On Vulnerability of Access Control Restrictions to Timing Attacks in a Database Management System

Alexander Rasin
DePaul University
Chicago, IL, USA
arasin@cdm.depaul.edu

James Herbick
DePaul University
Chicago, IL, USA
jherbick@depaul.edu

Ben Lenard
DePaul University
Argonne National Laboratory
Chicago, IL, USA
blenard@anl.gov

Nick Scope
DePaul University
Chicago, IL, USA
nscope52884@gmail.com

James Wagner
University of New Orleans
New Orleans, LA, USA
jwagner4@uno.edu

Abstract

Side-channel attacks leverage implementation of algorithms to bypass security and leak restricted data. A timing attack observes differences in runtime in response to varying inputs to learn restricted information. Most prior work has focused on applying timing attacks to cryptoanalysis algorithms; other approaches sought to learn about database content by measuring the time of an operation (e.g., index update or query caching). Our goal is to evaluate the practical risks of leveraging a non-privileged user account to learn about data hidden from the user account by access control.

As with other side-channel attacks, this attack exploits the inherent nature of how queries are executed in a database system. Internally, the database engine processes the entire database table, even if the user only has access to some of the rows. We present a preliminary investigation of what a regular user can learn about "hidden" data by observing the execution time of their queries over an indexed column in a table. We perform our experiments in a cache-control environment (i.e., clearing database cache between runs) to measure an upper bound for data leakage and privacy risks. Our experiments show that, in a real system, it is difficult to reliably learn about restricted data due to natural operating system (OS) runtime fluctuations and OS-level caching. However, when the access control mechanism itself is relatively costly, a user can not only learn about hidden data but they may closely approximate the number of rows hidden by the access control mechanism.

CCS Concepts

• **Information systems** → *Database query processing*; *Data access methods*; • **Security and privacy** → **Cryptography**; **Database and storage security**.

Keywords

Side-Channel Attack, Timing Attack, Data Privacy

SSDBM 2024, July 10–12, 2024, Rennes, France

ACM ISBN 979-8-4007-1020-9/24/07
https://doi.org/10.1145/3676288.3676306

ACM Reference Format:
Alexander Rasin, James Herbick, Ben Lenard, Nick Scope, and James Wagner. 2024. On Vulnerability of Access Control Restrictions to Timing Attacks in a Database Management System. In *36th International Conference on Scientific and Statistical Database Management (SSDBM 2024), July 10–12, 2024, Rennes, France.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3676288.3676306

1 Introduction

Database management systems (DBMS) serve as the main data repository for most organizations because they support a full complement of data management capabilities with a variety of security mechanisms. These security capabilities include access control, different levels of encryption, data masking, and audit logs (e.g., to monitor user access). Although DBMSes incorporate robust security tools, they are also a common target of security attacks.

Much of the research related to side-channel attacks has been to bypass and reverse engineer encryption [13]. More recently, timing attacks that attempt to learn about the data directly have been considered as well [10]. We believe that such attacks represent a particular risk to overall security and, specifically, to compliance in organizations. Compliance with data access rules is especially important for organizations that are in heavily regulated industries such as healthcare and financial industries. Although the timing attack explored in this paper does not directly bypass security, it can nevertheless expose sensitive data to users who were explicitly prevented from seeing it. The main contribution of this paper is to investigate the practical risks of privacy compromise that can be perpetrated by local users without elevated privileges.

We investigate leveraging of a database user account without elevated privileges to learn about data hidden behind access control. Although we make some simplifying assumptions such as the existence of an underlying indexed column (e.g., *Salary*) and clear database cache, this attack is simple to execute in practice. Intuitively, our timing attack seeks to detect the extra cost imposed by filtering of access-restricted rows. The database engine performs additional work to eliminate rows which match the user's query predicates but are excluded from results by access control because the user does not have access to these rows. Although our results so far show that the timing attack is the most reliable when the access control mechanism itself is expensive (e.g., when it involves

lookup tables), this threat deserves further investigation because of the inherent difficulty of defending against side-channel attacks.

2 Related Work

Randolph and Diehl [15] present a twenty-year summary of research connecting power consumption and what it can tell about cryptographic algorithm behavior. Such attacks may include observing power use to extrapolate the difference between keys being used by cryptographic computations [13]; monitoring the heat signature to exploit differences in thermal output to learn data used by cryptographic algorithms [12]; measuring the time required for the execution of the cryptographic algorithm in order to reverse-engineer keys, e.g., discovering private keys from OpenSSL by attacking a web server on a LAN [7]. Other attacks include measurement of electromagnetic emanations, optical output, and acoustics output. In sum, any measurable output produced by the system that varies depending on the data it is computing can potentially reveal secret information if observed and measured.

In the context of a database management system, side-channel attack can also achieve a more specialized discovery of data leaks. For example, attacks against data anonymization mechanism bypass query anonymization that obfuscates source data in database queries. Boenisch et al. [6] demonstrate a timing attack that discovers underlying data despite a differential privacy framework through a timing attack (measuring query cost and impact of exceptions caused by a query to learn about anonymized data). Futoransky et al. [11] presented an attack technique that uses a timing attack to reverse engineer the data in a B-Tree. By observing the cost of `INSERT`s and `UPDATE`s, they could observe the runtime cost increased caused by B-Tree splits and merges, thus learning about the data contained within a B-Tree. Dar et al. performed a similar type of an attack by timing repeated query execution [10]. However, their approach relies on measuring query execution times in nanoseconds, learning about underlying data based on sub-millisecond runtime differences. Although similar to our analysis, such attack is difficult to perform in practice (e.g., we found that differences of up to 3ms in runtime did not present a statistically significant difference on a database server). Other timing attacks have sought to detect runtime differences caused by cache hits and misses to learn what data has been queried in the database [16].

3 Methodology

3.1 Assumptions and Threat Model

We consider timing attacks perpetuated by a non-malicious database user with regular access to a database table. We do not assume any privileges beyond the ability to execute a SQL query and to time the execution of that query. For the purposes of the analysis in Section 4, we considered query plans when choosing our queries. However, the attacker could easily approximate that information with additional server-specific analysis (e.g., by creating their own controlled table on this or similar server and timing queries). We further assume that the user is able to execute queries locally and that network transfer does not factor into query execution cost.

3.2 Access control

`GRANT` capabilities (including `SELECT`, `UPDATE`, `INSERT`, or `DELETE`) is the most common mechanism supported by relational databases. Another typical approach for restricting a user's access to underlying tables is to deploy views. A database administrator would grant access to a view instead of the underlying table so that data restrictions on the data can be imposed with the view's `WHERE` clause [1].

Row Level Security (RLS) or Label Security has been available in Oracle starting from Oracle 8i. RLS applies predicates to the `WHERE` clause when a SQL statement is executed [2, 8]. RLS filters on the basis of rows and cannot change the columns returned in the result set. Similar to Oracle, Db2 Linux Unix Windows (LUW), has label-based security, where a special label column is appended to each table. Table access can be controlled by the policy matching the condition to the label for any row [4, 9]. Postgres also offers an RLS mechanism that applies a filtering `WHERE` clause to the queries based on applicable policies [14].

Oracle Virtual Private Database (VPD) technology allows for a policy (using PL/SQL) to be executed when fetching rows; the policy determines what predicates are added to the `WHERE` clause and can apply restrictions to the columns, so that the projection of data being returned is restricted [2, 3]. Oracle has reinvented VPD as Real Application Security (RAS) which builds upon VPD and adds additional features and ease of use. Consider a multi-tenant application, where many organizations share the same application but a guarantee of complete isolation is required. A VPD policy can be used to append organization's identification number to the `WHERE` clause to ensure that only that company's data is displayed for a particular query.

In sum, most access control approaches are ultimately applied to the query through a `WHERE` clause. Although there is an endless variety of mechanisms, in practice most come down to revising the query, introducing access control `WHERE`-restrictions. For a more controlled fine-grained restrictions tools such as Oracle Data Redaction [5] could be used. In general, the database engine does not have the ability to distinguish predicate restrictions introduced due to access control and the original query predicates.

3.3 Cost of Access Control

In our evaluation, we consider both a simple access control predicate (i.e., `Department = 'Health'`) and simulate a more expensive predicate (by introducing a 1ms overhead using *pg_sleep*).

Access control restriction implementation may be relatively costly in practice. For example, consider a customer relationship management system at JPMorgan Chase. A person in sales team for Treasury services can only see their clients; however, if their client is Ford, the condition does not map to a simple `WHERE Company = 'Ford'` because Ford has hundreds of sub-companies ranging from Ford Canada, to Mopar to Avis and Mazda. Similarly, a condition such as "manager can see data of their employees" may require multi-table joins to materialize that relationship for access control.

A second category where access control may be costly is multi-tenancy environment such as in super-computer setting. For example, in order for the user to query the computer information regarding their computing job, a number of checks is needed: 1) Determine the right project (group) that has the relevant job running

on the system, 2) Identify which of the hundreds or thousands of nodes has the job, 3) Differentiate between jobs sharing a node or running exclusively on that node. Making these determinations and translating them into access control requirements may require a non-trivial amount of time.

3.4 Dataset

We used publicly available San Francisco municipal employee salary data spanning the years 2011-2014, inclusive[1]. We further derived a *Department* column based on the employee job titles and bucketed data into different departments: e.g., Fire Department, Police Department, Health Department (and a catch-all "Other" deparment). The full table included 148,654 records in total; the table comprised 3,106 pages on disk.

3.5 Database Initialization

We loaded data into a PostgreSQL 16.0 database in a Unix-based environment. As salaries turned out to be correlated with their physical offset, we created *salary_data_shuffled* which randomized the data prior to loading. We created a user account which was restricted to only seeing data from the "Health" department. We implemented a Row Level Security (RLS) policy for our user which restricted access based on `Department = 'Health'`. Additionally, we simulated an "expensive" policy which used the same condition but also introduced 1ms sleep delay to the access control check.

Both tables are indexed on the *Id* column (primary key) by default. We created an additional index over the *TotalPay* column which we use in our experiments. We ran some simple queries to validate that the row-level restrictions were working correctly.

3.6 Metrics Capture

We activated the *pg_stat_statements* in the postgresql.conf configuration file, restarted the database server, and then implemented *pg_stat_statements* as an extension in the PostgreSQL database. We also turned on the 'timing' option for returning information at the time of executing our SQL queries. We did so with the *timing* on for SQL statements. In Python, we used the *%timeit magic* command and ran each SQL statement 30 times, to get a consistent timing. Although the times were similar, we ultimately used the times reported in database logs in our reported experiments.

4 Experiments

We compare the difference between query runtimes that access different numbers of rows on disk based on the user predicate while returning the same number of rows. As the queries return the same number of rows, the only significant difference between the queries is the number of rows that the query was forced to exclude from the results due to access control. In order to capture that difference, we consider queries that request ranges over the indexed column, *TotalPay*. A range query over *TotalPay* will use an index to identify all matching salaries. However, as our user is restricted to one department ('Health'), some of these rows may be excluded due to being in a different department. In order to exclude the rows, DBMS engine must scan the *salary_data_shuffled* table because the department column is not indexed. Our hypothesis is that when a user executes queries against an indexed column, they can learn about the presence of inaccessible data in requested ranges.

In our evaluation, we executed a set of 30 queries by the user with access limited to 'Health' department. All queries use the index on the *TotalPay* column. Between each run of an individual query, we ensure that the database cache has been cleared by restarting the PostgreSQL server. Note that a regular user lacks the privilege to restart the server. However, there are several strategies to prevent caching from affecting our timing attack. Perhaps the simplest approach is to rotate different query ranges in order to gather information about the column(s) with sensitive data. We enabled logging and capture query execution times in the PostgreSQL logs and used ElasticSearch to retrieve query timing.

We first evaluated `SELECT COUNT("TotalPay") AS CNT FROM salary_data WHERE "TotalPay" BETWEEN XXX AND YYY` across different salary ranges shown in Table 1 for Query Q1. Ranges were chosen to always return 5 rows to the user, even as the *TotalPay* range matched more rows. For each execution range, we report the number of rows falling within that salary range, the number of rows that were restricted (i.e., excluded) by access control, and the average runtime of 30 executions.

We compared the query execution times using the statistical t-test for two means, unequal variances, and independent samples. We use the first *TotalPay* range in Table 1 as the baseline, comparing the runtimes to each subsequent row. We consider a critical value of p=0.05 to determine statistical significance. Our null hypothesis is that the two means being compared are equal. The alternative hypothesis is that the query runtime mean being compared to the baseline is not equal. Our results indicate a statistically significant difference between the queries only for the $\$251,000 - \$275,000$ range. In other cases, even though the average runtime is slower, the null hypothesis could not be rejected.

Our next query attempted to increase the cost of evaluation by accessing the table twice. Specifically, we used a query that queried the same range a second time using a subquery: `SELECT COUNT(*) + (SELECT COUNT("Id") FROM salary_data WHERE "TotalPay" BETWEEN XXX AND YYY) AS DBL FROM salary_data WHERE "TotalPay" BETWEEN XXX AND YYY`. In Table 1 Query Q2, although the runtimes are slower compared to the baseline, the null hypothesis could not be rejected for any of the ranges.

Our third query evaluation uses the same query as Table 1 Query Q1, but emulates a more costly access control mechanism by introducing a 1ms sleep (using *pg_sleep* call) into the access control rules. As shown in Table 1 Query $Q1_{HA}$, a high-cost access control causes a drastic difference between query runtimes. We then used the runtimes to fit a linear regression equation to the data:

$$y = 2.2707x + 24.81 \quad (1)$$

where y represents query execution time in milliseconds, and x represents the number of restricted rows. We used these results to estimate the number of restricted rows based on arbitrarily chosen ranges of salaries. We executed queries over the *TotalPay* ranges shown in Table 2. As an example, plugging in an execution time into our estimation regression equation $220.017 = 2.2707x + 24.81$, estimates the number of restricted rows to be $x = 86$. The actual number of restricted rows in this case was 79.

[1] https://www.kaggle.com/datasets/kaggle/sf-salaries

TotalPay Range	Rows in range	Restricted rows	Returned rows	Query	Runtime (ms)	p-value
225,700-225,900	7	2	5	Q1	16.098	-
				Q2	14.636	-
				$Q1_{HA}$	28.178	-
152,500-152,780	84	79	5	Q1	17.548	0.361
				Q2	17.802	0.0512
				$Q1_{HA}$	204.095	3.8737E-78
251,000-275,000	136	131	5	Q1	21.875	0.004
				Q2	16.640	0.185
				$Q1_{HA}$	325.095	6.6642E-85
255,000-350,000	237	232	5	Q1	18.438	0.154
				Q2	16.201	0.252
				$Q1_{HA}$	550.059	1.3786E-82

Table 1: Query results for Q1: `SELECT * FROM salary_data WHERE "TotalPay" BETWEEN XXX AND YYY`
Q2: `SELECT COUNT(*) + (SELECT COUNT("Id") FROM salary_data ...) AS DBL FROM salary_data WHERE "TotalPay" BETWEEN XXX AND YYY`
$Q1_{HA}$: `SELECT * FROM salary_data WHERE "TotalPay" BETWEEN XXX AND YYY` **using high-cost access control**

TotalPay Range	Restricted rows (Actual)	Restricted rows (Estimate)	Runtime (ms)
255,000-255,300	0	-6	10.762
255,000-256,300	7	1	26.192
200,000-200,500	37	32	97.082
180,000-180,500	79	86	220.017
180,100-180,700	105	122	302.404

Table 2: Linear estimates of rows hidden by access control using Eq. 1.

5 Conclusion

Our experiments show that side-channel timing attack can allow a user to discover information about inaccessible data in the database. Although our results show that simple (i.e., low-overhead) access control predicate does not leak significant information, we believe that the user can craft a more costly query (e.g., by augmenting the query with additional sub-queries) that will allow for a more precise estimate of restricted data. Our experiments further demonstrate that in cases where access control condition is complex (and high-overhead), the information about underlying hidden data is impossible to hide. In such scenario, a user could estimate the number of rows being hidden and approximate the ranges and distribution of indexed columns in a database table.

Our next goal is to quantify how such attack can be used to discover data ranges within the column without knowing the underlying data distribution and to consider the same attack against different DBMSes. Finally, our goal is to help detect and counter such attacks. Side-channel attacks are difficult to counter because they are exploiting normal system behavior. However, we believe that database administrators can protect the data by scanning for query patterns associated with a side-channel timing attack.

Acknowledgments

This work was partially funded by US National Science Foundation Grant IIP-2016548, Argonne National Laboratory, and Louisiana Board of Regents Grant AWD-10000153. Argonne National Laboratory's work was supported by the U.S. Department of Energy, Office of Science, under contract DE-AC02-06CH11357.

References

[1] 2003. https://docs.oracle.com/cd/B12037_01/network.101/b10773/accessre.htm
[2] 2017. Database 2 Day + Security Guide. https://docs.oracle.com/database/121/TDPSG/GUID-72D524FF-5A86-495A-9D12-14CB13819D42.htm#TDPSG90066
[3] 2017. Database security guide. https://docs.oracle.com/en/database/oracle/oracle-database/21/dbseg/using-oracle-vpd-to-control-data-access.html#GUID-06022729-9210-4895-BF04-6177713C65A7
[4] 2023. https://www.ibm.com/docs/en/db2/11.5?topic=lbac-how-security-labels-are-compared
[5] 2023. Advanced security guide. https://docs.oracle.com/en/database/oracle/oracle-database/21/asoag/introduction-to-oracle-advanced-security.html#GUID-F91E22D5-7B2D-4D67-BC96-4C738C54FFE1
[6] Franziska Boenisch, Reinhard Munz, Marcel Tiepelt, Simon Hanisch, Christiane Kuhn, and Paul Francis. 2021. Side-channel attacks on query-based data anonymization. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 1254–1265.
[7] David Brumley and Dan Boneh. 2005. Remote timing attacks are practical. *Computer Networks* 48, 5 (2005), 701–716.
[8] Burleson Consulting. 2017. Row Level Security Tips. http://www.dba-oracle.com/concepts/restricting_access.htm
[9] Ember Crooks. 2018. LBAC – label based Access Control. https://datageek.blog/en/2014/12/09/lbac-label-based-access-control/
[10] 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.* (2023).
[11] Ariel Futoransky, Damián Saura, and Ariel Waissbein. 2007. The ND2DB Attack: Database Content Extraction Using Timing Attacks on the Indexing Algorithms.. In *WOOT*.
[12] Peng Gu, Dylan Stow, Russell Barnes, Eren Kursun, and Yuan Xie. 2016. Thermal-aware 3D design for side-channel information leakage. In *2016 IEEE 34th International Conference on Computer Design (ICCD)*. IEEE, 520–527.
[13] Paul Kocher, Joshua Jaffe, and Benjamin Jun. 1999. Differential power analysis. In *Advances in Cryptology—CRYPTO'99: 19th Annual International Cryptology Conference Santa Barbara, California, USA, August 15–19, 1999 Proceedings 19*. Springer, 388–397.
[14] PostgreSQL. 2023. Row Security Policies. www.postgresql.org/docs/current/ddl-rowsecurity.html
[15] Mark Randolph and William Diehl. 2020. Power Side-Channel Attack Analysis: A Review of 20 Years of Study for the Layman. 4, 2 (2020), 15. https://doi.org/10.3390/cryptography4020015 Number: 2 Publisher: Multidisciplinary Digital Publishing Institute.
[16] Aria Shahverdi, Mahammad Shirinov, and Dana Dachman-Soled. 2021. Database Reconstruction from Noisy Volumes: A Cache {Side-Channel} Attack on {SQLite}. In *30th USENIX Security Symposium (USENIX Security 21)*. 1019–1035.