

[29] K. Browder and M. A. Davidson. The Virtual Private Database in Oracle9iR2. *Oracle Corporation*, 2002.

[30] S. Buckley. Swiftkey leaked user email addresses as text predictions. http://tinyurl.com/zj5wv37, 2016.

[31] cguler. Can I switch the 'connected' user within an sql script that is sourced by mysql? http://tinyurl.com/gv3rhwd, 2011.

[32] S. Chaudhuri, T. Dutta, and S. Sudarshan. Fine-grained authorization through predicated grants. In *ICDE*, 2007.

[33] I.-Y. Chen and C.-C. Huang. A service-oriented agent architecture to support telecardiology services on demand. *Journal of Medical and Biological Engineering*, 2005.

[34] C. Cimpanu. Teamp0ison hacks time warner cable business website, dumps customer data. http://tinyurl.com/zxvwjmj, 2016.

[35] A. Coyne. Hacker convicted for infiltrating country liberals' website. http://tinyurl.com/znfsnt3, 2016.

[36] M. Dalton, C. Kozyrakis, and N. Zeldovich. Nemesis: Preventing authentication & access control vulnerabilities in web applications. In *USENIX*, 2009.

[37] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudre-Mauroux. Oltp-bench: An extensible testbed for benchmarking relational databases. In *PVLDB*, 2013.

[38] D. Drinkwater. Up to 100k archos customers compromised by sql injection attack. http://tinyurl.com/jpv6mhj, 2015.

[39] A. P. Felt, M. Finifter, J. Weinberger, and D. Wagner. Diesel: Applying privilege separation to database access. In *ICCS*, 2011.

[40] E. Kohler. hotcrp. https://hotcrp.com/.

[41] A. Levai. Using queryband. http://tinyurl.com/hu2l6cj, 2014.

[42] J. Murdock. Qatar national bank leak: Security experts hint 'sql injection' used in database hack. http://tinyurl.com/h7ew4zf, 2016.

[43] B. Parno, J. M. McCune, D. Wendlandt, D. G. Andersen, and A. Perrig. Clamp: Practical prevention of large-scale data leaks. In *S & P*, 2009.

[44] I. Raafat. Vulnerability in Yahoo allowed me to delete more than 1 million and half records from Yahoo database. http://tinyurl.com/hb4jvn2, 2014.

[45] S. Rizvi, A. Mendelzon, S. Sudarshan, and P. Roy. Extending query rewriting techniques for fine-grained access control. In *SIGMOD*, 2004.

[46] M. Stonebraker and E. Wong. Access control in a relational data base management system by query modification. In *ACM Annual Conference*, 1974.

[47] N. Teodoro and C. Serrao. Web application security: Improving critical web-based applications quality through in-depth security analysis. In *i-Society*, 2011.

[48] S. Visveswaran. Dive into connection pooling with j2ee. http://tinyurl.com/hpfl9b9, 2000.

A Proof of Theorem 1

To prove Theorem 1, we first prove the following lemmas.

Lemma 1. ***Distributive Laws for Tables and their Read-/Write Sets***

The read and write set, $V \in \{V_r, V_w\}$, are distributive with respect to the basic set operations. That is, for any tables or other subsets $A, B \subseteq \mathscr{S}$,

$$V(A \cup B, u) = V(A, u) \cup V(B, u)$$
$$V(A \cap B, u) = V(A, u) \cap V(B, u)$$
$$V(A \setminus B, u) = V(A, u) \setminus V(B, u)$$

Proof. These follow trivially from laws for set operations since $V(A,u) = A \cap p(u)$, where p is p_r or p_w depending on V being V_r or V_w. We omit the details. For example, $V(A \cup B, u) = A \cup B \cup p(u) = (A \cup p(u)) \cup (B \cup p(u)) = V(A,u) \cup V(B,u)$.

Also, these results apply for $NV \in \{NV_r, NV_w\}$, since $NV(A,u) = A \setminus p_u(u) = A \cup p^c(u)$, where $p^c(u) = S \setminus p(u)$ and S represents the tuple space for the database.

□

Lemma 2. *Given a policy p_w, a user $u \in U$, a write set V_w, a table update operation $\mathscr{W}$, and any table $T \subseteq \mathscr{S}$, the following conditions are equivalent:*

$$\begin{aligned} &(1) \quad NV_w(A,u) = NV_w(D,u) = \varnothing, \\ &(2) \quad NV_w(T,u) = NV_w(\mathscr{W}(T),u) \end{aligned}$$

where A and D are the set of tuples added and removed from T.

Proof. Suppose that condition (1) holds. Then we find that

$$\begin{aligned} NV_w(\mathscr{W}(T),u) = NV_w((T \cup A) \setminus D, u) &\quad \text{def of } \mathscr{W} \\ = (NV_w(T,u) \cup NV_w(A,u)) \setminus \quad NV_w(D,u) &\quad \text{Lemma 1} \\ = NV_w(T,u) &\quad \text{by condition (1)} \end{aligned}$$

Conversely, suppose that condition (2) holds. Then

$$\begin{aligned} NV_w(A,u) &= NV_w(\mathscr{W}(T) \setminus T, u) && \text{def of } A \\ &= NV_w(\mathscr{W}(T),u) \setminus NV_w(T,u) && \text{Lemma 1} \\ &= NV_w(T,u) \setminus NV_w(T,u) && \text{condition (2)} \\ &= \varnothing \end{aligned}$$

The same approach also reveals that $NV_w(D,u) = \varnothing$.

□

Theorem 1. *(Same statement as in Section 5)*

Proof. **Part 1** — First, we show that the two conditions imply

$$NV_w(T,u) = NV_w(\mathscr{W}(T),u)$$

. We can partition the sets A and D into the disjoint portions consisting of those entries accessible to user u, and those that are not, giving

$$\langle \mathscr{W}(T) \rangle = |V_w(A,u) \cup NV_w(A,u)| + |V_w(D,u) \cup NV_w(D,u)|.$$

From their definitions, these are each clearly disjoint so that they may be separated into

$$\langle \mathscr{W}(T) \rangle = |V_w(A,u)| + |NV_w(A,u)| + |V_w(D,u)| + |NV_w(D,u)|.$$

If we define $A_v = \mathscr{W}(V_w(T,u)) \setminus V_w(T,u)$ and $D_v = V_w(T,u) \setminus \mathscr{W}(V_w(T,u))$, then from the definition of $\langle \mathscr{W}(T) \rangle$, we have

$$\langle \mathscr{W}(V_w(T,u)) \rangle = |A_v| + |D_v|$$

However, we also see that from making use of condition (1) we have

$$\begin{aligned} V_w(A,u) &= V_w(\mathscr{W}(T) \setminus T, u) && \text{def of } A \\ &= V_w(\mathscr{W}(T)) \setminus V_w(T,u) && \text{Lemma 1} \\ &= \mathscr{W}(V_w(T,u)) \setminus V_w(T,u) && \text{condition (1)} \\ &= A_v && \text{def of } A_v. \end{aligned}$$

Likewise, the same procedure reveals that $V_w(D,u) = D_v$.

Applying these two results, along with condition (2), we find that

$$\begin{aligned} |A_v| + |D_v| &= \langle \mathscr{W}(V_w(T,u)) \rangle \\ &= \langle \mathscr{W}(T) \rangle \\ &= |V_w(A,u)| + |NV_w(A,u)| + \\ &\quad |V_w(D,u)| + |NV_w(D,u)| \\ &= |A_v| + |NV_w(A,u)| + |D_v| + |NV_w(D,u)| \end{aligned}$$

Removing $|A_v|$ and $|D_v|$ from both sides, we are left with

$$|NV_w(A,u)| + |NV_w(D,u)| = 0.$$

But clearly, since both values are non-negative, this means that we must have

$$NV_w(A,u) = NV_w(D,u) = \varnothing$$

Hence by Lemma 2, we also have

$$NV_w(T,u) = NV_w(\mathscr{W}(T),u).$$

We have shown that conditions (1) and (2) imply this condition.

Part 2 — Now we will show that $NV_w(T,u) = NV_w(\mathscr{W}(T),u)$, implies both condition (1) and condition (2). Suppose that for an arbitrary update operation f,

$$NV_w(T,u) = NV_w(\mathscr{W}(T),u)$$

is true. We partition the set A into its disjoint portions consisting of the entries accessible to user u and those that are not, giving

$$\begin{aligned} A &= V_w(A,u) \cup NV_w(A,u) \\ &= V_w(A,u) \cup \varnothing && \text{by Lemma 2} \\ &= V_w(A,u) \end{aligned}$$

The same procedure shows that $V_w(D,u) = D$. With these results, we find that

$$\begin{aligned}
V_w(\mathscr{W}(T),u) &= V_w((T \cup A) \setminus D, u) && \text{def of } \mathscr{W}(T) \\
&= (V_w(T,u) \cup V_w(A,u)) \\
&\quad \setminus V_w(D,u) && \text{Lemma 1} \\
&= (V_w(T,u) \cup A) \setminus V_w(D,u) && V_w(A,u) = A \\
&= (V_w(T,u) \cup A) \setminus D && V_w(D,u) = D \\
&= \mathscr{W}(V_w(T,u)) && \text{def of } \mathscr{W}(V_w(T,u))
\end{aligned}$$

This shows that condition (1) is true. For condition (2), We again partition the sets A and D into the disjoint portions consisting of those entries accessible to user u, and those that are not, giving

$$\langle \mathscr{W}(T) \rangle = |V_w(A,u)| + |NV_w(A,u)| + |V_w(D,u)| + |NV_w(D,u)|.$$

Using Lemma 2, this simplifies to $|V_w(A,u)| + |V_w(D,u)|$. Focusing on $V_w(A,u)$, we find

$$\begin{aligned}
V_w(A,u) &= V_w(\mathscr{W}(T) \setminus T, u) && \text{def of } A \\
&= V_w(\mathscr{W}(T),u) \setminus V_w(T,u) && \text{Lemma 1} \\
&= \mathscr{W}(V_w(T,u)) \setminus V_w(T,u) && \text{condition (1)}
\end{aligned}$$

With a similar procedure, we can show that $V_w(D,u) = V_w(T,u) \setminus \mathscr{W}(V_w(T,u))$. If we define $A_v = \mathscr{W}(V_w(T,u)) \setminus V_w(T,u)$ and $D_v = V_w(T,u) \setminus \mathscr{W}(V_w(T,u))$, then from the definition of $\langle \mathscr{W}(T) \rangle$, we have

$$\langle \mathscr{W}(V_w(T,u)) \rangle = |A_v| + |D_v|$$

With this, we find

$$\begin{aligned}
\langle \mathscr{W}, T \rangle &= |V_w(A,u)| + |V_w(D,u)| \\
&= |\mathscr{W}(V_w(T,u)) \setminus V_w(T,u)| + |V_w(T,u) \setminus \mathscr{W}(V_w(T,u))| \\
&= |A_v| + |D_v| \\
&= \langle \mathscr{W}(V_w(T,u)) \rangle
\end{aligned}$$

Since conditions (1) and (2) imply $NV_w(T,u) = NV_w(\mathscr{W}(T),u)$ and vice versa, the conditions are necessary and sufficient.

□

B Query Transformation Examples

To better understand the transformation described in Section 6, we describe the steps of the No-Copy Strategy, which is comprised of Algorithms 1 and 3, using three example queries. In these examples, we use the osCommerce database schema and focus on queries that read or modify the *reviews* table. In osCommerce, a customer can read a reviews for any product but can only write reviews for products that the customer has purchased. After writing a review, the customer can also edit it. These policies can be expressed in SafeD as the read and write sets shown in Listings 4 and 5.

```
DEFINE READSET FOR ROLE customer USER $i
    ON TABLE reviews
        AS SELECT * FROM reviews
```

Listing 4: Customer's read set for the *reviews* table.

```
DEFINE WRITESET FOR
   ROLE customer USER $i
    ON TABLE reviews
        AS SELECT R.* FROM reviews
           R,
        orders_products OP,orders O
        WHERE
        O.customers_id=current_id
           AND
        O.orders_id=OP.orders_id
           AND
        OP.products_id=R.
           products_id AND
        R.customers_id=current_id
```

Listing 5: Customer's write set for the *reviews* table.

B.1 Select Query Example

Suppose a customer with current_id=2 manages to (e.g., by exploiting a bug in the application) cause the web application to issue the following query:

```
SELECT * FROM reviews
WHERE products_id IN (
    SELECT products_id
    FROM orders_products OP, orders
        O
    WHERE O.customers_id = 1 AND
    O.orders_id=OP.orders_id)
```

Listing 6: Original SELECT query issued by the application

First, SafeD must obtain the current customer's read policy and intersect it with the customer's query. The only tables appearing in this query are *reviews*, *orders*, and *orders_products*. Hence, SafeD only needs to obtain the read sets of these three tables (Alg. 1 line 2). The read set for *orders*, and *orders_products* are given in Listing 7 and Listing 8. For *orders*, a customer is only permitted to view their own order information. For *orders_products*, a customer is only permitted to view order product information for their own orders.

```
DEFINE READSET FOR ROLE customer USER $i
    ON TABLE orders
        AS SELECT * FROM orders O
            WHERE O.customers_id=
               current_id
```

Listing 7: Customer's read set for the *orders* table.

```
DEFINE READSET FOR ROLE customer USER $i
    ON TABLE orders_products
        AS SELECT OP.*
        FROM orders_products OP,
        orders O
        WHERE O.customers_id=
           current_id
        AND O.orders_id=OP.
           orders_id
```

Listing 8: Customer's write set for the *orders_products* table.

Since the query in Listing 6 is a nested query, SafeD performs read set intersections recursively, starting with the deepest sub-query (Alg. 1 line 3). As stated in Section 5.2, SafeD appends additional tables and conditions in accordance with the read set definition, thus transforming each SELECT query into a read-safe one. The original query is thus transformed into the following read-safe query and then executed (using Alg. 1 lines 4-5):

```
SELECT * FROM reviews
WHERE products_id IN (
    SELECT products_id
    FROM orders_products OP, orders
        O
    WHERE (O.customers_id = 1 AND
    O.orders_id=OP.orders_id) AND
    O.customers_id = 2 AND
    O.customers_id = 2 AND
    O.orders_id=OP.orders_id);
```

Listing 9: write-safe SELECT query created by SafeD

Note that, in the original query, the customer attempted to see reviews for products purchased by another customer with customers_id=1. Although the customer has full read access to *reviews*, it is a breach of policy for a customer to read another customer's information in the *orders_products* table.

B.2 Delete Query Example

Suppose a customer with current_id=2 causes the application to issue the following query:

```
DELETE FROM reviews
```

Listing 10: Original DELETE query issued by the application

First, SafeD obtains the customer's read policy for tables used in any SELECT's in the query, but there are no SELECT queries. This means, by definition, the current query is write-safe. SafeD then identifies the table modified by the query, *reviews*, and obtains the customer's write set definition for this table (Alg. 1 lines 6-7). SafeD passes the user context, the write-safe DELETE query, and the write set definition to PhantomExtract (Alg. 1 line 8).

Given that the current write-safe query is a DELETE, SafeD performs write set intersection by appending additional conditions to the outer query's WHERE clause (Alg. 3 lines 3-4). This results in the following query:

```
DELETE FROM reviews
WHERE customers_id = 2 AND
products_id IN (
        SELECT products_id
        FROM orders_products OP,
           orders O
        WHERE O.customers_id = 2
           AND
        O.orders_id=OP.orders_id)
```

Listing 11: Transformed write-safe DELETE query created by SafeD 2

Since the write set includes a join (see Listing 5, an additional nested query is added to obtain a list of products purchased by the current customer. This list represents the set of products the customer is allowed to reviews.

Since the transformed query is a DELETE, it is deemed safe and executed by SafeD (Alg. 3 lines 7-8). Note that the original query (Listing 10) attempted to remove all of the reviews in the database, but SafeD transformed it into a safe form, i.e., a query that only deletes the reviews of the customer with current_id=2.

B.3 Insert Query Example

As a last example, suppose a customer with current_id=2 causes the application to issue the following query.

```
INSERT INTO reviews
      (reviews_id, products_id,
       customers_id, customers_name
          ,
       reviews_rating, date_added,
       last_modified, reviews_read)
VALUES(-1, 1, 1, 'John',
       5, 1-1-2016, 1-1-2016, 50)
```

Listing 12: Original INSERT query issued by the application

Similar to the DELETE query example, the original INSERT query is read-safe by definition. SafeD identifies the table modified by the query, *reviews*, obtains the customer's write set definition for that table, and passes the user context, the write-safe INSERT query, and the write set definition to PhantomExtract (Alg. 1 lines 6-8).

Given that the current write-safe query is an INSERT, SafeD does not perform write set intersection (Alg. 3 lines 5-6) . Since the query is not a DELETE and the write set contains a join, Algorithm 2 is invoked (from Alg. 3 lines 9-10). SafeD creates an empty copy of *reviews*, which we will call *temp* (Alg. 2 lines 3-5). Then, it extracts the phantom of the INSERT query, by copying the write-safe query and executing it on temp (Alg 2. lines 8-9). After execution, SafeD determines if the rows added to and removed from *temp* both belong to the write set (Alg. 2 lines 10-12). Based on the query in Listing 12, we see that the original query adds a single row with customers_id=1. Therefore, the phantom adds a single row with customers_id=1 to *temp*, which is not in write set, thus the phantom is not write-safe. No modification is made to the *reviews* table (Alg. 2 line 17) and *temp* is dropped.