MultiverseMySQL: Multiverse Database Support for MySQL

Casey Nelson, *Brown University*

Abstract

MultiverseMySQL provides a proxy service that can be used with existing MySQL databases to apply privacy policies to user queries automatically. As users provide companies with increasing amounts of personal data, they also demand increasing assurance that this data is only being used in approved settings and following their agreed privacy policy. Currently, ensuring that these policies are followed requires developers to include many checks in their applications to ensure users are only ever viewing the data to which they have permission. However, these manual code checks leave room for errors that result in inadvertent data leaks. The goal of MultiverseMySQL is to move privacy checks from the application level to the database level as to minimize these mistakes.

1. Introduction

The data leaking application bugs that multiverse databases aim to prevent are not just hypothetical. There are many real examples of popular services that have possessed such vulnerabilities. One major example is Facebook, which had a multiple year long vulnerability which leaked users' synched, but not shared, photos. With Facebook, most applications should not be able to access all of a user's synched photos; they should only be able to access photos that a user has shared with Facebook explicitly. However, Facebook was only checking whether requests for synched photos had a certain permission; they were not checking if the application sending the request was actually trusted. As such, an adversary could create a malicious application to steal users' private photos as long as they gave it the right permission [6].

One of the major issues with Facebook's data leak mentioned in [6] is that the vulnerability existed for years uncaught. In fact, the bug was not even caught by Facebook itself. This speaks to the risk of large applications where privacy checks need to be placed in several locations. It is easy to overlook a missing privacy check that does not produce an obvious, glaring bug when privacy checks are located all throughout the application.

A multiverse database helps solve this kind of data leak by putting the privacy specification all in one location (the privacy policy), and by making the enforcement of the policy the job of the database instead of the application. For example, to solve the Facebook synched photo leak, the database can enforce that queries which do not come from the subset of trusted applications do not see any photos which were synched but not shared.

1.1. Threat Model

A multiverse database is meant to protect against the threat of application bugs and developer oversights that can lead to inadvertent data leaks. Multiverse databases on their own are not designed to protect against an actively malicious attacker attempting to leak data from the proxy or backend server. Data is not enforced to be sent or stored in any encrypted form. However, applying multiverse database features to a system like CryptDB [3] might be an opportunity for more complete database security.

1.2. Trusted Computing Base

The assumed TCB for a multiverse database is the proxy, the developer, and the database itself. There is no check in place to be sure the proxy is running the correct policy enforcing code. Additionally, there is no feature which ensures that privacy policies are valid. Finally, since there is no encryption by default, you must trust that the DBA is not snooping the database to read sensitive data.

However, a multiverse database does not require trusting the application because, even if the application issues unchecked requests, the proxy will intercept those requests and adjust them accordingly by following the privacy policy.

1.3. Multiverse Solution

The multiverse database approach, as outlined in [1], aims to prevent these potentially dangerous application errors by sandboxing a user's queries to only the subset of data that the user should have the ability to access. All user data is stored in what is referred to as the "base universe". However, when a user creates a query, it is directed to only a subset of the database, known as the "user universe". This is accomplished through an application defined privacy policy, which the multiverse database uses to determine what subset of the data a user should be able to view. The privacy policy represents the extent to which developers have to write data privacy code when using a multiverse database.

In general, the overall goals of a multiverse database, as introduced in [1], are to:

1. Move privacy enforcement out of the application and into the database
2. Ensure that a user's queries only see the data to which the user has access
3. Accomplish goals 1 and 2 with minimal overhead

1.4. Outline

The remainder of this paper is organized as follows:

Section 2 provides background for this work and discusses relevant papers that aim to accomplish similar goals.
Section 3 covers the specific multiverse database design used in this work including the format of the privacy policies and the role of the proxy.
Section 4 covers implementation specifics as to how MultiverseMySQL handles applying policies to both read and update requests to ensure that users' queries are being properly sandboxed.
Section 5 covers the experimental set up and evaluates the results.
Section 6 concludes with potential directions for future work.

2. Background

The concept of a multiverse database was first formalized in [1] as an opportunity to combine recent improvements in dynamic data flows with the traditional database concept of user views. The goal of the multiverse database designed in [1] was to apply privacy policies to the database to enforce that user queries go through their own user universe rather than the base universe of the database. This is a shared goal of MultiverseMySQL. However, the proof of concept presented in [1] uses data flows to improve performance. Their proof of concept was implemented using Noria, which allowed the base universe, the privacy policy, and the resulting user universes to all be represented in a single data flow. This work aims to provide equivalent functionality to MySQL database backends by using the features of the platform.

Similar functionality to multiverse databases has been achieved through query rewriting. For example, Qapla applies privacy policies by writing their restrictions directly into the queries [2]. Qapla additionally implements policies over specific combinations of tables or aggregations of tables, which has not yet been done in multiverse databases. The approach of applying privacy policies to queries by writing them directly into the query itself is functionally equivalent to the approach used in this work. However, MultiverseMySQL has some benefits over Qapla. Specifically, MultiverseMySQL allows restrictions on updates, which Qapla does not. MultiverseMySQL also allows the modification of column values rather than just the omission of columns which is done in Qapla.

3. Design

The design of MultiverseMySQL contains 2 major components. The first is the privacy policy and its syntax. The privacy policy is written by the developer, and it specifies what subsets of the database users should have access to when they make read or write queries. The second component is the proxy. In MultiverseMySQL, all requests to the database are first processed through a proxy. It is the proxy which applies the privacy policy to queries to ensure that they are sandboxed to the subset of data to which the user has permission.

3.1. Privacy Policy

The majority of a privacy policy for MultiverseMySQL is written using SQL syntax in a similar manner to [1]. The privacy policy definition language defines 4 keywords: `TABLE`, `ALLOW`, `REWRITE <column name>`, and `MOD`. `TABLE` specifies the table to which the following policy restrictions should apply. `ALLOW` and `REWRITE` both define read restrictions. `ALLOW` restricts access to rows and is specified using SQL's `WHERE` clauses. `REWRITE` allows the policy to blackout certain column values and is specified using SQL's `CASE` clauses. `MOD` is used to restrict updates to certain rows and is also specified using SQL's `WHERE` clauses. Currently, MultiverseMySQL's support of update policies is weaker than its support of read policies since updates can only be restricted at row granularity and not by column. However, this still builds on prior work which does not define or does not implement update restrictions [2][1].

Figure 1 shows an example policy for the table "posts" in a database based off of Piazza [5]. In this policy, students can only see public posts from classes with which they are associated. Additionally, all posts marked as anonymous are anonymized before being returned to a student. Instructors have the ability to read private and non-anonymized posts for their courses. Additionally, they have the ability to update the status of students in their courses. Here, the curly bracket syntax represents a value taken from the user's context to make the policy user specific.

```
TABLE :
    posts
ALLOW :
    WHERE ((visibility = 1 OR username = {username})
        AND class IN
        (SELECT class FROM people WHERE people.username = {username}))
    OR (class IN (SELECT class FROM people WHERE people.username = {username}
        AND people.position = 'instructor'))
REWRITE username :
    CASE WHEN anon = 1 AND
    username != {username} AND
    class NOT IN (SELECT class FROM people WHERE people.username = {username}
        AND people.position = 'instructor')
    THEN 'anonymous'
    ELSE username
MOD :
    WHERE (username = {username}) OR
    (class IN (SELECT class FROM people WHERE people.username = {username}
        AND people.position = 'instructor'))
```

Figure 1

3.2. Proxy

The bulk of the multiverse database logic is implemented in the proxy. With each proxy instance, the developer provides a policy file similar to what is shown in figure 1. The proxy then parses and stores this policy. With each user that connects to the database, the proxy takes the stored policy and creates a user specific policy by replacing items in curly brackets with the actual values from the user's context. The values stored in a user's context are configurable by the developer by modifying the `SqlUser` data type. Currently, only username is stored, but this can be modified to contain any relevant values that the database stores for a user. Recall that in this model the proxy is trusted, so providing it with user data is not a concern.

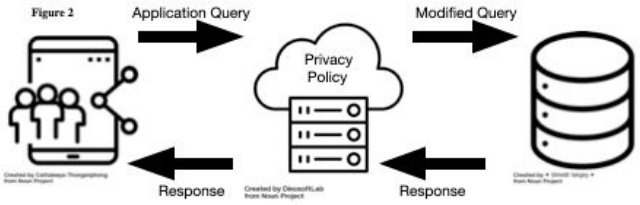


Figure 2

The overall design is depicted in figure 2. To sandbox user queries, the proxy translates the policy into a view, and replaces all mentions of tables protected by the policy with that user's view of the table. The views are created the first time a user references a given table, and a reference of the view is cached on the proxy until the user disconnects.

Implementing user universes with MySQL views is a space efficient implementation since MySQL views are simply stored as the queries themselves rather than duplicates of the data. However, a naive implementation of this will have a high performance cost because querying views becomes equivalent to Qapla's query rewriting technique, which had significant performance overheads [2].

4. Implementation

The testing implementation consists of a client and a proxy. The client allows users to enter SQL commands in a repl to be processed by the server. This is similar to the functionality of the standard MySQL command line client [7]. The client and the proxy exchange queries and responses using gRPC and Google's protocol Buffers [9][10]. The proxy processes these queries and forwards them to a MySQL server using the MySQL Connector library (for C++) [11]. The proxy also forwards responses from the server to the client when applicable. Overall, the client, proxy, and backend logic resulted in about 800 lines of C++ code.

Whenever the proxy receives a query, it first parses the command using the Hyrise SQL Parser [8]. If the command is a `SELECT` or `UPDATE`, it is further processed by the proxy. Otherwise, it is immediately forwarded to the server.

4.1. Reads

When the proxy receives a `SELECT` statement, it first identifies all tables referenced by the query and replaces them with their respective views for that user. The Hyrise SQL Parser can find any number of tables referenced by the query, including those used in joins or nested select statements [8]. The views that replace tables in a user's query are built off of the `ALLOW` and `REWRITE` statements from the privacy policy. The views are select statements of the underlying table where column values are selected based on `REWRITE` statements and rows are selected following the `ALLOW` statements.

After the proxy has modified the query, it then sends the updated request to the server and forwards the response to the client. The proxy does not need to do any further processing on the results received from the server.

4.2. Updates

When the proxy receives an `UPDATE` statement, it appends `WHERE` clauses defined by `MOD` statements in the privacy policy as applicable. If the update already has conditions, the proxy appends `AND` followed by the body of the `WHERE` clause defined in the privacy policy. The proxy then forwards the modified update to the MySQL server.

4.3. Optimizations

In the implementation described thus far, whenever a user queries a certain table, they are in fact querying several tables that might be cross referenced in the view query. For example, in the Piazza database, a user's view of the "posts" table is based on values stored in the "people" table. Having to reference external tables for every query leads to slowed performance. This performance overhead will increase with increasing database size and privacy policy complexity.

To minimize the overhead of querying from a view, MultiverseMySQL "simplifies" privacy policies in advance. Here, privacy policy simplification entails pre-computing the results of subqueries within the policy's conditions. The subqueries are then replaced with their result set or the empty set (' ') in the case that there is no result. For example, if a user, Alice, is a member of classes 1 and 2, the subquery `IN (SELECT class FROM people WHERE people.username = {username})` will resolve to `IN (1, 2)` in the privacy policy. This optimization was implemented independently from a similar optimization discussed, but not concretely implemented, in Qapla. However, in Qapla's suggested scheme for pre-computation, only user specific information that can be extracted from the application and is not dependent on the database can be pre-computed [2]. MultiverseMySQL does not limit pre-computation to this scope.

The current implementation of MultiverseMySQL makes the assumption that a user's privilege is static for the duration of the user's session. This is because the user specific privacy policy is computed and simplified once when a user logs in, so updates to a user's privileges while logged in will not be viewable to the user until the next session.

This assumption is acceptable for testing purposes because for many applications user privilege is updated so infrequently that the overhead of needing to dynamically update user specific privacy policies would not affect long term performance. For example, in Piazza, users' privileges would be modified about twice a year (once a semester) [5]. In HotCRP, user's privileges would be updated about once per conference [12]. The policy simplification scheme would be a less effective optimization for applications that update user privileges frequently. However, there are enough examples of applications with infrequent privilege updates to make this optimization viable.

5. Evaluation

Experiments were conducted on a Piazza style database [5]. This database included ~100,000 posts from ~50 users in ~100 different classes. All experiments were conducted on a VM running ubuntu 16.04. The client, proxy, and server were all hosted on the same machine to omit network latencies from the evaluation.

5.1. Experimental Setup

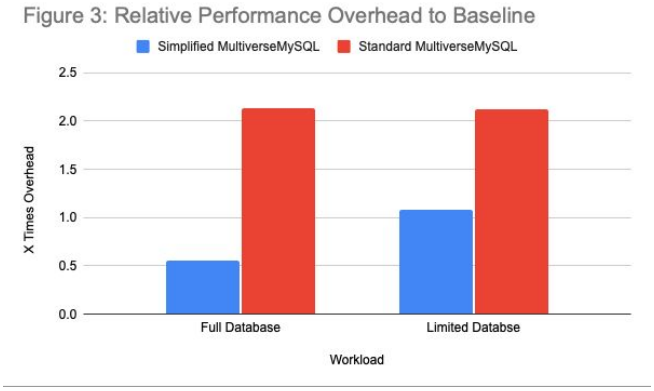
Experimental results were collected by running 10,000 `SELECT` queries through the proxy. There were 2 experimental schemes. The first generated 10,000 completely random queries over the entire database (full database). The second restricted these queries to only a subset of the database that the user could potentially have access to (limited database). For example, only selecting posts from classes that they are associated with or only selecting posts from users in similar classes. Experiments were run on 3 different implementations. The first was the baseline in which there were no privacy policies applied to the queries, the second was standard MultiverseMySQL, and the third was MultiverseMySQL with policy simplification. To help omit extraneous overheads, at most one row from the result set was returned from the proxy to the user. This was to help limit the additional overhead that the policy free implementation would acquire from having to return more results than the 2 multiverse implementations. However, there was no restriction on the number of rows sent from the server to the proxy.

5.2. Results

Results are depicted in table 1 and figure 3. The experiments showed that policy simplification provides the potential for a 2-4x throughput improvement over the standard implementation of applying the policy through unsimplified views. Additionally, the results showed that the policy simplification approach has the potential to only slow down query throughput by approximately 1 query per second, which is a very minimal overhead in comparison to the non-simplified implementation.

Implementation:	Full Database Throughput (queries/s)	Limited Database Throughput (queries/s)
Simplified MultiverseMySQL	35.21126761	16.83501684
Standard MultiverseMySQL	9.107468124	8.517887564
Baseline (No Privacy Policy)	19.45525292	18.11594203

Table 1



Note that the full database experiments show that the simplified policy implementation of MultiverseMySQL had performance improvements over the policy free baseline. However, these performance gains are artificially caused by the drastic increase of data that the server had to return in the policy free implementation. However, this does speak to the general performance gains of techniques that apply privacy policies on the query rather than on the results. By applying policies to the query, you prevent the database from having to return extraneous results which can improve performance.

6. Conclusion

MultiverseMySQL accomplishes all 3 goals of a multiverse database.

1. MultiverseMySQL allows developers to define a privacy policy in the proxy. This proxy intercepts queries going from the application to the database and applies the policy accordingly. This allows developers to minimize the amount of privacy checks they need to implement in the application because it is being handled by the proxy automatically.
2. By replacing the tables referenced in queries with the user's view of those tables as defined by the privacy policy, MultiverseMySQL ensures that the queries associated with a certain user are only seeing the data that the user is permitted to access.
3. By pre-computing the results of subqueries used to construct user views of the tables in a database, MultiverseMySQL doubles the query throughput in comparison to unoptimized query rewriting. Additionally, when subqueries are pre-computed, MultiverseMySQL only has about a 7% overhead when compared to having no privacy policy at all.

However, MultiverseMySQL also leaves open many potential avenues for improvements, especially in terms of strengthening its security guarantees. As mentioned previously, applying the logic used in MultiverseMySQL to a proxy used for an encrypted database, such as CryptDB, would allow guarantees to be made about the protection of data in storage rather than just while in use [3]. Additionally, in the current MultiverseMySQL model, you need to trust that the proxy is actually running the appropriate software to ensure that the privacy policy is being applied. However, attestation techniques similar to those used in Riverbed could allow the proxy to prove to the user or application that it can be trusted rather than the client having to blindly trust that fact [4]. The authors of [1] also suggested multiverse databases and differential privacy as a potential avenue for adding increased privacy guarantees. Overall, MultiverseMySQL successfully accomplishes all of the goals of a multiverse database while also remaining adaptable enough to broaden its use cases.

References

[1] Alana Marzoev, Lara Timbó Araújo, Malte Schwarzkopf, Samyukta Yagati, Eddie Kohler, Robert Morris, M. Frans Kaashoek, and Sam Madden. 2019. Towards Multiverse Databases. In *Workshop on Hot Topics in Operating Systems (HotOS '19)*

[2] Aastha Mehta, Eslam Elnikety, Katura Harvey, Deepak Garg, and Peter Drushel. 2017. Qapla: Policy Compliance for Database-backend Systems. In *26th USENIX Security Symposium*

[3] Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. 2011. CryptDB: Protecting Confidentiality with Encrypted Query Processing. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*

[4] Frank Wang, Ronny Ko, and James Mickens. 2019. Riverbed: Enforcing User-defined Privacy Constraints in Distributed Web Services. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation*

[5] Piazza Technologies, Inc. *Piazza*. Visited 12/2/2020. URL: https://piazza.com.

[6] Laxman Muthiyah. "How I Exposed Your Private Photos". *The Zero Hack*, January 2020. Visited 11/30/2020. URL: https://thezerohack.com/how-i-exposed-your-private-photos .

[7] MySQL. *mysql - The MySQL Command-Line Client*. URL: https://dev.mysql.com/doc/refman/8.0/en/mysql.html. Visited: 12/2/2020

[8] Pedro Flemming and David Schwalb. hyrise/sql-parser (commit 33c0013). [Source Code]. URL: https://github.com/hyrise/sql-parser

[9] Google. "Documentation". *gRPC*. Visited 12/2/2020. URL: https://grpc.io/docs/.

[10] Google. *Protocol Buffers*. Visited 12/2/2020 URL: https://developers.google.com/protocol-buffers.

[11] MySQL. *MySQL Connector/C++ 8.0 Developer Guide*. URL: https://dev.mysql.com/doc/connector-cpp/8.0/en/. Visited 12/2/2020

[12] Eddie Kohler. HotCRP. Visited 12/2/2020 URL: https://hotcrp.com/