RLS Side Channels: Investigating Leakage of Row-Level Security Protected Data Through Query Execution Time

CHEN DAR*, Tel Aviv University, Israel
MOSHIK HERSHCOVITCH*, Tel Aviv University, Israel
ADAM MORRISON, Tel Aviv University, Israel

Many modern use cases of relational databases involve multi-tenancy. To allow a tenant to only access its data, relational database systems (RDBMSs) introduced *row-level security* (RLS). RLS enables specifying per-row access controls, which the database enforces by rewriting tenant queries to add an RLS policy filter that filters out rows the tenant is not allowed to view. Unfortunately, while RLS blocks queries from returning unauthorized data, side-effects of query execution can form a side-channel that leaks information about such secret data.

This paper investigates how RLS *query execution time* can leak information about rows that the querying tenant is restricted from viewing. We show that in PostgreSQL and SQL Server, an attacker can craft index-using queries to learn whether a value they are not authorized to view exists in an RLS-protected table, and in some cases, how many times such a value exists in the table. Our attack succeeds in a realistic cloud setting: we successfully attack managed PostgreSQL and SQL Server database instances on AWS from virtual machines in the same and different data centers.

To block the RLS time side-channel, we design a data-oblivious query scheme for the case of unique keys. We also analyze the trade-offs created by the data-oblivious approach for non-unique keys.

To facilitate the evaluation of RLS attacks and defenses, we introduce a benchmark that supports multi-tenancy and RLS, which are not supported by established benchmarks such as YCSB. We implement our solution in PostgreSQL and show that it achieves security with minimal performance impact.

CCS Concepts: • **Security and privacy** → **Database and storage security**.

Additional Key Words and Phrases: Row-level security; time side-channel

ACM Reference Format:
Chen Dar, Moshik Hershcovitch, and Adam Morrison. 2023. RLS Side Channels: Investigating Leakage of Row-Level Security Protected Data Through Query Execution Time. *Proc. ACM Manag. Data* 1, 1, Article 89 (May 2023), 25 pages. https://doi.org/10.1145/3588943

1 INTRODUCTION

Modern use cases of relational databases often involve *multi-tenancy*, in which the database is shared by multiple tenants. Examples of multi-tenancy range from (1) software-as-a-service (SaaS) and web applications, which use a database to store data of multiple customers, each of whom should only access their own data; through (2) medical applications, which store patient data in a

*Both authors contributed equally to this research.

Authors' addresses: Chen Dar, Tel Aviv University, Israel; Moshik Hershcovitch, Tel Aviv University, Israel; Adam Morrison, Tel Aviv University, Israel.

2836-6573/2023/5-ART89 $15.00
https://doi.org/10.1145/3588943

database but restrict viewing it to authorized staff roles, to (3) monitoring and analytics applications, which display a dashboard showing a subset of data the accessing user is allowed to view.

Multi-tenancy requires enforcement of a security (or isolation) policy that prevents a tenant from accessing data they are not authorized to access. While these policies can be implemented in application logic, doing so is restrictive, imposes significant costs and maintenance burdens, and may be insecure (if the application logic is flawed). We elaborate on these problems in § 2, but the bottom line is that an architecture in which the policy is enforced by the database has many advantages over one in which the policy is specified in and enforced by client application code.

To support security policy enforcement by the database, relational database systems (RDBMSs) introduced *row-level security* (RLS) [24, 30]. RLS enables defining a policy that enforces access control on table rows. RLS policies are implemented by rewriting input tenant queries to add a policy filter. The filter removes rows the tenant is not authorized to view from read queries and blocks update queries the tenant is not allowed to perform. Thus, RLS provides a centralized location for defining table access controls with minimal changes to application code.

A significant challenge in preventing a tenant from obtaining information about rows they are not authorized to access is that of *side-channels*. While an unauthorized query may not explicitly return data from unauthorized rows, side effects of the query's execution might "leak" information about the unauthorized rows. For instance, RLS developers found that carefully crafted queries can leak information by throwing a divide-by-zero exception, thereby leaking that a row contains the value that makes the query divide by zero, and this side channel was addressed by hardening the database's error handling mechanism [17].

RLS time side-channels. In this paper, we explore a different RLS side-channel: query execution time. Specifically, we show that for index-using queries, RLS policy enforcement creates timing differences in query execution, which can be exploited by an attacker to learn information about rows they are restricted from viewing. At a high level, the problem stems from the RDBMS query optimization efforts resulting in evaluating the RLS policy filter on the results of the tenant's query. Consequently, the cardinality of the policy filter's input should be *secret*—it is the cardinality of the tenant's query output, which the tenant might not be allowed to know. For example, a tenant may query for patients with name X in a medical information table, without the tenant being authorized to view X's medical data. In this case, the size of the query's output—zero or non-zero—reveals information about X. But while the cardinality of the RLS filter's input should be secret, in practice, evaluating the RLS filter on each row of the input incurs measurable overhead. The query's overall execution time thus reveals information about the secret cardinality to the attacker.

We demonstrate that these time side-channels are not theoretical and can be exploited in a realistic cloud setting on two of the most popular RDBMSs [22], PostgreSQL and SQL Server. We successfully attack managed PostgreSQL and SQL Server database instances on Amazon Web Services (AWS) from remote virtual machines, in either the same or different AWS zones—a scenario modeling an attacker using machines on the same cloud provider its victim.[1] In both RDBMSs, we show that the attacker can abuse RLS to identify whether a value they are not authorized to view exists in an RLS-protected table; and in PostgreSQL, an attacker can also learn how many times such values exist in the table.

Mitigation. We propose a mitigation of RLS time side-channels for queries on unique keys (such as primary table keys), by making the execution time of such queries independent of the cardinality of the query's result (i.e., data-oblivious). We implement our mitigation in PostgreSQL and show that it blocks the side channel with minimal performance impact.

[1]The attack is related to the underlying PostgreSQL and SQL Server database engines. The attack is not specific to any Amazon-provided database. We only use AWS as a vehicle to demonstrate the attack under realistic conditions.

In contrast, we find that blocking the attack on non-unique keys seems to inherently involve significant overhead. The problem is that such mitigation requires making the execution times of queries on low- and high-cardinality data indistinguishable, which means significantly slowing down low-cardinality queries. As a result, we recommend not using RLS on non-unique keys.

Multi-tenancy benchmark. To facilitate the evaluation of our attacks and defenses, we introduce a benchmark that supports multi-tenancy and RLS, which are not supported by established benchmarks such as YCSB [10]. This benchmark models multi-tenant tables, RLS policies, and both authorized and unauthorized queries. As a result, the benchmark is useful for evaluating both the security and performance of RLS side-channel mitigations.

Contributions. To summarize, we make the following contributions:

(1) Demonstrating that exploitation of time side-channels in RLS is practical (§ 4).
(2) Introducing a solution for the side-channel for unique keys, and highlighting the trade-offs in blocking the side-channel when keys are not unique (§ 5).
(3) Introducing a new multi-tenancy database benchmark, which can be used to evaluate RLS workloads and side-channel mitigations (§ 6).
(4) Empirically evaluating the security and performance of our proposed mitigation (§ 7).

2 BACKGROUND: ROW-LEVEL SECURITY

Row-level security (RLS) is an RDBMS feature available in Microsoft SQL Server [24] and PostgreSQL [30]. RLS enables creation of access-control security policies on database tables that specify which rows each user is allowed to view or update. RLS policies are implemented by rewriting input user queries to add a *filter* based on the RLS policy. For read-only queries, the filter removes rows the user is not allowed to view from the query's result. For update queries, the filter blocks the user from modifying protected rows.

RLS-like object-level security mechanisms also exist outside the RDBMS domain, e.g., in Elastic's document-level security [13], MongoDB [1], and in generic application-layer solutions [2].

2.1 Motivation

The goal of RLS is to enable implementing row-level access control by the RDBMS, without relying on external conditions or application code. RLS confers several benefits [12, 34], discussed next.

Direct access. Providing access control at the RDBMS level means that tenants can be allowed to directly access the RDBMS interface, without requiring some intermediate application tier such as a web application for enforcing access control. The result is a significantly improved tenant experience. Tenants are not limited in how they access the RDBMS—say, with a custom service provider API—and can use standard database access tools. Tenants also enjoy an improved performance experience (latency and throughput) since their operations do not go through extraneous software layers.

Logical segregation. An RDBMS with RLS can be shared by multiple distrusting tenants, with the RDBMS using RLS to prevent tenants from observing and/or modifying each other's data. Service providers can improve performance and reduce costs and management overhead by using RLS to logically partition a single table among tenants instead of physically partitioning the table into per-tenant tables or deploying a different RDBMS instance for each tenant [21].

Trustworthiness. RLS centralizes the specification and enforcement of the access control policy in the RDBMS, without relying on any application tier or external systems. RLS thereby significantly reduces the amount of code that needs to be trusted for access-control integrity. This reduces the risk of bugs and security holes that could occur if access control were implemented in complex multiple end-user applications.

Expressiveness. RLS enables supporting complex access-control policies using standard RDBMS query languages, i.e., without requiring developers to "open code" the security policy rules and related data structures. RLS policies are limited only by the expressiveness of the RDBMS query language. A policy can be as simple as checking a user name match in one of the row's columns or be complex and involve querying multiple auxiliary tables, using recursion, etc.

Maintenance. Having the RDBMS enforce access control makes building client facing applications easier, as they do not need to support access control. Similarly, future access control rule changes can occur seamlessly, by updating the RLS policy, without requiring application-level code and/or configuration changes.

2.2 Implementation

Both Microsoft SQL Server and PostgreSQL RDBMS support RLS. We find that both implement the RLS functionality similarly. An RLS policy consists of two parts: one for read operations (`SELECT` and the read part of `UPDATE` and `DELETE`[2]) and one for write operations (`INSERT` and the write part of `UPDATE` and `DELETE`). Both parts are defined as filter predicates on queries, with the write filter defined to block a write operation if it does not comply with the predicate. In this paper, we focus on the read aspect of RLS policies and how it is implemented for `SELECT` operations.

When using RLS for read queries, each query targeting an RLS-protected table is rewritten to include the policy's filter as an additional predicate. The idea is that the policy filter predicate will be evaluated on every queried row, and if it evaluates to false for a specific row, that row will be filtered out of the query's response and not returned to the user. The RLS filter can query auxiliary tables in order to make its decisions; it is not restricted to querying the protected table's data.

Importantly, *when* the RLS filter runs is decided by the RDBMS' query planning mechanism—also called the *query optimizer*—which decides how to perform a query. The query optimizer determines the execution plan of the query and hence the execution position of the policy filter with respect to the other predicates in the query. The query optimizer's goal is to make the execution of the query efficient, by using indexes (when possible), removing duplicate or redundant filters, and ordering filters to improve performance.

2.2.1 Query Execution Security Considerations. The placement of an RLS filter with respect to the user-supplied predicates can affect security. In particular, a user predicate or function that runs before the filter predicate executes over all rows, including rows that the policy disallows the querying user from accessing. Certain types of predicates can create user-observable side-effects, such as printing or logging their inputs, or including the input in a thrown error message. Therefore, if such a user-supplied predicate runs before the RLS filter, it can circumvent the RLS policy and leak the contents of rows that the querying user is not allowed to access by the policy.

One approach for blocking this attack is to make the RLS filter special and require the query optimizer to run it first, which guarantees that user-supplied predicates only execute over rows approved by the RLS policy. This is basically the approach taken by PostgreSQL. Unfortunately, always evaluating the RLS filter first constrains the query optimizer and can dramatically reduce query performance (§ 2.2.2). To address this problem, PostgreSQL defines functions and operators that do not have side effects as *leakproof* [32]. Examples of leakproof operators include many of the comparison operators such as "=", "<", ">", and more. Leakproof predicates and functions (i.e., that consist only of leakproof operators) are allowed to run before the RLS filter, which gives the query optimizer greater flexibility.

[2]We refer to the initial query performed by these operations to check if a value exists before updating or deleting it.

For SQL Server, we do not find explicit documentation of security issues related to the placement of the RLS filter in the query execution order. When running query tests, however, we find that comparison operators can and do run before the RLS policy filter.

2.2.2 Query Execution Performance Implications. Evaluating the RLS policy filter before any user-supplied predicate can degrade query execution speed, because RLS filters typically have to be evaluated for every row in the table without benefitting from the system's indexes to optimize the query's performance. The reason is that having the RLS policy rely on indexed columns of the protected table restricts the expressiveness of the policy, allowing it to rely only on data stored in the protected table. Moreover, such a restriction may still create sub-optimal query execution plans [4]. For example, in the representative RLS use case described in § 2.3, executing the RLS filter first creates a large set of rows that now needs to be matched against the user's query. On the other hand, if the RLS policy does not rely on indexed fields, evaluating it requires a sequential scan of the entire table.

PostgreSQL addresses this problem with the use of leakproof operations (§ 2.2.1) which, for many user queries, enable the query optimizer to construct an execution plan under which the RLS filter executes on a small subset of the protected table's rows instead of on all of them. Specifically, the query optimizer usually chooses to run leakproof filters against an index before running the RLS policy filter. This process discards rows the user's query is not interested in, leaving the RLS filter to run only on rows that could potentially be returned by the user query.

In SQL Server, we do not find explicit discussion of how query execution order is affected by the existence of an RLS policy. Our examination of SQL Server's generated query plans, however, shows that it does not force the RLS policy filter to execute before other filters, similarly to PostgreSQL.

2.3 Running Example: Hospital Patient Data

We consider an example RLS use case, which we use as a running example throughout the paper. This use case is taken from user community of OpenEMR [3], a popular open-source electronic health records and medical practice management system. OpenEMR offers support for medical billing, clinical decision rules, etc. It is available for both on-premise and cloud-based deployments. The OpenEMR community has raised the access control problem when an OpenEMR system is shared by multiple clinics belonging to the same healthcare organization [6]. OpenEMR supports restricting viewing of the patient record table only to certain users (e.g., doctors), but it does not support *per-patient* access controls. OpenEMR thus allows a doctor to view any patient record in its database, which enables doctors to view records of patients at clinics other than their own.

Before the introduction of RLS to the backend RDBMS, the suggested way to solve the above challenge was by creating a separate database per site—with the related increase in computational requirements and management overhead. In contrast, RLS enables sharing a single `patients` table between multiple sites, with the RLS policy ensuring isolation of patient data between sites. Figure 1 demonstrates how a PostgreSQL backend can implement this approach. Lines 1–14 define the policy filter with the `site_policy` function. This policy excludes rows of patients that do not belong to the querying doctor's site; the site information is obtained from an auxiliary `doctors` table, not from the protected table. Lines 16–21 enable RLS protection of the `patients` table with this policy. Subsequently, any read query on the `patients` is rewritten to include the policy. For instance:

```
SELECT * FROM patients WHERE
Name='Lucía Lopez';
```

is rewritten by the query optimizer to:

```
SELECT * FROM patients WHERE
site_policy(site_id,current_user) and Name='Lucía Lopez';
```

```
CREATE FUNCTION
 site_policy(row_site int8, curr_user text)
RETURNS bool
AS $$
BEGIN
    return (
    (select site_id
    from doctors
    where user_name = curr_user)
        = row_site
    );
END;
$$
LANGUAGE plpgsql;

ALTER TABLE patients
ENABLE ROW LEVEL SECURITY;

CREATE POLICY doctor_read
ON patients FOR SELECT
USING(site_policy(site_id, current_user));
```

Fig. 1. OpenEMR multi-site scenario: PostgreSQL environment setup.

Notice that the execution order of `site_policy` and `Name` equality filter is determined by the query optimizer, because the equality filter is leakproof.

3 THREAT MODEL AND SECURITY GOAL

We consider an RDBMS offering services to multiple *tenants*. For example, a cloud-based RDBMS serving multiple customers (with customers directly querying the RDBMS) or a web application's database backend (with the application querying the RDBMS to respond to user actions).

The RDBMS stores data of different tenants in the same table(s), and the security goal is to isolate data between the different tenants using RLS. That is, each tenant should logically observe only a subset of the table dictated by the RLS policy and should not be able to learn *anything* about other rows of the table.

The threat under consideration stems from untrusted *users*. Attacks by the operating system (OS) or hardware are out of scope. The OS and hardware are assumed either to be trusted or protected using orthogonal mechanisms (e.g., structured data encryption [16, 19, 28]). Our threat model thus covers many typical RDBMS contexts, from in-house, on-premise systems to contemporary commercial cloud services [9, 37].

Attacker. The attacker is an authorized non-privileged user/tenant with read access to an RLS protected table. The attacker's goal is to learn information about rows that the RLS policy does not authorize it to view. For example, if we have a table of vaccinated people and the RLS policy allows users to view only rows that belong to their region, an attack would be for the attacker to determine if an ID or phone number of people outside of their region exists in the table.

Attacker power. We assume that the attacker can interact with the RDBMS only by querying it. In particular, the attacker cannot observe the RDBMS's storage access pattern or learn information via microarchitectural side-channels (e.g., via cache attacks [20]).

We further assume that the system is correct. That is, the attacker cannot compromise the RDBMS or the operating system hosting it and then run arbitrary code to download the entire protected table. Finally, we assume that the attacker cannot eavesdrop on queries performed by other users and/or on their responses.

4 RLS TIME SIDE-CHANNEL

We show that RLS policy enforcement creates timing differences in query execution, which can be exploited by an attacker as a *side-channel* through which they can learn certain information about rows they are restricted from viewing by RLS. We first describe the attack's goals, i.e., the kind of information it obtains (§ 4.1). We then describe the side-channel attack and the conditions sufficient for exploiting it (§ 4.2), and demonstrate exploitation of managed PostgreSQL and SQL Server database instances on AWS (§ 4.3).

4.1 Attack Goals

The goal of the attack is to learn information about the cardinality of protected table rows that match an attacker-supplied filter. (We use the term *filter* to refer to a boolean function that receives a table and returns a table composed of the *matching* rows on which the function evaluates to true.)

We define several notions of *data privacy* that our attack can violate, depending on the attack conditions:

(1) *Existence privacy*: An attacker should not be able to know if a value v exists in the table. Existence privacy is important, for example, to avoid leaking personal information, such as if a certain name or identifying number is present in a table of patients.
(2) *Cardinality privacy*: An attacker should not be able to tell how many times a value v appears in the table.

The above privacy properties are listed in increasing "magnitude" of privacy. For instance, breaking cardinality privacy implies we can also break existence privacy, because we can check if v exists in the table by checking if v appears more times than a value x known not to appear in the table (e.g., an illegal value). As we shall see, it becomes more difficult to breach these privacy properties as their privacy "magnitude" increases.

It may seem that breaching one of these privacy notions does not provide an attacker with significant information. However, the possibility of *compounding* an attack makes even these seemingly weak privacy notions important. For example, suppose we are able to violate existence privacy and determine that a patients table has a patient with ID x. We can then refine the attack filter and search for the existence of a patient with ID x and age y, for various y, and thereby eventually learn the patient's age. We can continue repeating this type of process and incrementally expose more information from RLS protected rows. In other words, compounding the attack enables translating a "weak" existence privacy breach into a much more significant privacy breach.

4.2 Attack Description

We analyze query execution time under RLS and show how, under certain conditions, differences in the execution time of specific queries—carefully chosen by the attacker—can reveal information about rows that the attacker is unauthorized to view by RLS.

Attack overview. Let T be the table protected by an RLS policy filter P. The attack invokes a query filter Q, to learn information about the cardinality of $Q(T)$. In a nutshell, the attack exploits query plans whose execution evaluates the filter Q and passes its output to the RLS policy filter P, which means that the cardinality of the row set passed to P is potentially secret. We find that applying the RLS filter imposes measurable overhead—sometimes even measurable per-row overhead—which enables the attacker to infer information about the (secret) number of rows on which the RLS filter was applied from the query's execution time, even through the filter's output is itself non-secret.

We now walk through the attack, explaining the required conditions to mount it as we go.

Requirement: Query plan evaluates Q before RLS filter P. The attack requires the system's query plan to evaluate the attacker's filter Q before it evaluates the RLS policy filter P. We find that in practice, if Q is defined on indexed columns, then the query optimizer will evaluate Q first, provided that Q is composed of "leakproof" operators. The reason is that from the query optimizer's perspective, evaluating P first would require a sequential table scan, whereas evaluating Q first does not and therefore produces a more efficient execution plan. We remark that even if the RLS policy P is also over indexed columns, an RLS policy typically also relies on data from the table T, which leads Q to run before P (see § 2.2.2). Indeed, forcing evaluation of the RLS filter to occur first would cause unacceptable performance overheads (see § 5.3.1).

Breaking existence privacy. We use $time(F(T))$ to denote the execution time of filter F on a table T. We assume that running any filter on an empty table takes zero time. The time it takes to perform the attacker's query Q is:

$$run\_time(T, Q, P) = time(Q(T)) + time(P(Q(T)))$$

Eq. 1: Query run-time calculation with RLS

Consider now two attacker queries, Q_k and Q_u, which filter for the keys k and u whose existence in the protected table T the attacker wishes to compare. To break existence privacy, k can be an invalid value whose cardinality is known to be zero. The difference in the execution time of Q_k and Q_u is:

$$\begin{aligned} delta &= run\_time(T, Q_u, P) - run\_time(T, Q_k, P) \\ &= time(Q_u(T)) + time(P(Q_u(T))) \\ &\quad - (time(Q_k(T)) + time(P(Q_k(T)))) \end{aligned}$$

We define:

$$\begin{aligned} query\_delta &= (time(Q_u(T)) - time(Q_k(T))) \\ policy\_delta &= (time(P(Q_u(T))) - time(P(Q_k(T)))) \\ delta &= query\_delta + policy\_delta \end{aligned}$$

Eq. 2: Query run-time delta calculation

We show that the following conditions are sufficient for $delta$ to encode sufficient information for comparing the cardinality of u and k in the table T:

(1) Negligible $query\_delta$. Formally: for any k, u in T, $abs(query\_delta)) < \epsilon$. As discussed below, query delta in practice is < 50 microseconds.
(2) Evaluation time of the RLS filter P is linear in the number of input rows.
(3) The per-row time of evaluating P is non-negligible (measurable). Formally: $time(P(T_r)) \approx \delta$, for any single-row table T_r.

If these conditions hold, we obtain the following, which enable using $policy\_delta$ to breach existence privacy:

- if $abs(policy\_delta) < \delta$, then the number of instances of k in T is equal to that of u.
- if $policy\_delta > \delta$, then number of instances of u in T is larger than of k.
- if $policy\_delta < -\delta$, then number of instances of u in T is lower than of k.

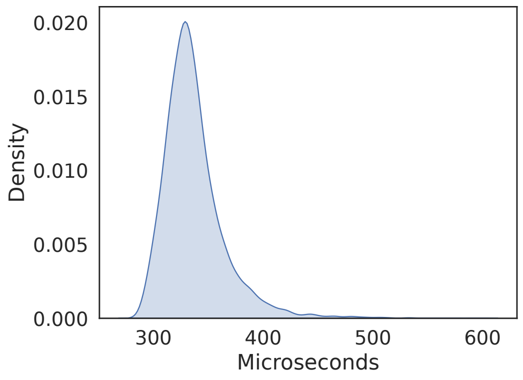


Fig. 2. KDE plot of PostgreSQL random SELECT query latency.

Because we know k does not exist in the table, the above enables determining if u exists in the table.

Next, we explain why these sufficient conditions for the attack hold.

Negligible *query_delta***:** There are several scenarios in which this condition holds. One example is when Q queries an indexed column(s) (as we assume) and the index lookups are satisfied from DRAM, which the attacker can arrange with high probability by running Q two times, so that the first execution brings in the index from disk if it is not already in memory.

When index operations are satisfied from DRAM, they are fast and have similar time (on the order of hundreds of CPU cycles) whether the queried key exists in the index or not. To demonstrate this effect, Figure 2 shows the latency distribution of PostgreSQL SELECT queries of random keys on an indexed column of a 2 million row table over an AWS data center network (the hardware setup is described in § 4.3). The figure shows that 80% of the queries take 310–360 microseconds, indicating that with high probability, the *query_delta* will be $<$ 50 microseconds.

Even when the columns Q queries are not indexed, *query_delta* can be negligible if Q is satisfied with a sequential scan and the queried table is small. In this case, *query_delta* is negligible because the scan time is negligible. In our experiments, *query_delta* was negligible for sequential scans of $<$ 1000 rows. Due to the limited applicability of this setting, the rest of our experiments and attack demonstrations consider the indexed column(s) setting. However, our proposed mitigation (§ 5) applies to small tables as well.

Measurable *policy_delta***:** Evaluation time of the RLS filter is indeed linear in the number of input rows, since filters are evaluated via a sequential scan of the input. Our main concern is therefore whether the time of evaluating the filter on a row is measurable. According to prior work [38], time differences as low as 150 nanoseconds and 10–50 microseconds are measurable over local and wide area networks, respectively. Such policy execution times arise if the RLS filter contains a non-trivial computation (e.g., recursion [36]) or if it is a SELECT operation on a dedicated policy table (because even the act of accessing the querying user's context to match against policy table rows incurs measurable latency). Dedicated policy tables are common, because they simplify management and reduce storage costs by creating an indirection level between the protected table and security-related information about users (e.g., each user's site id or access permissions). Without a dedicated table, all such information has to be stored in each row of the protected table. Indeed, using dedicated RLS tables is a recommended practice [27].

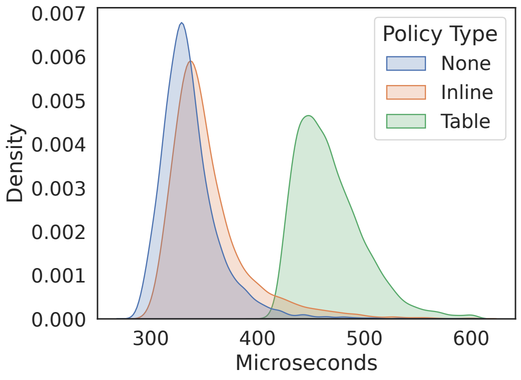


Fig. 3. Comparison of PostgreSQL SELECT query latency distribution with different policies over 5000 queries.

Figure 3 demonstrates the magnitude of RLS policy execution time in practice. It shows the latency distribution of SELECT queries on an indexed column of a 2 million row table under various RLS configurations, using PostgreSQL on the AWS setup described in § 4.3. We compare a baseline with no security policy (*none*) to two ways of deploying RLS: with the security information stored in the rows of the protected table (*inline*) or stored in a dedicated table (*table*). The *inline* policy in this comparison evaluates the *CURRENT_USER* of the session against the user name stored in the protected table column, and the *table* policy evaluates if the *CURRENT_USER* is found in the policy table and has read permission. In both cases, the user name column is indexed. Figure 3 shows that both *inline* and *table* have latency distributions that statistically differ from *none*. With *inline*, we find an observable average latency difference of $\approx$ 16 microseconds from *none*. With the common *table* configuration, the difference is an even more significant $\approx$ 130 microseconds.

Breaking cardinality privacy. Depending on the run time of the RLS policy and the cardinality of the set of rows that pass the attacker's filter, the query's execution time can be used to obtain a tight estimate on the cardinality. Simply, if the attacker knows that the execution time is about $n \cdot \delta$, where n is the (unknown) cardinality, they can divide by δ to obtain n.

4.3 Attack Proof-of-Concept

This section shows a proof-of-concept (PoC) demonstration of the RLS side-channel against managed PostgreSQL and SQL Server database instances in AWS. Our PoC models a realistic cloud setting, in which (1) the DBMS is fully managed by the cloud provider and only accessible via the network, even for its administrators; and (2) the DBMS virtual machine is hosted with other tenants on a shared physical host, in the production environment of a popular public cloud provider, and it experiences varying load conditions and competes for hardware resources with co-tenants. Moreover, AWS fully-managed DBMS provide an optimized deployment of the DBMS, which shows that the attack's success is not due to a DBMS misconfiguration.

Environment setup. For running the DBMS, we use AWS RDS [8] which provides fully-managed relational database services. We provision PostgreSQL and SQL Server on `db.m6i.large` instances, which have 2 vCPUs and 8 GB of DRAM. For the clients (attacker's machines), we use `t2.micro` instances with 1 vCPU and 1 GB of DRAM.

Client setup. To run our experiments, we use Python scripts on client virtual machines. We use the `psycopg2` and `pyodbc` libraries to interact with PostgreSQL and SQL Server, respectively. Time

PK	*Name*	*ID_Number*	*Age*	*Site_ID*
1	Anthony Moore	9788453650	24	0
2	Sofia Jones	8456601663	57	3
3	Zoey Martinez	2697388300	61	2

Table 1. Patients table example data.

PK	*UserName*	*Name*	*ID_Number*	*Site_ID*
1	lily978845	Lily Brown	3716565274	1
2	lily519804	Lily Jones	2697388300	2
3	joseph451687	Joseph Taylor	9719231653	3

Table 2. Doctors table example data.

is measured with the `perf_counter_ns` method, which uses the most accurate available system clock.

Database setup. We use a hospital patient data schema, taken from the OpenEMR community (§ 2.3). We set up a `patients` table and a `doctors` table. Table 1 and Table 2 show examples of these tables' data. We define an RLS policy limiting a doctor to only view patient data of patients from their site, as described in Figure 1.

We populate each table with 2 million random patient records and 10,000 random doctor records, split over five sites. We define the *id_number* column to have a unique index.

We purposefully target a simple table and RLS policy, because this simple scenario has the most efficient queries and hard-to-observe timing variance. Showing that the attack succeeds in this challenging scenario indicates that it will succeed against more complex tables/queries. § 4.3.3 provide evidence for this argument by evaluating the attack against a complex RLS policy.

4.3.1 PostgreSQL. We target PostgreSQL 13.7. We first target breaching existence privacy. We craft a query in the form of:

```
SELECT * FROM patients WHERE id_number=X;
```

The DBMS applies the RLS policy, transforming the query into:

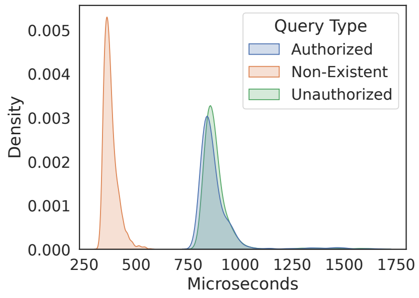
```
SELECT * FROM patients WHERE id_number=X
AND site_policy(site_id,current_user);
```

We verify that the query planner runs the filter $id\_number = X$ before the policy filter, due to the *id_number* column being indexed. To this end, we use the "explain analyze" command of PostgreSQL, which displays the query plan.

To perform the attack, we execute a benchmark that queries the patient's table *id_number* column for several types of values, which we denote as follows:

- **Non-existent**: Values that do not exist in the `patients` table.
- **Authorized**: Values in the `patients` table that the querying user is allowed to read.
- **Unauthorized**: Values in the `patients` table that the RLS policy restricts the querying user from reading.

Figure 4 shows the latency distribution of the different query types. Each query returns at most one row; time differences between non-existent and authorized queries are due RLS policy evaluation time. There is a clear separation of about 0.5 milliseconds between values that do not



(a) Distribution (KDE plot).

Query Type	Avg (μs)	Std. Dev
Non-Existent	377.25	34.31
Authorized	880.87	108.6
Unauthorized	889.7	101.55

(b) Average and standard deviation.

Fig. 4. PostgreSQL query latency of existence privacy breaching query, by queried value type.

Algorithm 1 Procedure for checking if `target_key` exists in the database (`nkey` is non-existent and `akey` exists the database).

```
1: procedure CheckIfKeyExist($nkey, akey, target\_key, rounds$)
2:     $rt\_not\_exists = RunQueryNAndGetMinElapsed(nkey, 10)$
3:     $rt\_exists = RunQueryNAndGetMinElapsed(akey, 10)$
4:     $threshold = rt\_not\_exists + (rt\_exists - rt\_not\_exists)/2$
5:     $runtimes = []$
6:     for $i := range(rounds)$ do
7:         $runtimes[i] = RunQuery(target\_key)$
8:     end for
9:     $guess = min(runtimes) > threshold$
10:    return $guess$
11: end procedure
```

exist and those that do exist—even if the user is not allowed to view them. This means that by measuring the run time of the query, the attacker can determine if a patient ID is in the `patients` table, even if the policy restricts the attacker from viewing that patient's data.

End-to-end attack. We demonstrate that the query time separation shown in Figure 4 can be exploited to determine if a *target* key exists in the database. Algorithm 1 shows our end-to-end attack procedure. We assume the attacker has prior knowledge of two keys: *nkey*, which does not exist in the database, and *akey*, which does. The attacker uses queries for these keys to calculate a threshold distinguishing a query for a non-existent key from an existing key. Next, they query for the target key and determine if it exists in the database by comparing the query's run time with the threshold. The attacker performs this experiment multiple times and makes a decision based on the minimum run time of the query.

To evaluate the attack's practical relevance, we emulate a production environment in the following ways. First, we introduce external load on the DBMS: while our experiments are running, the DBMS also serves fluctuating YCSB workloads running concurrently on other machines. Second, we take into account effects of varying load on the DBMS machine: we perform the attack every 3 minutes over a 24-hour period of a business day. Finally, we take into account network delay effects: we perform the attack from a virtual machine in the local and a remote "zone" as the DBMS. (The distance between AWS zones is up to 100 kilometers.)

Figure 5 reports the attack's success rate in determining the status of 100 random keys (some of which exist in the database and some do not) for a DBMS in the AWS `eu-west-1` region. The

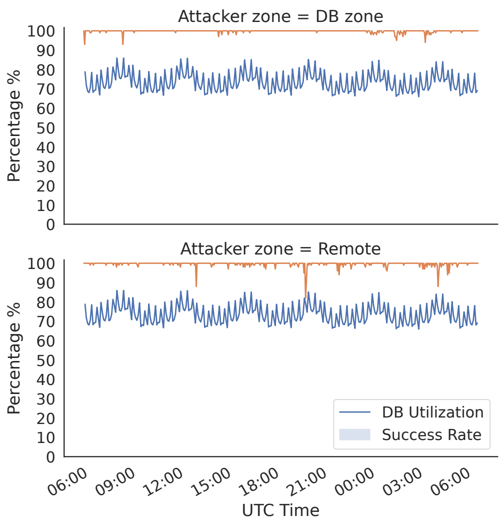


Fig. 5. End-to-end attack success and database load over a 24-hour period. Top: attacker machine is in the same zone as the database. Bottom: attacker machine is in a remote zone.

overall success rate after 10 queries for each key is 99.77% and 99.63% for attacks originating from the local and a remote zone as the DBMS, respectively. Moreover, the attack succeeds for varying times of day and fluctuating DBMS loads.

Having demonstrated that query time separation translates into a full attack, our following attack demonstrations show only query times (still against the AWS managed PostgreSQL service).

Attack compounding. We demonstrate compounding of the attack—that is, having found that a specific patient whose data we are not authorized to view exists in the `patients` table, we will learn information about their age. To this end, we craft a filter that depends on a patient's ID and some predicate over the age, such as whether age is greater than y:

```
SELECT * FROM patients WHERE
id_number=X AND function(age,y);
```

PostgreSQL's query planner places the age filter before the RLS policy filter, allowing the attacker to search for the age of a patient with a specific ID. Figure 6 compares the distribution of query execution times for patients whose record satisfies the age predicate to patients which do not satisfy it, for various types of predicates (equality, greater than, etc.). For every predicate, there is a significant separation in execution time that depends on whether the patient's record satisfies the predicate, allowing the attacker to learn whether the patient's age satisfies the predicate.

Cardinality privacy. Finally, we demonstrate a breach of cardinality privacy. We use a table in which patient ages are first distributed between 1 to 100 such that $Age = 1$ appears 100 times, $Age = 2$ appears 200 times and so on, until $Age = 100$ which appears 10,000 times. The remaining rows are then assigned random ages between 101–120. We craft the following query:

```
SELECT * FROM patients WHERE Age=Y;
```

We run this query 20 times for each age between 1 and 100. Figure 7 shows the average query execution time as a function of the cardinality of the queried age in the table. The linearity of the execution time shows that cardinality information can be easily derived.

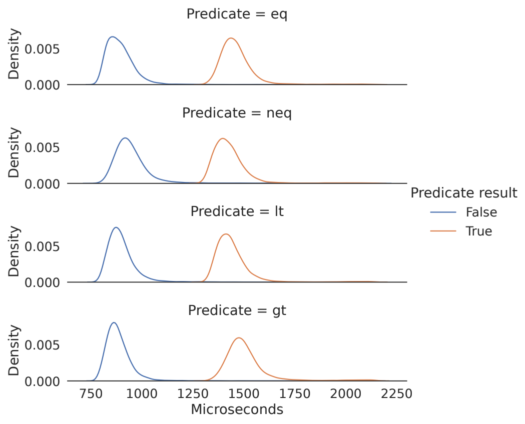


Fig. 6. PostgreSQL query latency distribution of predicates over a specific patient's age. "Predicate result" refers to whether the predicate returns true.

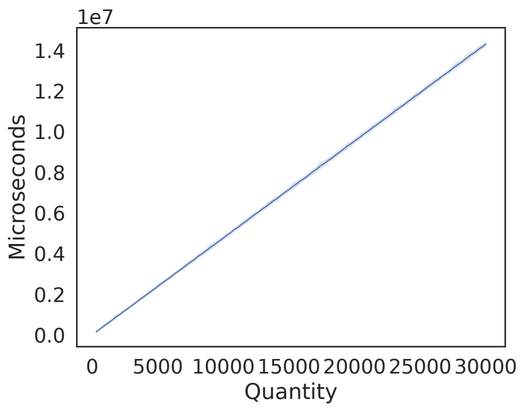
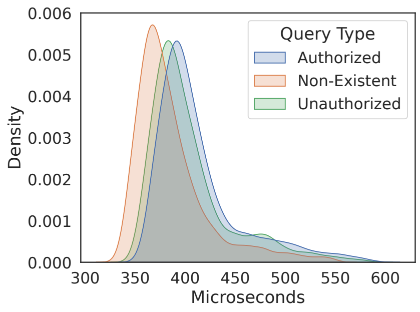


Fig. 7. PostgreSQL query latency of the cardinality privacy breaching query, as a function of the cardinality of Age value in the patients table.

4.3.2 SQL Server. We target Microsoft SQL Server 2019, Standard Edition. We perform the same benchmarks as in PostgreSQL.

Existence privacy. Figure 8 shows the latency distribution of the existence privacy breaching query. While the run time results are closer than with PostgresSQL, it is possible to distinguish values that do not exist from those which exist but the querying user is unauthorized to view. Figure 8b also shows IO statistics [23] which can be collected from SQL Server. (An attacker cannot view these statistics, we show them only to explain the results.) We report the number of "logical reads" performed by the query. These are 8 KB page reads from the SQL Server cache stored in DRAM. We see that an Unauthorized query performs the same amount of IO as the Authorized query, while a query for the non-existent value performs fewer logical reads on `patients` table and no reads on the `doctors` table. This explains why queries for non-existent values are faster.

We believe the reason for the large difference in query run time between SQL-Server and PostgreSQL is because SQL Server runs the RLS filter more efficiently. The RLS policy filter (Figure 1)



(a) Distribution (KDE plot)

Query Type	Avg. (μs)	Std. Dev	Logical Reads	
			Patients	Doctors
Non-Existent	386.01	37.23	4	0
Authorized	412.80	42.74	6	4
Unauthorized	404.49	41.04	6	4

(b) Average, standard deviation, and IO metrics.

Fig. 8. SQL Server query latency of existence privacy breaching query, by queried value type.

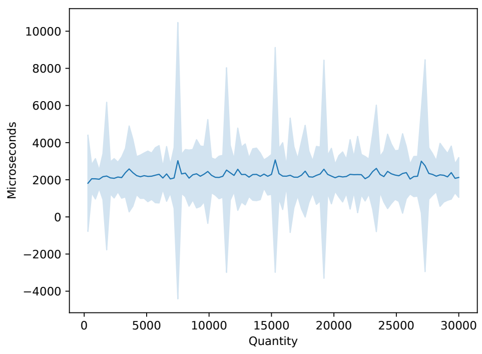
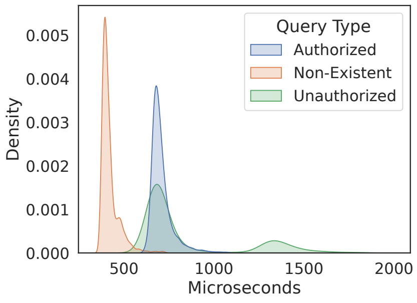


Fig. 9. SQL Server query latency distribution of cardinality privacy breaching query, as a function of the cardinality of Age value in the patients table. The line shows the average time, encompassed by the standard deviation.

has two parts: searching for the querying user in the `doctors` table and then comparing the user's *site_id* to the row's *site_id*. Since searching for the user does not depend on data from the *patients* table, SQL Server does it in parallel to running the query, whereas PostgreSQL executes the RLS policy filter as a black box only after the query filter finishes.

Cardinality privacy. In contrast to PostgreSQL, when SQL Server optimizes our cardinality privacy breaching query, it decides to evaluate the RLS filter before the user-supplied filter. As a result, the query always takes 2–4 milliseconds to execute, regardless of the cardinality of the queried Age in the table (Figure 9), which blocks the attack.

4.3.3 Attacking Complex Policies. Here, we show that our attack becomes easier to mount as the RLS policy becomes more complex and thus its execution time increases. To demonstrate this effect, we define a policy that leverages Active Directory (AD) permission services. AD is a popular directory-based service for identity/permission management in the Microsoft Windows ecosystem. SQL Server offers methods to query AD services. These are often used to define an RLS policy based on organization-level permissions defined in AD, rather than DBMS-specific permissions and dedicated tables.



(a) Distribution (KDE plot)

Query Type	Avg (μs)	Std. Dev
Non-Existent	420.65	50.02
Authorized	707.78	60.18
Unauthorized	841.3	285.74

(b) Average and standard deviation.

Fig. 10. SQL Server with AD-based policy: Existence privacy breaching query latency, by queried value type.

We evaluate our attack when the AWS SQL Server service is configured with such an AD-based policy, in which the policy filter uses the SQL Server *IS_MEMBER*() [26] method to check if the querying user belongs to the appropriate AD group. Figure 10 shows the running times of existence privacy breaching query on AD policy. The separation between queries for values that do and do not exist is significantly better than in prior experiments, as the policy filter is slower due to making AD queries.

5 MITIGATION APPROACHES

This section discusses the challenges of blocking RLS side-channel information leakage and approaches for mitigating the side-channels.

5.1 Data-Oblivious Queries

The standard approach for eliminating side-channels is via data-oblivious (or constant-time) techniques [7, 11, 14, 42, 43]. The basic idea is to perform a computation over secret information whose execution time is independent of the input secret data. This approach comes with a performance cost, because every computation runs in the same time as the worst-case computation. In this section, we explore the viability of a data-oblivious solution to RLS side-channels. We first propose a solution that provides privacy for *unique keys*, e.g., primary table keys. We also outline the challenges of implementing data-oblivious solutions for non-unique keys (§ 5.2).

5.1.1 Unique key solution. Here, we consider a unique key scenario, e.g., as with a keys that serve as a table's primary keys. In this case, all our privacy notions "collapse" to existence privacy: any key can either exist or not in the table, but cannot appear in more than one row if it exists. Maintaining existence privacy in this case therefore reduces to the problem of "hiding" the difference between a "miss" query targeting a non-existent key and a successful query targeting a key that exists in the table.

Recall that an existence privacy breach assumes (1) a non-measurable query delta, i.e., that the time difference between point queries for different queries is indistinguishable; and (2) a measurable policy delta, i.e., that the attacker can compare query execution times and distinguish whether the RLS filter was evaluated at least once in one query but not the other (§ 4.2). Assumption (1) holds for unique keys, because (excluding RLS costs) both types of queries perform one index search. It follows that making query execution data-oblivious requires addressing assumption (2), as assumption (1) implies that point queries *without RLS* are data-oblivious.

To maintain existence privacy, we need to "hide" the time difference between a query for a non-existent key and a query that returns a row. This time difference is due solely to the evaluation of the RLS policy on each row returned by the user query, as evidenced in Algorithm 2. The listing shows pseudo code of PostgreSQL's implementation of an indexed point query, which iterates over the set of returned rows and evaluates the RLS policy filter on each row.

Algorithm 2 Original behavior for index scan with RLS

```
1: procedure ExecuteQuery($Predicates$)
2:     $item, isFound = searchIndex(Predicates)$*
3:     if $isFound \neq true$ then
4:         return $null$
5:     end if
6:     $result = ExecuteRLSPolicy(item)$
7:     return $result$
8: end procedure
```

*The *searchIndex* function returns the last item that was traversed in the index scan even if no match value was found, and a Boolean value to signal if a match was found.

Algorithm 3 Existence Privacy for index scan with RLS

```
1: procedure SafeExecuteQuery($Predicates$)
2:     $item, isFound = searchIndex(Predicates)$
3:     $invalidQuery = !isFound$
4:     $result = ExecuteRLSPolicy(item)$
5:     if $invalidQuery = true$ then
6:         return $null$
7:     else
8:         return $result$
9:     end if
10: end procedure
```

Our proposed solution is to make any point query on an RLS-protected table execute exactly the same code path, regardless of whether the set of rows is empty or not—that is, we make *policy_delta* (§ 4.2) equal to zero. Algorithm 3 shows the pseudo code of our implementation in PostgreSQL. We add a flag that marks the query as invalid if no match was found in the index scan. However, we rely on the semantics of the index, which returns the row whose key is the predecessor of the searched key. This allows us to continue and evaluate the RLS policy filter on *some* row in the table—the row containing said predecessor. Finally, we drop the result if the query was marked invalid. Overall, a query executes exactly the same code path whether or not the searched key is found, and thus executes in a similar time.

Implementation. We implement our solution in PostgresSQL. Our implementation modifies the B^+-tree index implementation, which is the default index in PostgreSQL and fits most settings [31], and the query execution logic. We modify the index to return a value even for "miss" searches, while indicating to the query logic that this query should later be dropped. We modify the query logic to evaluate the RLS policy in these cases and subsequently drop the query.

Security evaluation. To evaluate the resistance of our implementation to existence privacy leakage, we re-run our proof-of-concept RLS side-channel demonstration (§ 4.3.1). Figure 11 shows the results. With our solution implemented in PostgreSQL, the distribution of query run times of the different queries becomes essentially identical, thereby defeating the attack.

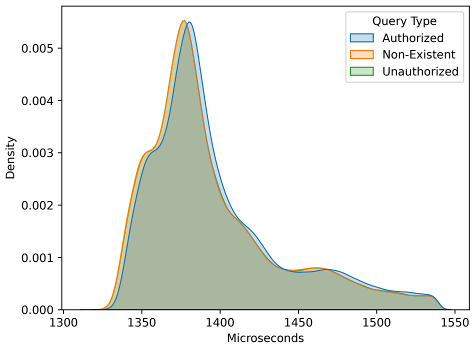


Fig. 11. Query latency distribution (KDE plot) of non-existent, authorized, and unauthorized queries.

One potential issue with these results is that this is a relatively lightweight workload. To alleviate this concern, we develop a benchmark for RLS side-channel protection (§ 6) and our evaluation finds similar results even with this heavier workload (§ 7).

Performance implications. Intuitively, our solution only adversely affects "miss" queries, which are slowed down by the amount of time the RLS policy filter takes to run. There are therefore two workload factors that determine the performance impact of our solution: the cost of the RLS filter and the fraction of queries that target non-existent keys ("miss"). In particular, *our solution imposes negligible overhead on any workload without "miss" queries"*. § 7 shows a detailed breakdown of how RLS filter time and miss query ratio impact performance with our solution.

5.2 Non-Unique Keys Data-Obliviousness Challenges

The main challenge in extending the data-oblivious solution approach to non-unique keys is that we risk significantly degrading performance, because we cannot bound the policy time delta between the query types. With non-unique keys, the "secret" becomes the cardinality of some key in the table, so a data-oblivious solution needs to make the query time over a value with high cardinality indistinguishable from that over a query with low cardinality—which means potentially significantly slowing down queries over low-cardinality keys, because we cannot magically decrease the execution time of queries of high-cardinality keys.

The above issue seems fundamental. If it is possible to bound the number of duplicate keys, we can implement data-oblivious queries in a manner similar to the unique key solution, whose latency will increase with the duplicate key bound. When no such bound can be guaranteed, we believe that in domains with high privacy requirements, *RLS should be restricted to unique keys*, where privacy can be efficiently achieved, as described in the previous section.

5.3 Discussion: Alternative Approaches

5.3.1 Applying Policy First. A straightforward approach for blocking RLS information leakage is by always applying the policy filter first, so that the user-supplied query filter never executes on rows the user is unauthorized to view. Unfortunately, such a solution would have unacceptable performance overheads, as detailed next.

Dictating that the policy filter executes first (or, equivalently, declaring all query operators not "leakproof") limits the query optimizer's options for constructing the most efficient query plan. Typically, an RLS policy cannot efficiently extract the rows that pass the filter for the querying user (e.g., via an index). As a result, the policy filter must be applied via a sequential scan. But this can dramatically increase query times. For instance, on a table with 20 M rows, a sequential scan that executes a 200 microsecond policy filter on each row will require more than an hour to complete. In fact, PostgreSQL introduces the limited set of "leakproof" operators to mitigate the performance problems caused by applying policy filters first [4].

Another option is defining the policy on an indexed column(s) of the protected table, but this means that all data required to verify the policy have to be in the protected table, which does not allow indirect access to security-related information. This approach imposes significant storage costs and management burden, because each row must contain access permission cell(s) and every table record must be updated whenever some access permission changes.

5.3.2 Volume Hiding. Techniques for secure cloud storage may seem applicable to our problem, as their goal is to prevent information leakage. We argue, however, that these techniques are not appropriate to block RLS information leakage due to scope and performance reasons.

In secure cloud storage, the goal is to prevent the system hosting the RDBMS from learning information about the data. The attacker is thus more powerful than our attacker—a regular user interacting with RDBMS using queries—and the protection techniques are accordingly heavyweight, e.g., requiring encryption of the data and data oblivious storage access patterns. While a full survey of secure storage techniques is outside the scope of this paper, we exemplify the issues by going through the exercise of evaluating the particular technique of *volume hiding* [16, 19, 28] as a solution to our problem.

Volume-hiding techniques protect against leakage of the number of values associated with a key (its *volume*) from structured encryption (STE) databases. Volume hiding thus requires an encrypted database, which is already overkill for many RLS applications. Our target workloads are dynamic (with table updates), but volume-hiding techniques must rebuild data structures after dynamic updates. Last but not least, volume hiding focuses on leakage from the storage representation (as the attacker can observe the storage content) and not on leakage from query execution time. In fact, designs are explicitly allowed to leak the so-called "query equality pattern," i.e., whether two queries are to the same key or not—which violates our existence privacy notion (§ 4.1).

5.3.3 Other Approaches.

Timing obfuscation. In general, timing side-channels can be blocked by adding fixed or dynamic delays to the timed operation [18, 33, 35]. This technique is expensive and does not completely eliminate the timing channel. In addition, timing obfuscation in our context requires constantly evaluating query run times—as load changes—to determine what a "typical" query time looks like, in order to artificially increase the execution time of all queries to look like a "typical" query.

Materialized views. A per-user materialized view reflecting the user's access permissions achieves RLS's security goal, but it has to be recalculated on table changes. In contrast, RLS efficiently supports dynamic updates.

6 RLS BENCHMARK: MULTI-TENANT YCSB

This section introduces *multi-tenant YCSB*, an adaption of the popular YCSB [10] to the context of evaluating the security and performance of RLS side-channel mitigations.

Motivation. There are several reasons why RLS side-channel mitigation requires a new benchmark for evaluation, and cannot be done using existing benchmarks such as TPC-C, TPC-E [5], or standard YCSB:

- *Multi-tenancy support.* Existing schemas and workloads target a single user (tenant). But without multi-tenancy, there is no use case for RLS. We therefore need schema and data that admit multiple tenants and have meaningful security restrictions on the data viewable by each tenant, as well as workloads that simulate queries but different tenants. In particular, we need to able to define an RLS policy in the benchmark.
- *Unauthorized queries.* In addition to supporting multi-tenancy, workloads in an RLS benchmark need to perform unauthorized queries, which are disallowed by the RLS policy.
- *Miss queries.* Query workloads in existing benchmarks, such as YCSB and TPC-C, do not contain "miss" queries, which *fail* to find any data in the queried tables. As discussed in § 5, however, miss queries are likely to suffer the most negative performance impact from an RLS side-channel mitigation. A fair evaluation therefore requires a benchmark that contains miss queries.

6.1 Benchmark Description

The scenario our benchmark simulates is that of a table shared between tenants, each of which can have multiple users. To prevent users from viewing data of other tenants, the table is protected by an RLS policy that limits users of a tenant to viewing only records that belong to the tenant.

Our benchmark is an adaption of YCSB. We choose to adapt YCSB for our purposes because it is both widely accepted and used, while having a simple schema that is straightforward to adapt. In addition, looking forward to other uses of our benchmark, a YCSB-based workload is more suitable for additional non-relational uses of RLS that exist in practice—for example, in NoSQL databases like Google BigQuery and in Open Policy Agent [2] (a service providing object-level security). YCSB defines a single table, called the *usertable*. YCSB workloads perform different ratios of SELECT, INSERT, and UPDATE operations. The SELECT operations are performed against a column named *YCSB_KEY*, which is indexed by default and is the table's primary key. To create our benchmark, we modify YCSB's table structure as well as the workload phases that populate and execute the operation workload.

Figure 12 describes the changes to YCSB's table structure. First, we extend the usertable schema with a tenant id (string) column. We then add an RLS policy table, called *acls*, which is a mapping between a database user [29] to its tenant id. Finally, we create an RLS policy that limits a database user to viewing only rows in the usertable that match its tenant id.

We next adapt the benchmark itself. YCSB has a *load* phase, which populates the usertable, and a *run* phase, which executes a specified operation workload against the usertable. To support multi-tenancy, we add a *load-tenant* phase, which populates the RLS policy table with users and tenant ids, creates the users in the database, and adjust their permissions to be able to read from the usertable restricted by the RLS policy. We modify the load phase to populate each row with a tenant id that exists in the RLS policy table. The number of tenants and the number of users per tenant are benchmark parameters. Tenants ids are generated randomly, and subsequently, we generate X users per tenant, as specified in the benchmark's configuration.

Finally, we add a new workload that contains miss queries and unauthorized queries. We also need to modify the YCSB client code, which assumes a single users, to support database connections by multiple users.

7 MITIGATION EVALUATION

We use multi-tenant YCSB to evaluate two aspects of our RLS side-channel mitigation. First, we verify the mitigation's security (i.e., that no information is leaked) not only "in vacuum" but under load, in the presence of normal traffic by concurrent clients (§ 7.2). Second, we evaluate the performance impact of the mitigation, by investigating how policy run time, table size, and ratio of miss queries impact workload throughput (§ 7.3).

```
  CREATE TABLE usertable (
    YCSB_KEY VARCHAR(255) PRIMARY KEY,
    FIELD0 TEXT, FIELD1 TEXT,
    FIELD2 TEXT, FIELD3 TEXT,
    FIELD4 TEXT, FIELD5 TEXT,
    FIELD6 TEXT, FIELD7 TEXT,
    FIELD8 TEXT, FIELD9 TEXT,
    tenant_id VARCHAR(36)
  );

/*
* tenant_id - 36 chars long to accommodate a UUID
* user_name - 50 chars long to accommodate
* the maximum length of a user name in PostgreSQL.
*/
  CREATE TABLE acls (
    pkey serial PRIMARY KEY,
    user_name VARCHAR(50),
    tenant_id VARCHAR(36),
    CONSTRAINT user_name_unique UNIQUE (user_name)
  );

  CREATE FUNCTION tenant_read_policy(varchar(36) tenant_id, text curr_user)
  RETURNS bool
  AS $$
  BEGIN
    RETURN ((SELECT tenant_id FROM acls WHERE user_name = curr_user) = tenant_id);
  END;
  $$
  LANGUAGE plpgsql;

  CREATE POLICY tenant_read on usertable
  FOR SELECT
  USING(tenant_read_policy(tenant_id, current_user));

  ALTER TABLE usertable ENABLE ROW LEVEL SECURITY;
```

Fig. 12. Multi-tenant YCSB setup.

7.1 Experimental Setup

We run a multi-YCSB workload that performs 5 million SELECT queries. Of these queries, 1% are unauthorized queries and the rest are authorized queries, a user-configurable fraction of which are miss queries. We run 63 different *configurations* of the benchmark, which explore the space of table sizes, RLS policy overhead, and ratio of miss queries in the workload, as described below. We compare PostgreSQL 13.2 with our RLS side-channel mitigation to the baseline PostgreSQL 13. (We cannot implement our mitigation in SQL Server because its source code is not publicly available.)

Configurations. We evaluate 63 configurations, covering the space of the following parameters:

- *Table size:* 200 K, 2 M, and 20 M rows.
- *RLS policy overhead:* We use the policy table size (number of rows) as a proxy for policy overhead. Because the table is not indexed, the policy filter's evaluation time is linear in the policy table's size. We use policy table sizes of 50, 1000, and 10 K rows.
- *Miss query ratio:* 1%, 5%, 10%, 20%, 30%, 40%, and 50%.

Hardware. We use a server with a 32-core AMD EPYC 7502 processor. The server is equipped with 256 GB of DDR4 DRAM and a 280 GB SSD. The server runs Ubuntu 18.04.

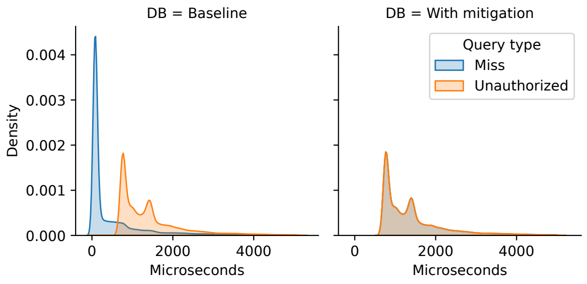


Fig. 13. Query latency distribution (KDE plot) for miss and unauthorized queries on baseline PostgreSQL 13.2 and PostgreSQL with our RLS side-channel mitigation. This comparison uses a table with 20 M rows, an RLS policy table with 10 K rows, and a fraction of 1% of miss queries and unauthorized queries (each).

7.2 Security Under Load

Here, we evaluate the security of our mitigation with respect to existence privacy. To this end, we compare average query latency (in microseconds) of unauthorized queries, miss queries, and authorized successful queries. In every configuration, we observe essentially indistinguishable latencies between the unauthorized queries and the authorized miss queries, indicating that an attacker cannot tell whether an unauthorized query succeeds or misses. Figure 13 shows an example of the results from a specific configuration. With baseline PostgreSQL, there is a statistically significant difference between the latency distributions of "miss" queries and unauthorized queries (to keys that exist, but the user is not allowed to view). In contrast, with our mitigation implemented, the latency distribution of both query types is identical. The results in the other configurations are the same.

Overall, we find that our mitigation provides existence privacy even under the system load simulated by multi-tenant YCSB. The latency difference between miss queries and unauthorized queries is less than 100 nanoseconds. We do see authorized queries taking longer than miss queries and unauthorized queries, but this is caused by the time it takes an authorized query to read the query's results (i.e., a set of rows), because only authorized successful queries produce non-empty results.

7.3 Overheads of Existence Privacy

Here, we quantify the performance impact of our mitigation, and analyze its causes. To this end, Figure 14 shows the overhead (relative throughput degradation) impose by enabling our mitigation in each tested configuration. In the figure, the X axis varies the fraction of miss queries, and for each fraction, we show different lines for each pair of table size and RLS table size.

We see that the factors dictating the overhead are the RLS policy table size and the fraction of miss queries, but table size does not affect the relative overhead. The overhead depends on the ratio of miss queries, because only miss queries are slowed down by our mitigation. Specifically, when miss queries are rare (e.g., below 10%) we observe little overhead (less than 10% in the worst case). For each fraction of miss queries, the magnitude of the overhead observed depends only on the RLS policy table size. This is because the policy table size determines the policy filter's run time, which is an extra cost imposed on the query's processing of each row in the table.

We conclude that production workloads with low miss query ratios experience little overhead, regardless of the RLS filter's execution time. But if the RLS policy execution time is not negligible, then workloads with a large fraction of miss queries may observe significant slowdowns.

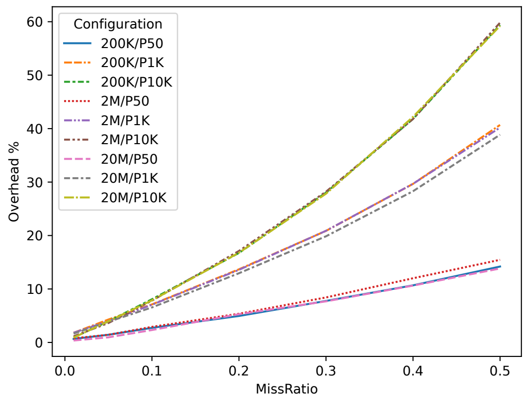


Fig. 14. Throughput overhead of existence privacy on different configurations. Lines labeled X/Y refer to a configuration with a table size and RLS policy size of X and Y rows, respectively.

8 RELATED WORK

Data store side-channels. Side-channel attacks and mitigations have mostly been explored in cryptographic software (e.g., [20]) in the context of *microarchitectural* leakage, e.g., over a last-level cache shared between victim and attacker program co-located on the same machine [41]. There is less research on side-channel attacks against data stores. Futoransky et al. [15] use a timing side-channel attack on insertions into a MySQL database to extract private keys. The attack succeeds when a B$^+$-tree index is used and the attacker is able to insert records with chosen data that is used as the search key of one of the table's indexes. They observe that insertions take longer when a B$^+$-tree node splits and use a divide-and-conquer approach to learn the private keys. Wang et al. [39] show a practical side-channel attack on a multi-user search system, such as Elasticsearch. The attack exploits TF-IDF scores (a popular statistical score that is meant to reflect how important a word is to a document in a corpus and is used to rank search results) to leak information about another user's documents.

Work on oblivious algorithms presents query optimizers aimed at preserving data privacy and mitigating side-channels. Most of these works [14, 43] target side-channels caused by memory and network access patterns while ignoring timing side-channels. Xu et al. [40] combine oblivious algorithms with differential security to provide security against a variety of side-channels, including timing attacks, and allow the user to decide the trade-off between privacy and performance. Their solution is designed to replace existing query planners, not to enhance them, and has limited functionality compared to the full SQL specification.

RLS side-channels. Developers of RLS describe and mitigate non-timing side-channels in RLS [25, 30]. They describe how carefully crafted queries can leak information by throwing exceptions such as divide-by-zero or by performing insert queries that violate a table constraint such as inserting duplicate keys to a column with a unique constraint. These attacks can be mitigated by hardening the database engine's error handling [17] to silence exceptions and limiting user write permissions.

9 DISCUSSION AND FUTURE WORK

We explore time side-channels introduced by RLS and their mitigation. We show that RLS time side-channels are exploitable in practice, and can be used to violate existence and cardinality privacy. We propose an oblivious query mechanism to block these side-channels on unique keys with minimal overhead, and identify protection of non-unique keys as an open problem. Accordingly, security conscious systems may wish to limit RLS usage to unique keys. We further introduce a new multi-tenant benchmark based on YCSB, which can be used to evaluate security solutions for multi-tenant settings.

This work considers only time side-channel created by SELECT SQL queries. In future work, it is important to investigate whether other SQL operations, such as UPDATE, INSERT and DELETE, can create side-channels via RLS. These operations have more complex interactions with the system, as their ability to modify tables requires additional security measures, such as block predicates [24, 30].

More generally, RLS is but one example of object-level security mechanisms, which are gaining traction in various data stores. There are already examples outside of relational databases, such as in MongoDB [1] and generic application layer solutions [2], which provide object-level security that is similar to RLS. It is likely that our RLS attacks and defenses can be generalized to these other object-level security implementations, due to the similar way they are implemented. Side-channels in RLS thus may be only the tip of the iceberg of information leakage created by object-level security mechanisms.

ACKNOWLEDGEMENTS

We thank ChatGPT for summarizing the paper with the following free-form haiku:

Many tenants, one DB
Row-level security, but wait
Information leaks

REFERENCES

[1] [n.d.]. MongoDB Atlas Overview - MongoDB Realm. https://docs.mongodb.com/realm/mongodb/
[2] [n.d.]. Open Policy Agent. https://www.openpolicyagent.org/
[3] [n.d.]. OpenEMR. https://www.open-emr.org/
[4] [n.d.]. Security-level constraints on qual clauses. https://raw.githubusercontent.com/postgres/postgres/REL_13_STABLE/src/backend/optimizer/README
[5] [n.d.]. TPC. http://tpc.org/
[6] 2012. OpenEMR - Patient and Doctor Record Sharing. https://sourceforge.net/p/openemr/discussion/202506/thread/8d94ff0a/
[7] Jose Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. 2016. Verifying Constant-Time Implementations. In *USENIX Security '16*.
[8] AWS. [n.d.]. Amazon Relational Database Service. https://aws.amazon.com/rds/.
[9] Microsoft Azure. 2022. Multi-tenant SaaS database tenancy patterns. https://docs.microsoft.com/en-us/azure/azure-sql/database/saas-tenancy-app-design-patterns?view=azuresql.
[10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *SoCC '10*.
[11] Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. 2009. Practical Mitigations for Timing-Based Side-Channel Attacks on Modern x86 Processors. In *IEEE S&P'09*.
[12] Aveek Das. 2020. Introduction to Row-Level Security in SQL Server. https://www.sqlshack.com/introduction-to-row-level-security-in-sql-server/
[13] Elastic. [n.d.]. Document level security. https://www.elastic.co/guide/en/elasticsearch/reference/current/document-level-security.html.
[14] Saba Eskandarian and Matei Zaharia. 2019. ObliDB: Oblivious Query Processing for Secure Databases. *Proceedings of the VLDB Endowment* 13, 2 (2019), 169–183.

[15] Ariel Futoransky, Damián Saura, and Ariel Waissbein. 2007. The ND2DB Attack: Database Content Extraction Using Timing Attacks on the Indexing Algorithms. In *WOOT '07*.

[16] Marilyn George, Seny Kamara, and Tarik Moataz. 2021. Structured Encryption and Dynamic Leakage Suppression. In *Eurocrypt '21*.

[17] Rajendra Gupta. 2019. Methods to avoid the SQL divide by zero error. https://www.sqlshack.com/methods-to-avoid-sql-divide-by-zero-error/

[18] Wei-Ming Hu. 1992. Reducing Timing Channels with Fuzzy Time. *Journal of Computer Security* 1, 3-4 (1992), 233–254.

[19] Seny Kamara and Tarik Moataz. 2019. Computationally Volume-Hiding Structured Encryption. In *Eurocrypt '19*.

[20] Ramesh Karri, Kaijie Wu, Piyush Mishra, and Yongkook Kim. 2002. Concurrent error detection schemes for fault-based side-channel cryptanalysis of symmetric block ciphers. *IEEE Transactions on CADICS* 21, 12 (2002), 1509–1517.

[21] Rouven Krebs, Alexander Wert, and Samuel Kounev. 2013. Multi-tenancy Performance Benchmark for Web Application Platforms. In *ICWE '13*.

[22] Shanhong Liu. 2021. Statista | Ranking of the most popular relational database management systems worldwide, as of January 2021. https://www.statista.com/statistics/1131568/worldwide-popularity-ranking-relational-database-management-systems/

[23] Microsoft. 2016. SQL Server | Microsoft Docs - SET STATISTICS IO (Transact-SQL). https://docs.microsoft.com/en-us/sql/t-sql/statements/set-statistics-io-transact-sql?view=sql-server-ver15

[24] Microsoft. 2020. SQL Server | Microsoft Docs - Row-Level Security. https://docs.microsoft.com/en-us/sql/relational-databases/security/row-level-security

[25] Microsoft. 2020. SQL Server | Microsoft Docs - Row-Level Security - Security Note: Side-Channel Attacks. https://docs.microsoft.com/en-us/sql/relational-databases/security/row-level-security?view=sql-server-ver15#SecNote

[26] Microsoft. 2022. SQL Server | Microsoft Docs - IS_MEMBER (Transact-SQL). https://learn.microsoft.com/en-us/sql/t-sql/functions/is-member-transact-sql?view=sql-server-ver16.

[27] Rajesh Nadipalli. 2017. Using Row-Level Security (RLS) to Restrict Access to a Dataset. https://docs.aws.amazon.com/quicksight/latest/user/restrict-access-to-a-data-set-using-row-level-security.html

[28] Sarvar Patel, Giuseppe Persiano, Kevin Yeo, and Moti Yung. 2019. Mitigating Leakage in Secure Cloud-Hosted Data Structures: Volume-Hiding for Multi-Maps via Hashing. In *CCS '19*.

[29] PostgreSQL. 2021. PostgreSQL Documentation | Create role. https://www.postgresql.org/docs/13/sql-createrole.html

[30] PostgreSQL. 2021. PostgreSQL Documentation | Row Security Policies. https://www.postgresql.org/docs/13/ddl-rowsecurity.html

[31] PostgreSQL. 2021. PostgreSQL Documentation | Row Security Policies. https://www.postgresql.org/docs/13/indexes-types.html

[32] PostgreSQL. 2021. PostgreSQL Documentation | Rules and Privileges. https://www.postgresql.org/docs/13/rules-privileges.html

[33] Ashay Rane, Calvin Lin, and Mohit Tiwari. 2015. Raccoon: Closing digital side-channels through obfuscated execution. In *USENIX Security '15*.

[34] David Reed. 2017. New SQL Server 2016 Features for Business Data Protection – Part 3: Row Level Security. https://www.ktlsolutions.com/business-data-protection-row-level-security/

[35] Sebastian Schinzel. 2011. An Efficient Mitigation Method for Timing Side Channels on the Web. In *COSADE '11*.

[36] Hans-Jürgen Schönig. 2019. Using "row level security" to make large companies more secure. https://www.cybertec-postgresql.com/en/using-row-level-security-to-make-large-companies-more-secure/

[37] Anubhav Sharma and Ujwal Bukka. 2021. Building a Multi-Tenant SaaS Solution Using AWS Serverless Services. https://aws.amazon.com/blogs/apn/building-a-multi-tenant-saas-solution-using-aws-serverless-services/.

[38] Tom Van Goethem, Christina Pöpper, Wouter Joosen, and Mathy Vanhoef. 2020. Timeless Timing Attacks: Exploiting Concurrency to Leak Secrets over Remote Connections. In *USENIX Security '20*.

[39] Liang Wang, Paul Grubbs, Jiahui Lu, Vincent Bindschaedler, David Cash, and Thomas Ristenpart. 2017. Side-Channel Attacks on Shared Search Indexes. In *IEEE S&P '17*.

[40] Min Xu, Antonis Papadimitriou, Ariel Feldman, and Andreas Haeberlen. 2018. Using Differential Privacy to Efficiently Mitigate Side Channels in Distributed Analytics. In *EuroSec '18*.

[41] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security '14*.

[42] Jiyong Yu, Lucas Hsiung, Mohamad El Hajj, and Christopher W. Fletcher. 2019. Data Oblivious ISA Extensions for Side Channel-Resistant and High Performance Computing. In *NDSS '19*.

[43] Wenting Zheng, Ankur Dave, Jethro G. Beekman, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. 2017. Opaque: An Oblivious and Encrypted Distributed Analytics Platform. In *NSDI '17*.

Received July 2022; revised October 2022; accepted November 2022