Synthesis of Allowlists for Runtime Protection against SQLi

Kostyantyn Vorobyov
kostyantyn.x.vorobyov@oracle.com
Oracle Labs
Brisbane, Queensland, Australia

François Gauthier
francois.gauthier@oracle.com
Oracle Labs
Brisbane, Queensland, Australia

Padmanabhan Krishnan
paddy.krishnan@oracle.com
Oracle Labs
Brisbane, Queensland, Australia

ABSTRACT

Data is the new oil. This metaphor is commonly used to highlight the fact that data is a highly valuable commodity. Nowadays, much of worldwide data sits in SQL databases and transits through web-based applications of all kinds. As the value of data increases and attracts more attention from malicious actors, application protections against SQL injections need to become more sophisticated. Although SQL injections have been known for many years, they are still one of the top security vulnerabilities. For example, in 2022 more than 1000 CVEs related to SQL injection were reported. We propose a runtime application protection approach that infers and constrains the information that can be disclosed by database-backed applications. Where existing approaches use syntax or hand-crafted features as a proxy for information disclosure, we propose a lightweight information disclosure model that faithfully captures the semantics of SQL and achieves finer-grain security.

KEYWORDS

SQLi, Synthesis, Generalisation

ACM Reference Format:
Kostyantyn Vorobyov, François Gauthier, and Padmanabhan Krishnan. 2024. Synthesis of Allowlists for Runtime Protection against SQLi. In *New Ideas and Emerging Results (ICSE-NIER'24).* ACM, New York, NY, USA, 5 pages.

1 INTRODUCTION

A common way of securing web applications against cyber threats is by using a Web Application Firewall (WAF) that monitors HTTP(S) traffic and blocks suspicious requests. To identify SQL injections (SQLi), WAFs analyse request payloads for syntactic anomalies. For instance, payloads containing apostrophes, comments, or SQL keywords could be blocked as potentially dangerous.

While effective at deterring simple SQLi attacks, WAFs can be evaded using malicious payloads whose syntax resemble benign inputs. For instance, an attacker can send a harmless-looking hexadecimal string that will be executed as code or use functions like `concat` and `chr` to obfuscate malicious payloads. WAF's analysis is also limited to network payloads, which may not contain SQL fragments but still influence how SQL queries are built at runtime. Such payloads are typically application-specific and difficult to detect at the network level.

A more robust prevention of SQLi attacks is offered by database firewalls or runtime application self protection (RASP) tools that intercept and check incoming SQL queries before they reach the database. Checking queries for SQLi is delegated to a security policy that is typically generated from known benign queries.

A popular way of detecting SQLi attacks is by comparing the parse trees of known benign and incoming queries. For example, Sofia [6] is a SQLi detector that uses parse tree similarity to cluster known benign queries and report incoming queries that are too distant from any benign cluster as potential attacks. SEPTIC [12], on the other hand, generates context-sensitive policies (or profiles) as parse-tree signatures. At runtime, a SQLi attack is reported if the signature of an incoming query does not match any pre-recorded benign signatures. Other techniques utilising parse trees for prevention of SQLi attacks include [2] and [4].

Because they report *all* syntactic deviations as SQLi, parse tree-based methods can be prone to false positives and negatives. Dynamic yet benign queries that add or remove predicates e.g., a search and filter functionality, can be rejected while syntactically similar yet malicious queries, e.g., mimicry attacks [19] can be accepted. Syntactic properties alone are thus insufficient to accurately model benign queries. In an attempt to produce syntax-agnostic models, SQLBlock [9] generates context-sensitive "profiles" that capture features such as table names, logical operators, functions and types of queries (e.g., **`SELECT`**) emitted from a given function in the application. At runtime, incoming queries are allowed if *all* their features are listed in the profile matching their calling function. In practice, SQLBlock has been shown to be effective against common SQLi attacks that access system tables, execute new functions or use additional operators. However, depending on the training queries, SQLBlock can also be prone to false positives and negatives, as highlighted in the motivating examples below. Furthermore, because SQLBlock operates at the granularity of database tables, it provides limited protection against data ex-filtration attacks targeting specific columns or rows.

Column and row granularity is typically required by applications that need to achieve a trade-off between performance (e.g., minimising the number of tables and table joins), maintainability (e.g., minimising the number user- role- or application-specific views) and privacy and might not be obvious from the open-source applications that have been studied in past work. This need has however been recognised by several database vendors who offer various flavours of row- and column-based access control [1, 14–17].

Motivating Example. Consider a "members" table, as shown in Table 1, that contains sensitive information, highlighted in red. Next, assume that the following queries pertaining to the "members" table were collected from the application under test:

Table 1: The "members" table

	1	2	3	4	5
1	fname	lname	ssn	dept	email
2	John	Adams	XXXX	HR	j.adams@foo.com
3	Mary	Baker	XXXX	HR	m.baker@bar.com
4	Juan	Lopez	XXXX	Sales	j.lopez@foo.com
5	Amir	Zafar	XXXX	Sales	a.zafar@bar.com
6	Kira	Patel	XXXX	CEO	k.patel@baz.com

```
SELECT lname, fname, email FROM members WHERE dept = 'HR' ;
SELECT dept, email FROM members WHERE lname = 'Baker' OR
↪  dept = 'Sales';
```

These queries access or disclose the blue cells in Table 1 and make up our training set. After training, our approach would infer that the yellow cells can be disclosed, based on the fact that columns 1 and 2 *and* rows 4 and 5 have been partially disclosed during training. Let's now consider queries encountered at runtime and how various approaches might classify them.

```
SELECT email FROM members;
```

Because the above query uses fewer operators than training queries, it will be allowed by SQLBlock despite the fact that it discloses sensitive fields from row 6. On the other hand SQLBlock will reject the following query.

```
SELECT email FROM members WHERE dept = 'HR' AND lname =
↪  'Baker';
```

This query, while syntactically similar to training queries, uses more operators in its predicate to disclose less information (i.e. cells (3,4) and (3,5) only). The addition of an **AND** operator would cause SQLBlock to reject this legitimate query. In the next example,

```
SELECT * FROM members WHERE dept = 'HR';
```

the * operator causes the disclosure of sensitive social security numbers (SSN). Reasoning about the * requires knowledge of the database schema, which is not available in the AST. Hence SQLBlock would allow this query. The final example,

```
SELECT fname || ' ' || lname AS name FROM Members M HAVING
↪  dept IN ('Sales', 'HR');
```

discloses similar information than training queries (e.g. blue and yellow cells), it is syntactically different and introduces several new operators. This benign query would be rejected outright by syntax-based approaches and SQLBlock.

In summary, existing approaches are designed to prevent common SQLi attacks that alter the syntax of queries or attempt to access additional tables, functions and operators. As shown in our examples, however, using syntax or hand-picked features as a proxy for information disclosure is fundamentally limited and might result in false positives and negatives that will hinder usability, privacy and security.

In this paper we present a novel allowlist-based technique to prevent data ex-filtration attacks at the application level, before queries reach the database. Similar to SQLBlock, our technique automatically synthesises context-sensitive security policies from benign queries. Unlike existing bodies of work, however, our approach uses ASTs to derive an information flow-based allowlist from queries. At runtime, incoming queries are allowed if they not disclose more information than permitted by the allowlist. Queries that violate this condition are flagged as malicious and rejected. The following sections detail the process by which we synthesise allowlists, in the form of generalised abstract queries, from sets of benign queries.

2 OVERVIEW

The idea of synthesising SQL queries is not new. Past work has included synthesis from natural language descriptions [21], input examples [20], or using machine learning techniques from features in SQL queries [3]. Our work is focused on generating an abstract query that over-approximates a collection of queries and based on a simplified version of the information flow model from Guarnieri et al. [8]. As the aim is not to prove non-interference, we do not consider explicit security labels associated with a trace-based semantics. This simplified model is generalised as part of the synthesis process.

2.1 Information Model

Our information model of a SQL query is based on *disclosure* – columns disclosed to the user (e.g., via a select statement but can also include **INSERT**, **UPDATE** or **DELETE**), *access* – columns used to compute the disclosed information, and *predicates* – conditions for disclosure to occur expressed as a boolean formula. For a query q, we let $D(q)$ denote disclosed columns, $A(q)$ denote accessed, and $P(q)$ denote predicates. For example, in the following query

```
SELECT dept, email FROM members WHERE lname = 'Baker';
```

$D(q)$ is $\{dept, email\}$, $A(q)$ is $\{lname\}$, and $P(q)$ is $lname = \text{'Baker'}$.

Information Tuple. In a query, different columns can be disclosed under different conditions. For instance, the following example that uses a **LEFT JOIN** returns all rows from the left (*members*) table even if the condition in the **ON** clause evaluates to false.

```
SELECT M.lname, O.info FROM members AS M LEFT JOIN orders AS
↪  O ON M.email = O.email;
```

That is, *O.info* is disclosed under predicate $M.email = O.email$ and requires access to both *email* columns, whereas *lname* is disclosed unconditionally. To account for this issue we represent a query as a set of *information tuples* where each tuple $\langle d, A, P\rangle$ models the disclosure of a single column d, the set of accessed columns A needed for the disclosure and the predicates P leading to the disclosure. An information tuple is thus an information model of a query that discloses a single column. The above example query is represented by (or decomposed into) two tuples: $\langle M.lname, \emptyset, true\rangle$ and $\langle O.info, \{M.email, O.email\}, M.email = O.email\rangle$

Query Decomposition. Computing the decomposition of a query into a set of information tuples is done by traversing a graph that represents the semantics of the query in a structural form, called a structural graph. The nodes of the structural graph correspond to syntactic elements (e.g., **SELECT**, **WHERE**) and annotated with predicates. The edges of the structural graph capture direction of information flow. While supporting the full semantics of SQL is beyond the scope of this paper, we explain key concepts using a simple example shown below.

```
SELECT M.lname, O.info
FROM members AS M
```

```
    LEFT JOIN orders AS O ON M.email = O.email
WHERE M.dept = 'HR'
```

The structural graph for the above query is shown in Figure 1. Note the difference in direction of information flow between the join and the tables. This models **LEFT JOIN** used in this query, where **ON** predicates (associated only with table *orders*) are not reachable from the *members* table graph node. For each disclosed column in the query we compute an information tuple by identifying the table of that tuple and traversing the graph starting with that node. The *predicates* part of the tuple is constructed by conjunction of all reachable predicates. The set of accessed columns is then constructed from all encountered columns. For instance, to compute the information tuple for *O.info* we first locate the table node (O) and collect predicates from reachable **ON** and **WHERE** nodes that generates tuple $\langle O.info, \{O.email, M.email, M.dept\}, M.email = O.email \wedge M.dept = \text{'HR'}\rangle$. The information tuple of *M.lname* is then $\langle M.lname, \{M.dept\}, M.dept = \text{'HR'}\rangle$.

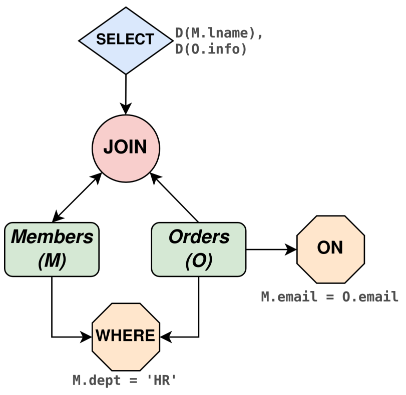

Figure 1: Example Structural Graph

In summary, structural decomposition represents a SQL query as a set of information tuples, where each tuple captures information about a distinct disclosed column. We let $Inf(q)$ denote the set of information tuples of a query q.

2.2 Information Disclosure

Given two information tuples $t_1 = \langle d_1, A_1, P_1\rangle$ and $t_2 = \langle d_2, A_2, P_2\rangle$ we say that t_1 discloses no more information than t_2 (written as $t_1 \preceq t_2$) if the following three conditions hold.

(1) $d_1 = d_2$
(2) $A_1 \subseteq A_2$ and
(3) $P_1 \Rightarrow P_2$.

Query Disclosure. Given two queries, q_1 and q_2, we say that q_1 discloses no more information than q_2 if for each information tuple t in $Inf(q_1)$ there exists a tuple t' in $Inf(q_2)$ such that $t \preceq t'$.

In the context of security checking we can say that an input query q_i is malicious with respect to an allowlist given by query q_a, if the property q_i does not disclose more information than q_a.

3 GENERALISATION

We construct an allowlist of permitted queries by using a training set of known benign queries. Then, an incoming query q_i is permitted by the allowlist $\mathcal{A} = \{q_1, \ldots, q_n\}$ if $\mathcal{A}$ contains some query q_a that discloses at least as much information as q_i (i.e., $q_i \preceq q_a$). Since a training set of queries is rarely complete, however, such an allowlist is likely to lead to false positives. Furthermore, a large set of training queries can result in performance bottlenecks because rejecting a malicious query requires checking disclosure of all queries in the allowlist.

To solve the above issues, we generalise $\mathcal{A}$. Given a set of benign queries $\{q_1, \ldots, q_n\}$ whose semantics is represented as $\{[\![q_1]\!], \ldots, [\![q_n]\!]\}$ the aim is to synthesise a generalised query q^+, such that $[\![q^+]\!]$ over-approximates all the queries in $\{[\![q_1]\!], \ldots, [\![q_n]\!]\}$.

Recall that a query q is represented by a set of information tuples $Inf(q)$ obtained from structural decomposition. The generalised query q^+ can be represented by a collection of generalised information tuples, i.e., the problem of query generalisation can be reduced to generalisation of information tuples. The generalised query is synthesised from the initial allowlist $\mathcal{A} = \{q_1, \ldots, q_n\}$ as follows:

(1) Use structural decomposition to generate the set of information tuples of $\mathcal{A}$, i.e., $\{Inf(q_1) \cup \ldots \cup Inf(q_n)\}$.
(2) Collect and generalise tuples disclosing the same column.

For example, consider an initial allowlist $\mathcal{A}$ of queries q_1 and q_2 that disclose columns $\{a, b\}$ and $\{a, b, c\}$ respectively. In the first step $\mathcal{A}$ is converted into the set of information tuples $\{t_1(a), t_1(b), t_2(a), t_2(b), t_2(c)\}$, where $t_1(a)$ denotes the information tuple disclosing column a in query q_1 and so on. The generalised allowlist $\mathcal{A}_g$ is then the set of tuples $\{t_g(a), t_g(b), t_g(c)\}$, where $t_g(a)$ denotes a tuple generalising $t_1(a)$ and $t_2(a)$ and so on.

One way to generalise information tuples is by combining respective predicates and access columns. For example, generalisation over $\langle d, \{a, b\}, p_1\rangle$ and $\langle d, \{b, c\}, p_2\rangle$ becomes $\langle d, \{a, b, c\}, p_1 \vee p_2\rangle$. Such an approach, however, may be considered too permissive. For example, for disclosure of the column d, a query accessing columns a and c together will be permitted even though it has never been observed. Furthermore, access to column c will now be permitted under the predicate p_1 that has not been observed either.

To resolve this issue, we extend information tuples to capture groups of columns observed together in the training set along with their respective predicates. We now show synthesis of an extended information tuple using a simple example. Consider information tuples $t_1 = \langle d, \{a, b\}, p_1\rangle$ and $t_2 = \langle d, \{b, c\}, p_2\rangle$. These tuples both disclose column d and therefore can be generalised. An extended information tuple combining t_1 and t_2 is shown below (the disclosed column d is omitted to simplify the presentation).

Accessed	Predicate	Correlation
a	p_1	$\{b\}$
b	$p_1 \vee p_2$	$\{a, c\}$
c	p_2	$\{b\}$

A row of the table above describes an accessed column of the tuple, the predicate the column can be accessed under and its correlated set (i.e., set of columns it can be used with). The first row thus shows that column a can be accessed under predicate p_1 and together with column b. This information is gathered from t_1 tuple exclusively because a has not been observed in t_2. Column b, on the other hand, appears in both tuples, therefore its correlated set is $\{a, c\}$ and the predicate is a disjunction of predicates from tuples where it appears ($p_1 \vee p_2$).

More formally, an *extended information tuple* is a quadruple $\langle d, A, pred, corr\rangle$, where d is a disclosed column, A is a set of accessed columns, $pred$ is a mapping over A that associates accessed columns with predicates, and $corr$ is a mapping over A that associates accessed columns with sets of their correlated columns, e.g., in the example above A is $\{a, b, c\}$, $pred(b)$ is $p_1 \vee p_2$ and $corr(b)$ is $\{a, c\}$.

In summary, the generalised allowlist $\mathcal{A}_G$ consists of a set of extended information tuples that disclose distinct columns. We use $Inf_G(\{q_1, \dots, q_n\})$ to denote generalisation over a training query set $\{q_1, \dots, q_n\}$.

Predicate Generalisation. Another dimension to generalisation is the generalisation over predicates generated from queries. Overall, standard techniques (e.g., abstract interpretation [10]) can be used for predicate generalisation. One simple yet practical approach is the generalisation over constants, e.g., replacing all integers by 1, all strings by *'S'* and so on. The state-of-the-art tools such as Sofia or SQLBlock employ similar strategies and discard constants in subtrees or profiles. Another practical option is to use range abstraction [5, 18]. This type of generalisation is relevant for queries accepting limited numeric inputs. For more complex expressions domains such as Pentagons [11] or Octagons [13] can be used.

While specific predicate abstraction and generalisation is beyond the scope of this paper, we consider predicate generalisation to be a necessary step in generating a robust allowlist.

Cardinality Generalisation. Another dimension to generalisation is the generalisation over table columns. For instance, if a query accesses more columns of a table (say T) than some configured threshold then the generalisation can allow access to all columns of that table. In a given extended information tuple this generalisation should update both the set of accepted columns and the correlation mapping to include all columns of T.

3.1 Generalised Information Disclosure

Given two extended information tuples $t_1 = \langle d_1, A_1, pred_1, corr_1\rangle$ and $t_2 = \langle d_2, A_2, pred_2, corr_2\rangle$ we say that t_1 discloses no more information than t_2 (denoted as $t_1 \preceq t_2$) if

$d_1 = d_2$ and for each column a from A_1
(1) $a \in A_2$
(2) $pred_1(a) \Rightarrow pred_2(a)$
(3) $corr_1(a) \subseteq corr_2(a)$

Allowlist Checking. Let $\mathcal{A}_G = Inf_G(\{q_1, \dots, q_n\})$ be a generalised allowlist and q_i be an incoming query. We say that q_i is accepted by the generalised allowlist $\mathcal{A}_G$ if for each extended information tuple t_i in $Inf_G(\{q_i\})$ there exists an extended information tuple t_a in $\mathcal{A}_G$ such that $t_i \preceq t_a$.

3.2 Example

```
SELECT lname, fname, email FROM members WHERE dept = 'HR';
SELECT dept, email FROM members WHERE lname = 'Baker' OR
↪  dept = 'Sales';
```

The allowlist generated from the above training queries comprises 4 extended information tuples that allow disclosure of columns *fname*, *lname*, *email* and *dept*. Consider column *email* disclosed in both queries. The structural graph decomposition generates the following information tuples

$\langle email, \{dept\}, dept = \text{'HR'}\rangle$
$\langle email, \{lname, dept\}, lname = \text{'Baker'} \vee dept = \text{'Sales'}\rangle$

and further generalisation of these tuples (including predicate generalisation over constants) yields an extended information tuple

Accessed	Predicate	Correlation
$dept$	$lname = \text{'S'} \vee dept = \text{'S'}$	$\{lname, dept\}$
$lname$	$lname = \text{'S'} \vee dept = \text{'S'}$	$\{lname, dept\}$

Let us consider several incoming queries.

```
SELECT email FROM members;
```

Having no predicates our approach considers that this query accesses all columns including *SSN*. Since the allowlist does not permit access to *SSN* (even if it has not been disclosed) the query is rejected.

```
SELECT email FROM members WHERE dept = 'HR' AND lname =
↪  'Baker';
```

Our approach permits this query because it discloses no more information than specified by the allowlist, i.e., (1) information tuple disclosing *email* exists, (2) access to *dept* and *lname* has been observed, (3) correlation permits using *dept* and *lname* together and (4) the predicate checks under predicate generalisation hold.

```
SELECT * FROM members WHERE dept = 'HR';
```

This query is rejected because it discloses all column including *SSN*, whereas the allowlist does not permit disclosure of that column.

```
SELECT fname || ' ' || lname AS name FROM Members M HAVING
↪  dept IN ('Sales', 'HR');
```

Even though syntactically different from the training queries our approach allows it. The query discloses *fname* and *lname* columns under predicate $dept = \text{'S'}$ obtained by generalisation of the predicate build from the `IN` expression. This query therefore discloses no more information than the first query in the training set.

4 FUTURE WORK

In this paper we have described how finer-grain information protection in a world where "data is the new oil" [7] can be achieved. Our work is far from complete, however. Our allowlist synthesis strategy is designed to be used for runtime application protection, where overheads and false positives must be low. Achieving an acceptable trade-off between security, usability and performance will require careful experimentation on real-world applications and vulnerabilities. For example, we are currently investigating how context-sensitivity — synthesising and enforcing different allowlists for different execution contexts (e.g. call stacks) — could help us synthesise tighter allowlists, especially in applications that emit a wide spectrum of SQL queries. We are also considering various caching and optimisation strategies to lower the overhead of our runtime protections.

REFERENCES

[1] Oracle Database 21c. 2023. Using Oracle Virtual Private Database to Control Data Access. 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

[2] Sruthi Bandhakavi, Prithvi Bisht, P. Madhusudan, and V. N. Venkatakrishnan. [n. d.]. CANDID: preventing sql injection attacks using dynamic candidate evaluations. In *Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007, Alexandria, Virginia, USA, October 28-31, 2007*, Peng Ning, Sabrina De Capitani di Vimercati, and Paul F. Syverson (Eds.). ACM, 12–24.

[3] E. Bertino, A. Kamra, and J. P. Early. 2007. Profiling Database Application to Detect SQL Injection Attacks. In *IPCCC*.

[4] Gregory Buehrer, Bruce W. Weide, and Paolo A. G. Sivilotti. 2005. Using parse tree validation to prevent SQL injection attacks. In *Proceedings of the 5th International Workshop on Software Engineering and Middleware, SEM 2005, Lisbon, Portugal, September 5-6, 2005*, Elisabetta Di Nitto and Amy L. Murphy (Eds.). ACM, 106–113.

[5] V. H. S. Campos, R. E. Rodrigues, I. R. de Assis Costa a nd D. do Couto Texeira, and F. M. Q. Pereira. [n. d.]. A Tool for the Range Analysis of Whole Programs. http://range-analysis.googlecode.com/.

[6] Mariano Ceccato, Cu D. Nguyen, Dennis Appelt, and Lionel C. Briand. 2016. SOFIA: an automated security oracle for black-box testing of SQL-injection vulnerabilities. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, David Lo, Sven Apel, and Sarfraz Khurshid (Eds.). ACM, 167–177.

[7] The Economist. 2017. The world's most valuable resource is no longer oil, but data. https://www.economist.com/leaders/2017/05/06/the-worlds-most-valuable-resource-is-no-longer-oil-but-data

[8] M. Guarnieri, M. Balliu, D. Schoepe, D. Basin, and A. Sabelfeld. 2019. Information-Flow Control for Database-Backed Applications. In *EuroS&P*.

[9] Rasoul Jahanshahi, Adam Doupé, and Manuel Egele. 2020. You shall not pass: Mitigating SQL Injection Attacks on Legacy Web Applications. In *ASIA CCS '20: The 15th ACM Asia Conference on Computer and Communications Security, Taipei, Taiwan, October 5-9, 2020*, Hung-Min Sun, Shiuh-Pyng Shieh, Guofei Gu, and Giuseppe Ateniese (Eds.). ACM, 445–457.

[10] B. Jeannet and A. Miné. 2009. Apron: A Library of Numerical Abstract Domains for Static Analysis. In *CAV*.

[11] F. Logozzo and M. Fähndrich. 2008. Pentagons: a weakly relational abstract domain for the efficient validation of array accesses. In *Proceedings of the ACM Symposium on Applied Computing (SAC)*. ACM, 184–188.

[12] Ibéria Medeiros, Miguel Beatriz, Nuno Neves, and Miguel Correia. 2019. SEPTIC: Detecting Injection Attacks and Vulnerabilities Inside the DBMS. *IEEE Trans. Reliab.* 68, 3 (2019), 1168–1188.

[13] A. Miné. 2001. The Octagon abstract domain. In *Proceedings Eighth Working Conference on Reverse Engineering*. IEEE, 310–319.

[14] MySQL. 2023. Column privileges. https://dev.mysql.com/doc/refman/8.0/en/grant.html#grant-column-privileges

[15] PostgreSQL. 2023. Row Security Policies. https://www.postgresql.org/docs/current/ddl-rowsecurity.html

[16] SQL Server. 2023. Column-level Security. https://learn.microsoft.com/en-us/azure/synapse-analytics/sql-data-warehouse/column-level-security

[17] SQL Server. 2023. Row-level Security. https://learn.microsoft.com/en-us/sql/relational-databases/security/row-level-security

[18] Z. Su and D. Wagner. 2005. A Class of Polynomially Solvable Range Constraints for Interval Analysis without Widenings. *Theoretical Computer Science* 345 (2005), 122–138.

[19] David Wagner and Paolo Soto. 2002. Mimicry attacks on host-based intrusion detection systems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*. 255–264.

[20] C. Wang, A. Cheung, and R. Bodik. 2017. Synthesizing Highly Expressive SQL Queries from Input-Output Examples. In *PLDI*.

[21] N. Yaghmazadeh, Y. Wang, I. Dillig, and Thomas Dillig. 2017. SQLizer: Query Synthesis from Natural Language. In *OOPSLA*.