Qapla: Policy compliance for database-backed systems

Aastha Mehta[1], Eslam Elnikety[1], Katura Harvey[1,2], Deepak Garg[1], and Peter Druschel[1]

[1]Max Planck Institute for Software Systems (MPI-SWS), Saarland Informatics Campus
[2]University of Maryland, College Park

Abstract

Many database-backed systems store confidential data that is accessed on behalf of users with different privileges. Policies governing access are often fine-grained, being specific to users, time, accessed columns and rows, values in the database (e.g., user roles), and operators used in queries (e.g., aggregators, group by, and join). Today, applications are often relied upon to issue policy compliant queries or filter the results of non-compliant queries, which is vulnerable to application errors. Qapla provides an alternate approach to policy enforcement that neither depends on application correctness, nor on specialized database support. In Qapla, policies are specific to rows and columns and may additionally refer to the querier's identity and time, are specified in SQL, and stored in the database itself. We prototype Qapla in a database adapter, and evaluate it by enforcing applicable policies in the HotCRP conference management system and a system for managing academic job applications.

1 Introduction

Confidential information stored in systems backed by relational databases is often subject to complex access policies. In a personnel management system, for instance, ordinary employees may query their own personal information but not that of others. Members of a workers' council may be able to query the columns containing employee names and ages separately, but not together, to prevent them from linking employees to their ages. Similarly, members of the payroll department may not be able to query the health history of individual employees, but they may be able to query aggregates over the health histories of all employees.

Today, such fine-grained policies are enforced by adding policy compliance checks to application code wherever the database is queried. This approach is cumbersome, error-prone, and inappropriate: Policies are usually derived from the privacy requirements of the broader legal/enterprise context and are code-independent, yet every code path in every application leading to a query must be instrumented by a programmer to perform a check. It is easy to miss such checks. Moreover, when the policy changes, application developers must update these checks everywhere.

Alternatively, policy compliance can rely on fine-grained access-control support in the underlying database management system (DBMS). Unfortunately, the extent of the support and the language used to express the policies varies across DBMSs. For instance, a cell-level policy can be specified in Oracle using its VPD technology [11], whereas the same policy will require a combination of views (for column access control) and row-level policies in PostgreSQL [7].

Furthermore, DBMS support for policies is limited to standard row-, column- and cell-level access control but, in practice, policies are often more complex. For instance, a policy may prohibit the *linking or joining* of two or more columns, while allowing those columns to be read independently. Similarly, a policy may allow certain principals to query for aggregates (sometimes based on user-defined functions), while prohibiting them from reading individual values. To the best of our knowledge, such complex policies can be implemented in existing DBMSs only through extensive use of application-specific views. However, views can neither support link policies nor are they transparent to applications. When using policy-specific views, all queries, even if they are compliant, must be modified whenever policies change.

Goals. Based on these observations, our goal is to provide a policy compliance system for database-backed applications that satisfies the following requirements. (i) It must be able to express a rich class of policies including standard fine-grained row-, column- and cell-level policies and also complex policies that limit data linking or allow aggregation. (ii) The policy specification must be associated with the database schema and independent of applications, and it must be simple and intuitive for policy administrators to adopt. (iii) The system should not depend on specific support from the DBMS and it should be transparent to applications that issue policy-compliant queries.

We emphasize that our primary goal is to protect the confidentiality of data in the face of *application bugs*. The threat is not from active attacks, although our design defends against some kinds of application compromise.

Our design, Qapla, is a policy-compliance middleware for database-backed systems, which satisfies all the aforementioned goals with moderate overheads on application performance. In Qapla, policies are *specified* in a SQL-like language, as a function of the database schema, and stored in the database (in separate tables). SQL is a natural choice for Qapla's policy language since its syntax is widely understood. Furthermore, the use of SQL syntax leads to a simple enforcement mechanism that we describe below.

For policy *enforcement*, Qapla integrates a reference monitor with a generic database adapter, which intercepts all application queries, looks up applicable policies, and rewrites queries to ensure compliance. The SQL-like syntax of Qapla policies simplifies query rewriting. Moreover, the enforcement is transparent to application queries that are already policy compliant, so the application has to be changed only where its queries are not policy compliant.

Qapla requires no changes to and no specific support from the DBMS (although we describe how database-specific support like materialized views can be used to optimize Qapla's performance). Furthermore, since the Qapla reference monitor is integrated in a generic database adapter and does not depend on DBMS-specific access control support, it is portable across DBMSs. Qapla removes the often large and rapidly evolving applications from the codebase trusted for compliance, simplifies new applications by obviating the need for pervasive filtering code, and avoids compliance bugs due to incorrect or missing application checks.

Qapla's approach of stating policies in a high-level, declarative, and familiar SQL-like language, associated with the database schema and not within the application code provides additional benefits. Declarative, schema-based policies are easier to reason about, analyze, and audit than policies written in application code. Moreover, policy changes can be affected reliably based on the schema, without requiring inspection of queries or modification of compliant queries. The use of SQL-like syntax and the high-level of policy abstraction further aid policy writing, debugging and audit.

We demonstrate Qapla's portability by incorporating it with PHP's and Python's database adapters, and using it to enforce fine-grained policies in two applications: the widely-used HotCRP conference management system [2], and the APPLY system for managing academic job applications, which we use in our organization. HotCRP includes fine-grained policies to maintain confidentiality of paper submissions, provide author and reviewer anonymity, and prevent untimely disclosure of results to authors and PC members. APPLY likewise has policies to control access to application materials, reference letters, and evaluators' notes depending on user roles and to allow users access to aggregated historical information yet prevent them from seeing their own past case materials. The policies cover many important application workflows such as user login, searching for papers, reviews, comments by authors, chair, reviewers, etc., in HotCRP, and applications, letter request, review, and search in APPLY. We identified and implemented a total of 30 policies in HotCRP and 41 policies in APPLY. The policies are concise, specified in one place, and tend to require only local changes or extensions when new features are introduced to applications.

An experimental evaluation shows that Qapla incurs moderate overheads. Interestingly, we also observe that Qapla overheads are generally lower than the overheads of native access control support in a commercial database on policies that can be expressed using the latter.

To summarize, our contributions lie in the architecture, design, policy language and evaluation of Qapla, which enables the specification and enforcement of a rich class of complex and fine-grained policies (including those based on linking and aggregation) in a database-agnostic and application-transparent manner.

Organization. We present Qapla's policy language in Section 2 and its architecture in Section 3. Our application of Qapla to HotCRP and APPLY is described in Section 4, followed by an experimental evaluation in Section 5. We discuss related work in Section 6 and conclude in Section 7.

2 Qapla policy framework

Qapla allows a policy compliance team to associate a set of policies with a database schema. These policies specify data confidentiality requirements that take into account the database schema, contents, the authenticated user, time and operations like joins, aggregations and UDFs. We do not consider data integrity policies although we believe they can be added to our design.

Every Qapla policy applies to a class of queries and specifies how those queries must be restricted to be compliant. These restrictions are specified as SQL WHERE clauses that are added to the query by Qapla before the query is executed, thus filtering out non-compliant records. We formally define when a policy applies to a query and the query rewriting procedure in Section 3. An application can obtain a tuple using a query only if (a) at least one policy applies to that query, and (b) the query rewritten under the restrictions of the policy produces that tuple. If no policy applies to a query, the query is not executed. This whitelist principle ensures that data is accessed only due to some explicitly written policy and never leaked due to accidental omission of policies.

The remaining section provides an overview of Qapla policies using the running example of the human resource's database of a fictitious company, Acme. The database has three tables `Employees(empID, name, address, age, gender, dept)`, `Payroll(empID, salary)` and `Benefits(empID, health_plan)`. The first table maps employees to their home address, age, gender and department. The second table maps employees to their salary, while the third table specifies which health plan each employee subscribes to.

Single-column policies. The simplest Qapla policy protects a single database column by specifying which rows of the column can be accessed by each user, and when. It has the form: `col` :- *W*. Here, *W* is a SQL WHERE clause that specifies which rows from the column `col` can be returned. *W* may refer to the authenticated user and the wall clock time using the variables $user and $time, which are instantiated by the Qapla reference monitor when the clause is added to the query. The policy applies to any query that references only the column `col` (queries that read more than one column are subject to link policies described later).

Example 1 (name, age, health_plan) The names of Acme's employees should be accessible to all other employees. The following policy specifies this.

```
name :- EXISTS(SELECT 1 FROM Employees
         WHERE empID = $user)
```

The SQL fragment EXISTS(...) specifies a condition that holds only if the authenticated user exists in the table `Employees`. An identical policy applies to the columns `age` and `health_plan`. Reading the columns in isolation only allows enumeration of the set of ages or health plans of all employees.

Example 2 (address, salary) The columns `address` and `salary` can be read only by members of the HR (human resources) department. Additionally, an employee may read his or her own address or salary. The following policy enforces this on `address`. A similar policy applies to `salary`.

```
address :- (empID = $user) OR
           EXISTS(SELECT 1 FROM Employees
           WHERE empID = $user AND dept = HR)
```

Compared to the policy of `name`, this policy allows different employees access to different entries in `address`. Note that the WHERE clause is organized as a disjunction of conditions, one for each class of users.

This is an example of a role-based access control (RBAC) policy, where an employee's role is dictated by her affiliation with a particular department. This policy relies on the availability of the mapping from users to their roles in the database itself. In applications where this mapping is outside the database (e.g., on a file system), Qapla's policy language can be easily extended to support predicates that lookup this mapping outside the database. Qapla can interpret these non-database predicates in the policies using native procedures, and apply the remaining SQL policy to the database queries.

Link policies. When a query reads two or more columns, the applicable policy may be more restrictive than the individual policies of all the columns read, because additional information can be exposed by linking the columns to each other, as in the following example.

Example 3 (linking name and age) The policies of the columns `name` and `age` allow any employee to read these columns individually (Example 1). However, not every employee should be able to read the columns `name` and `age` *together* since that reveals every employee's age, which may be private. The right policy is that only members of HR and an employee himself/herself may read the employee's `name` and `age` together. In Qapla, this policy is expressed by mentioning both columns `age` and `name` to the left of :- in the policy.

```
{name,age} :- Employees : ((empID = $user) OR
              EXISTS(SELECT 1 FROM Employees
              WHERE empID = $user AND
                    dept = HR))
```

Such policies, which apply to simultaneous access of two or more columns, are called *link policies*. Their general form is

$$\{\texttt{col}_1, \ldots, \texttt{col}_n\} \text{ :- } \textit{filter conditions}$$

with *filter conditions* of the form T_1:W_1 , ..., T_m:W_m. Here, $\{\texttt{col}_1, \ldots, \texttt{col}_n\}$ are columns spanning the tables $T_1, \ldots, T_m$ and $W_1, \ldots, W_m$ are separate WHERE clauses for these tables. This policy applies to any query that reads a subset of the columns $\{\texttt{col}_1, \ldots, \texttt{col}_n\}$ (for any purpose including projection, selection, joining, grouping or aggregation). The WHERE clauses of all the tables mentioned in the query are added to the query by Qapla (see Section 3 for details).

Columns in separate tables. When the goal is to restrict the linking of data in two or more *separate* tables, the effect of a link policy can sometimes be simulated by simply restricting access to the individual columns containing the common keys of the two tables. However, when different sets of columns spanning the tables need different policies, the policies must be specified using the general form of link policies described above.

Transformation policies. Applications often apply functions or transformations to columns to hide sensitive information. A transformed column may have more permissive policies than the column itself. Qapla directly supports such transformations-aware policies.

Example 4 Suppose Acme provides a home-to-office shuttle service to its employees, run by Acme's "logistics" department. The shuttle service has a fixed stop in every neighborhood that houses an employee but it is not door-to-door. In order to provide this service, members of the logistics department must know the neighborhood in which every employee lives, but not their precise home addresses. To enforce this, the privacy compliance team can create a user-defined function (UDF), `neigh`, that maps an address to a neighborhood, and add the following Qapla policy.

$$\begin{array}{l}
\{\texttt{name}, \texttt{address}[\texttt{neigh}]\} \text{ :-} \\
\quad (\texttt{empID} = \$\text{user}) \text{ OR} \\
\quad \text{EXISTS}(\text{SELECT } 1 \text{ FROM } \texttt{Employees} \\
\qquad \text{WHERE } \texttt{empID} = \$\text{user AND} \\
\qquad (\texttt{dept} = \text{HR OR } \underline{\texttt{dept} = \text{logistics}}))
\end{array}$$

This policy says that an employee's `name` and `neigh(address)` can be linked by the employee, members of HR and members of logistics. The policy is strictly more permissive than the policy on $\{\texttt{name}, \texttt{address}\}$, which allows access only to the respective employee and HR, but not to logistics. The revised policy allows logistics to run the query "SELECT `name, neigh(address)` FROM `Employees`", but not "SELECT `name, address` FROM `Employees`".

The general form of a Qapla transformation policy is

$$\{\texttt{col}_1[\texttt{t}_1], \ldots, \texttt{col}_n[\texttt{t}_n]\} \text{ :- } \textit{filter conditions}$$

The *filter conditions* are of the same form as in a link policy. The policy applies to any query that accesses a subset of the columns $\texttt{col}_1, \ldots, \texttt{col}_n$ but only after the respective transformations $\texttt{t}_1, \ldots, \texttt{t}_n$ have been applied.

Aggregation policies. Many applications declassify aggregate statistics on otherwise private columns. Accordingly, Qapla provides *aggregation policies*. An aggregation policy specifies two sets of columns: 1) *LS* (link set)—columns which can be projected, used to join or group data (SQL's GROUP BY) or be aggregated in a query, and 2) *JS* (join set)—columns which can be used only to join tables in the query and nothing else. With each column in *LS* an optional transformation or aggregation operation can be specified, which restricts the use of that column to only that transformation or aggregation. The general syntax is

$$\begin{array}{l}
\{JS = \{\texttt{jcol}_1, \ldots, \texttt{jcol}_m\}, \\
\;\; LS = \{\texttt{col}_1[\texttt{t}_1], \ldots, \texttt{col}_n[\texttt{t}_n]\}\} \text{ :- } \textit{filter conditions}
\end{array}$$

Example 5 Suppose Acme has a workers' council (WoC) that periodically computes salary statistics to ensure fairness in worker compensation. One statistic it computes is the distribution of average salary over age ranges (20-30 years, 30-40 years, etc.). Rather than provide WoC full access to the `Employees` table, the policy compliance team can selectively provide WoC rights to compute only such statistics by writing the following aggregation policy. Here, `age_range` is a function that rounds an individual's age to a 10-year range.

$$\begin{array}{l}
\{JS = \{\texttt{Payroll.empID}, \texttt{Employees.empID}\}, \\
\;\; LS = \{\texttt{age}[\texttt{age\_range}], \texttt{salary}[\texttt{AVG}]\}\} \text{ :-} \\
\quad \text{EXISTS}(\text{SELECT } 1 \text{ FROM } \texttt{Employees} \\
\quad \text{WHERE } \texttt{empID} = \$\text{user AND} \\
\quad (\texttt{dept} = \text{HR OR } \underline{\texttt{dept} = \text{WoC}}))
\end{array}$$

This policy allows WoC to run any query that joins tables `Payroll` and `Employees`, and then uses only `age_range(age)` and average on `salary` (in any way). For example, it allows the following two queries among many other instances of similar queries:

(i) SELECT `AVG(salary), age_range(age)` FROM `Employees,Payroll` GROUP BY `age_range(age)` HAVING `AVG(salary)` > 50000
which lists age groups with average salaries above 50000.

(ii) SELECT `AVG(salary)` FROM `Employees, Payroll` WHERE `age_range(age)` = (30,40)
which lists the average salary of a specific age group.

Correctly, the policy does not allow queries that look at the `age` or `salary` columns directly. For instance, the following query is disallowed by the policy: SELECT `AVG(salary)` WHERE `age` = 75.

Relation between policy classes. Qapla's four policy classes—single-column policies, link policies, transformation policies and aggregation policies—are increasingly more general. Single-column policies are an instance of link policies, where the set of linked columns is a singleton. Link policies are a special case of transformation policies where the transformations are identity functions. A transformation policy S :- *filter conditions* is the same as the aggregation policy $\{JS = \{\}, LS = S\}$:- *filter conditions*.

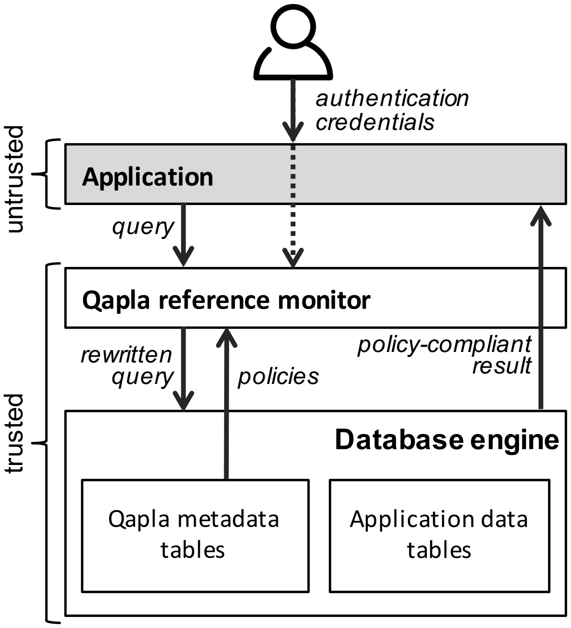

Figure 1: Qapla architecture

Policy inference heuristics. To reduce the burden of specifying policies, Qapla provides safe heuristics.

Heuristic 1: A link policy for a set of columns also automatically applies to any subset of those columns since reading a subset only reveals less information than does reading the whole set. Thus, there is no need to specify a link policy on a subset unless the subset's policy is strictly more permissive than the policy of the whole set, and some application needs the permissiveness.

Heuristic 2: If a query uses column transformations or aggregations but a specific applicable transformation or aggregation policy does not exist, Qapla applies the link policy of the set of columns that occur in the query, if one exists. This is safe because transforming or aggregating a column always reduces the information revealed.

Heuristic 3: In place of writing an explicit link policy on a set of columns, the designer can explicitly instruct Qapla to automatically construct a link policy for a set of columns by combining the policies of the individual columns in the set. This synthesized policy applies the *filter conditions* of the individual columns even when they are read together. This is useful in some cases. For instance, we may want to allow only HR members and an employee simultaneous access to the employee's `name` and `address`. However, this is exactly the policy of the individual column `address` (Example 2), so in this case, the policy designer can ask Qapla to synthesize the link policy for {`name,address`} by combining the individual policies of `name` and `address`.

3 Qapla design

Figure 1 depicts Qapla's architecture. Qapla's metadata and policies are stored in the database separate from the application data. The Qapla reference monitor authenticates with the database with its own unique credentials, and it has the exclusive privilege to access all tables directly. It intercepts the application's database queries, and associates each query with the authenticated end user on whose behalf the query was issued by the application. The query is rewritten to ensure its compliance with policies, and the rewritten query is executed by the database.

3.1 Threat model

Qapla is designed to prevent data leaks due to application *bugs* that result in non-compliant queries to the database. Qapla intercepts all queries to the database in a reference monitor and rewrites the queries to make them policy compliant. In our current design and implementation, the reference monitor runs in the same address space as the application. Consequently, any application-level bugs or vulnerabilities that circumvent this monitor to access the database directly or steal the reference monitor's privileged database credentials are out of scope. Additionally, we rely on the application to correctly tell Qapla which user's behalf it (the application) is acting on. However, it is not difficult to change this design to avoid these limitations (see Section 3.5 for initial ideas).

We also assume that users do not collude offline to combine non-overlapping parts of the database they are individually authorized to read, and that individual users do not link information they have obtained in separate queries.

The Qapla reference monitor, the database adapter it is embedded in, the database system, the operating system, the storage layer, and the communication between the database adapter and the database system are assumed to be correctly configured and trusted. The database curator or compliance team is assumed to have installed correct policies, and any information referenced by policies is assumed to be correctly stored in the database. Under these assumptions, Qapla guarantees that only policy compliant query results are returned to the application.

3.2 Enforcement

Qapla's policy enforcement on a query consists of two steps: 1) Identifying the set of policies that apply to the query, and 2) Rewriting the query to filter out tuples disallowed by all the applicable policies. We describe the two steps in detail.

Applicable policies. Internally, Qapla treats every policy as an aggregation policy of the form $\{JS, LS\}$:- *filter conditions*, where *JS* and *LS* are, respectively, the set of columns that may be used to (only) join two or more tables, and the set of columns that may be pro-

```
Does a policy apply to a query?
input: Query Q; Policy $\{JS, LS\}$ :- filter conditions
output: true if policy applies to Q, false otherwise

$\{js, ls\}$ = parseQuery(Q)
if ($js \not\subseteq JS$) return false
for each column c in $ls$:
    if (c $\notin LS$) then return false
for each transformed/aggregated column c[t] in $ls$:
    if (c[t] $\notin LS$ and c $\notin LS$) then return false
return true
```

Figure 2: Algorithm to decide if a policy applies to a query

jected, grouped by and aggregated. As explained in Section 2, this is the most general form of policies; all single-column, link and transformation policies can be expressed in this form. Qapla parses every application query to extract the corresponding sets *js* and *ls* of columns that are used only to join and those that the query actually projects, groups by, or aggregates.

A policy applies to a query if the query's use of the columns *js* and *ls* is allowed by the corresponding sets *JS* and *LS* of the policy. Formally, the policy applies to the query when $js \subseteq JS$ and when every column c and every transformed column c[t] in *ls* is *dominated* by a column or transformed column in *LS*. Domination is defined as follows: Every (transformed) column dominates itself, and a column dominates any transformation of itself. Thus, a policy with $JS = \{$Benefits.empID, Employees.empID$\}$ and $LS = \{$age, health_plan$\}$ applies to a query with $js = \{$Benefits.empID, Employees.empID$\}$ and $ls = \{$age[age_range], health_plan[COUNT]$\}$. Figure 2 summarizes this algorithm.

To efficiently find all policies that apply to a query, Qapla maintains two data structures. The first data structure maps every pair of a column and a transformation (that applies to the column) to a bitvector representing the policies in the system. The *i*th bit is set in the bitvector of the (transformed) column *j* if policy *i*'s *LS* contains a column that dominates *j*. To find all applicable policies whose *LS* matches a given query's *ls*, Qapla simply takes the bit-wise AND of the bitvectors of all (transformed) columns in *ls*. The second data structure is similar but applies to *JS* and allows finding all policies whose *JS* matches a query's *js*.

Query rewriting algorithm. The query rewriting algorithm modifies an application query to make it compliant. In the simple and common case where only one policy applies to the query, the policy rewriting algorithm replaces each reference to a table in the query with a subquery that generates a list of rows compliant with the *filter conditions* of the columns accessed from the table. The subquery is of the form (SELECT * FROM table WHERE *list-of-conditions*), where *list-of-conditions* are the *filter conditions* of the table provided in the policy. The overall effect is that the application query is executed over joins of policy-compliant sub-tables of one or more database tables, where the sub-tables have been created using the *filter conditions* of the applicable policy.

Example 6 In the context of Acme's database, assume that some link policy exists for the column set $\{$name, age, health_plan, Employees.empID, Benefits.empID$\}$ and that it specifies the WHERE clauses f_E and f_B for filtering the tables Employees and Benefits, respectively. Consider the following query: SELECT name, age, health_plan FROM Employees JOIN Benefits ON Employees.empID = Benefits.empID. This query will be rewritten to:

```
SELECT name, age, health_plan FROM
  (SELECT * FROM Employees WHERE $f_E$)
  Employees JOIN
  (SELECT * FROM Benefits WHERE $f_B$)
  Benefits ON
  (Employees.empID = Benefits.empID)
```

When more than one policy applies to a query and the query does not return an aggregate, Qapla rewrites the query according to each applicable policy and takes a SQL UNION of these. This ensures that a tuple exists in the result only when at least one applicable policy allows it. If the query returns an aggregate value and more than one policy applies, Qapla picks the first applicable policy, but the application may override this to a specific applicable policy at the cost of minor changes to its code. (We have not encountered the need for such changes in our evaluation.)

3.3 Optimizations

We describe three optimizations to reduce the overhead of policy enforcement in Qapla. In our current prototype, we implement the first optimization, and the second optimization only partially. Implementing the third optimization is not difficult[1].

Query template cache. The Qapla reference monitor implements a query template cache to amortize the overhead of parsing and rewriting queries with the same structure. A query template is a query with all its constant values replaced with placeholder variables. The Qapla template cache maps query templates to their rewritten forms. When a query is received, Qapla converts the query to a template and checks if a query template with the same hash is cached (if the application

[1] The USENIX Security'17 version of our paper states that only the first optimization is implemented. However, we do have a partial implementation of the second optimization.

query is already parametrized, Qapla hashes it directly). For a hit, Qapla retrieves the associated rewritten query template, and binds its variables with the values from the submitted query. For a miss, Qapla parses and rewrites the query with the applicable policies, and inserts the resulting rewritten query template into the cache.

Partial evaluation. The Qapla reference monitor often generates complex rewritten queries containing several nested sub-queries accessing one or more tables, and having large filter conditions. Executing the query efficiently depends on the ability of the DBMS to generate an efficient execution plan for the rewritten query. To reduce the complexity of the rewritten query, Qapla can pre-evaluate parts of the rewritten query that do not depend on database values (e.g., parts that depend only on time, or the identity of the user on whose behalf the application makes the access) before posting the query to the database. This can significantly simplify the query since any predicates connected by 'AND' to a pre-evaluated predicate that is false can all be replaced by a single false before the query is sent to the database. Similarly, any predicates connected by 'OR' to a pre-evaluated predicate that is true can all be replaced by a single true. In our prototype we only pre-evaluate time-based conditions.

Materialized Views. To offset the cost of policy checks during query evaluation, Qapla can create materialized views, one for each (group of) user(s) with similar permissions, by applying applicable policies to the tables offline. In a group's materialized view, every cell inaccessible to the group is replaced with a special value that is not a legal value for the underlying table. At runtime, every application query is run against the materialized view appropriate for the authenticated user. The query is rewritten by Qapla to disregard any record that contains the special value in a field that is used in the query.[2] Our preliminary evaluation suggests that this optimization can reduce runtime overheads on read-intensive workloads by an order of magnitude. However, proportional to the number of user groups with different policies, maintaining materialized views adds storage cost and runtime overhead to propagate updates to all materialized views.

3.4 Cell-blinding mode

The policy enforcement algorithm described in Section 3.2 drops a row during query execution if any field in the row is inaccessible according to the policy and is used in the query. This *row-suppression mode* of policy enforcement ensures that information about an inaccessible field cannot be inferred even when that information is correlated with other fields in the row. This makes row-suppression a very safe choice for policy enforcement (and, hence, Qapla uses it by default). However, row-suppression is not the only possible way of enforcing Qapla's policies. We briefly describe here a second mode of policy enforcement, the *cell-blinding mode*.

The primary consideration for the cell-blinding mode is compatibility with legacy applications, which may issue broad queries that project more columns than actually necessary, and eventually remove these extra columns internally in their own code. With the row-suppression mode, such broad queries may result in fewer records than expected by the application. Transitioning such applications to make them compatible with row-suppression may require effort and time, as developers may have to rewrite queries to not project unnecessary columns. This transition can be particularly difficult when the set of necessary columns depends on the application state.

The cell-blinding mode changes the semantics of policy enforcement to compromise some security and efficiency in return for accommodating overly broad queries. In this mode, Qapla rewrites the application queries to replace (blind) inaccessible cells with special values that can be returned in results, before executing the original query's logic. (This replacement is identical to the replacement of inaccessible cells in the creation of materialized views from Section 3.3 but, here, the special values must not depend on any secrets since they can be returned directly in query results.)

However, the cell-blinding mode has two drawbacks. First, if some fields of a record are inaccessible according to the policy, the record is still returned (with the inaccessible fields blinded). This leaks some information when the presence of the record in the database is sensitive and when blinded fields are correlated with other non-blinded fields. Second, the cell-blinding mode imposes significant overhead on query execution (up to two orders of magnitude for some queries with MySQL) due to the need to check policies on, and possibly blind, individual cells in every query. We believe that the use of materialized views described in Section 3.3 can reduce this overhead substantially. A full study of this approach remains as future work.

Due to these limitations of the cell-blinding mode, it is preferable to use the row-suppression mode and to modify the application to restrict overly general queries. The rest of the paper uses only the row-suppression mode of policy enforcement.

3.5 Discussion

We discuss some limitations of Qapla's current threat model and some ideas on how to strengthen the design

[2] For confidentiality, it is insufficient to disregard a record only when one of its inaccessible fields is projected. It is also necessary to disregard a record if one of its inaccessible fields will be tested by the query's WHERE clause(s). Doing so prevents implicit information leaks through the WHERE clause(s).

to eliminate these threats. We also discuss how Qapla can be used for logging policy violations.

Isolation of the reference monitor. Currently, we assume that the application, which runs in the same address space as the reference monitor, cannot circumvent the reference monitor or steal its authentication credentials. However, this is not a fundamental limitation. To provide guarantees against a malicious application, we can also isolate the reference monitor in a separate process [15, 24], or co-locate it with the DB servers. There are also efficient ways of isolating the reference monitor within the application address space, such as using light-weight contexts [30].

User authentication. Qapla's current design requires the application to specify which user's behalf it is acting on. An application may specify the wrong user to Qapla due to a bug, thus breaking Qapla's policy enforcement. This problem can be easily addressed by having the user authenticate to the reference monitor instead of the application. The application can then ask for the authenticated user's identity from the reference monitor.

Protection against offline linking attacks. Qapla does not protect against offline linking attacks that span multiple queries. For two queries whose results can be linked offline (such as in example 3), randomizing the order of query results may mitigate the attack in some cases. However, randomizing the order of query results cannot eliminate linking attacks in all cases. In particular, some linking may be possible due to information contained in the data itself (e.g., names may have high correlation with the nationality of users, or fine-grained aggregate queries may reveal individual records). We expect the policy designer to be aware of potential data leaks of this type, and design the policies such that compliant queries return a minimum threshold number of results (similar to k-anonymity [37]). Tools to check such conditions on policies can be easily designed.

Support for logging. A natural question is whether we can modify Qapla's reference monitor to detect and log non-compliant queries (e.g., for debugging or auditing). While this is not a design goal, Qapla can be used to detect non-compliant queries to a limited extent – by rerunning a query twice, with and without policy checks and comparing the results for any differences. Non-compliant queries can then be logged.

3.6 Implementation

The Qapla implementation consists of about 20K lines of C code. It provides the API to create application-specific policies, associates policies with column identifiers, and maintains an in-memory mapping from column identifiers to associated policies. It also provides an API for setting application-specific user authentication parameters in the reference monitor. Qapla uses an existing SQL parser from the MySQL workbench [4] to extract accessed tables and columns. A rewrite module implements the lookup for applicable policies and the query rewriting algorithm. A template cache module maintains a cache of rewritten query templates, and a customizable translation module can translate the SQL dialect of one DBMS to that of another, allowing Qapla uses across DBMSs. In our evaluation, we translate MySQL queries into a commercial DBMS's queries.

Qapla can support existing PHP and Python based applications. For PHP applications, we modified the PHP Data Objects (PDO) [5] module in the PHP interpreter. For Python applications, we rely on the Django framework [1], which provides an object-relational mapping (ORM) API for database interaction. Django provides a database-independent abstraction to the application developer. We modified this abstraction and interface with the Qapla reference monitor using the ctypes library. Both PDO and Django can be used to connect with different databases, such as MySQL, SQLite, MSSQL and Oracle. Modifications to PDO and Django were limited to 135 and 141 lines of code, respectively.

4 Case studies

In this section, we describe our use of Qapla to ensure policy compliance in HotCRP and APPLY.

4.1 HotCRP compliance with Qapla

Policies. We studied HotCRP's schema and wrote policies based on our knowledge of its workflow. In many cases, we reverse-engineered HotCRP's policies by inspecting its code base to confirm and correct our intuition. In total, we specified 30 policies for the 22 tables and 215 columns in the schema of HotCRP version 2.99, which supports a broad range of configurations for a conference. The policies cover a single-track conference with a double-blind submission process, handling of chair conflicts with paper managers, and a review process with no rebuttal. Due to space constraints we cannot show all the policies but Table 1 shows the policies associated with important tables like contacts, papers, reviews, and conflicts. The policies are explained in plain English for clarity and brevity of exposition but are actually written in the language introduced in Section 2. Macros abbreviate common SQL fragments that appear in many policies. Many of the policies are fine-grained access control predicates on user, time, and the content of various database tuples. There are also link and aggregation policies.

Link policy example. An author can independently view the names of all PC members, his own paper submission, and the reviews for his papers after the notification date. However, the author is not allowed to see

id	table	column list	allow the authenticated user U access to row R if ...
C1	ContactInfo	email	(U is a chair) or (R is U's contact information) or (U and R are on the PC)
C2	ContactInfo	password	(R is U's contact information) or (U is chair)
P1	Paper	paperId, title, abstract, timeSubmitted, timeWithdrawn	(U is R's author) or (U is on the PC and either the submission deadline has not passed or R was submitted fully)
P2	PaperStorage	paperStorageId, size, paper, other paper metadata	(U is R's author) or (U is on the PC and R was submitted fully)
P3	Paper	authorInformation, collaborators	(U is R's author) or (the notification deadline has passed, R was accepted and U is on the PC)
P4	Paper	outcome	(the notification deadline has passed and U is R's author or a PC member) or (U is R's paper manager or a non-conflicted PC member)
P5	Paper	shepherdContactId	(the notification deadline has passed and U is R's author) or (U is R's paper manager or a non-conflicted PC member)
P6	Paper	managerContactId	(U is a chair or R's manager or a non-conflicted PC member)
P7	Paper	leadContactId	(U is R's manager) or (U has submitted a review for R) or (U is a non-conflicted PC member and the discussion has started)
R1	PaperReview	reviewId, paperId, <review content>, reviewSubmitted	(P7 conditions) or (the notification deadline has passed and U is R's author or a non-conflicted PC member)
R2	PaperReview	contactId, reviewEditVersion, reviewRound, requestedBy, reviewType, commentToPC, reviewToken, timeRequested	(P7 conditions) or (R is a sub-review and U is the reviewer who asked for it)
C	PaperConflict	all columns	(U is R's author) or (U is a chair) or (U is a PC member and the subject of R)
AL	ActionLog	all columns	U is R's manager or a non-conflicted chair
AO	Outcome statistics	Total number of submissions and accepted papers	the notification deadline has passed
AS	Avg. review scores	Average score across all submitted reviews	U is a PC member
AR	Review statistics	Number of reviews submitted by each PC member	U is a PC member and statistics excludes each row conflicted with U

Table 1: Subset of HotCRP policies

the join of the three columns, which reveals the reviewers' identities. In the HotCRP schema, these columns reside in three different tables (ContactInfo, Paper, and PaperReview). The PaperReview table can be joined with Contact via the contactId key column, and with Paper via the paperId key column. The link policy can be implemented by specifying a restrictive policy for PaperReview.contactId, which does not allow the author to read the column (R2 in Table 1). The policy prevents PC authors from identifying reviewers of their own papers, yet allows them to know and participate in discussions with reviewers of non-conflicted papers.

Aggregation policy example. During the review and discussion process, HotCRP provides aggregate statistics to all reviewers. The statistics include the average review score across all papers as well as the number of reviews submitted by each PC member. To allow this feature to function correctly, we specify two aggregate policies (AS and AR in Table 1), one allowing an AVG computation on the overAllMerit score field and the other allowing a COUNT on the review field grouped by PC member. In the second case, conflicted papers must be excluded.

Implementation effort. We replaced the MySQLi database adapter [6] normally used in HotCRP with our modified Qapla-enabled PDO adapter. We modified HotCRP to forward the user authentication credentials to the Qapla reference monitor. (Apache was configured to fork a separate process for each HotCRP user session, so there is a separate instance of the adapter/reference monitor for each user session.) HotCRP uses broad queries and relies on post-filtering to remove the information the user should not see. We changed approximately 150 LoC in HotCRP's code to make these queries policy compliant so that they can work with Qapla. In most cases, we added a couple of queries to identify the contextual information required to convert the broad queries into more specific queries. With Qapla in place, we can remove the post-filtering queries, but we ignored them for now. Table 2 summarizes the changes we made in HotCRP.

Type of change	lines of code
Replace MySQLi with PDO adapter	96
Change paper query	110
Change review query	25
Change comment query	17
Authentication with Qapla	5

Table 2: HotCRP changes

4.2 APPLY compliance with Qapla

We briefly describe our use of Qapla to protect the application management system (APPLY) for managing faculty, PhD, post-doc, and internship applications in our organization. APPLY's database is similar to the fictitious Acme database from Section 2 and the confidentiality concerns are also similar. The database contains user accounts for applicants and reviewers, contact and application details of the applicants, references, and internal application review aspects such as comments. Users within the organization are assigned roles based on what application type (intern, PhD, postdoc, faculty) they are allowed to access. APPLY prevents reviewers from accessing applications created before they joined the organization. Additionally, APPLY allows explicit delegation of the right to view (sets of) applications to specific users or roles, and disallows a user from accessing an application in case of a conflict of interest. A single policy condition, listed below, covers a large number of columns across many tables.

User U has access to application A if:
(A is U's own application) or
((U joined before A was submitted) and
(U has no conflict of interest with A) and
((U is faculty) or (U has been delegated access to A)))

There are additional restrictions on many sensitive columns and exceptions for other roles. For example, users cannot see reference letters written for them and an applicant's country of birth and citizenship cannot be seen by reviewers until the application has been accepted (to prevent discrimination). Office staff can access all applicant names, emails, and postal addresses (to correspond with them) and CVs of accepted applicants (to prepare contracts). In total, we wrote 41 policies for APPLY.

Implementation effort. APPLY is implemented using Django and Python and stores its data in a database comprising 36 tables and 202 columns. The modifications necessary for APPLY were quite similar to those required for HotCRP. First, we modified 10 LoC to pass user authentication credentials to the Qapla reference monitor. Second, we changed 63 LoC to remove unused columns from queries to make them compatible with our policies.

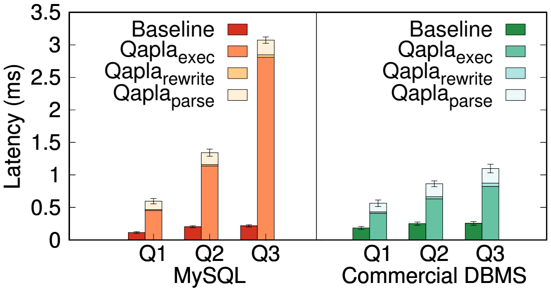

Figure 3: HotCRP query latency on MySQL and a commercial DBMS (baseline is measured without Qapla).

5 Evaluation

In this section, we present results of an experimental evaluation of Qapla's overhead when used with HotCRP and APPLY. We also perform a brief security evaluation by injecting HotCRP bugs that existed in prior versions.

All experiments were performed on Dell Precision T1600 workstations with an Intel Xeon E3-1225 3.1Ghz quad core CPU, 8GB main memory, and 10Gbit Ethernet links. The client and server machines were running OpenSuse Linux 12.1 (kernel version 3.1.10-1.29, x86-64). The HotCRP server software consisted of Apache HTTP server 2.4.18, PHP 5.6.15, and HotCRP 2.99. The APPLY software included Python 2.7 and Django 1.9.7. By default, the backend database for each application was MySQL Server 5.7.11. In some experiments, we used instead a well-known commercial DBMS, which remains unnamed due to license restrictions on the publication of benchmark results. Both DBMSs were configured with a query cache of 500MB, unless stated differently. Unless stated otherwise, the results correspond to the default setup with MySQL.

For HotCRP, we used an anonymized database snapshot of a major conference hosted on HotCRP in the past. The database included about 150 submissions, over 400 contacts, and over 700 reviews. The papers were reviewed in 3 rounds. For APPLY, we used an anonymized database snapshot of 9396 applications received by our organization for internships, doctoral, postdoc and faculty positions.

5.1 Query latency

The first experiment measures Qapla's latency overhead on individual queries. Qapla introduces overheads associated with query parsing, query rewriting, and execut-

	Baseline query	Policy summary
Q1	select title, abstract from Paper where paperId=X	paper author **or** PC member and the paper is under submission
Q2	select title, overAllMerit from Paper join PaperReview where paperId=X	paper author after notification **or** PC member who is not a conflict and has submitted his/her review and the paper is under submission
Q3	select title, overAllMerit, reviewerName from Paper join PaperReview join ContactInfo where paperId=X	PC member who is not conflicted and has submitted his/her review and the paper is under submission

Table 3: Microbenchmarks queries

ing the rewritten query in the database. Table 3 lists the actual HotCRP queries we used in the experiment. Figure 3 shows the average query latency over 1000 trials, on MySQL and on the commercial DBMS. The Qapla latency is broken down into three components: query parsing ($\text{Qapla}_{\text{parse}}$), query rewriting ($\text{Qapla}_{\text{rewrite}}$), and execution of the rewritten query ($\text{Qapla}_{\text{exec}}$). The error bars show the standard deviation. In this experiment, the query caches of the backend DBMSs and Qapla's template cache were disabled.

The contribution of query parsing and rewriting is small, particularly for the more complex queries (on MySQL, 23%, 15%, 8% of the overall query latency for Q1, Q2, and Q3 respectively). The query rewriting overhead is slightly larger with the commercial DBMS, because Qapla has to translate HotCRP queries, which were written for MySQL, to use a SQL syntax appropriate for that DBMS.

In all cases, Qapla's latency overhead is dominated by the execution time of the rewritten queries. A query rewritten with policy conditions may be significantly more complex than the original query as each relation in the query is replaced with a subquery, which may access additional tables that appear in the policy. The efficiency of the rewritten query depends on the database query optimizer being able to generate an efficient query plan. The costs of executing the rewritten queries are lower with the commercial DBMS, whose query optimizer likely is more sophisticated than that of MySQL. Thus, while the commercial DBMS has a slightly higher baseline latency, it is able to execute the rewritten queries relatively faster than MySQL, reducing Qapla's overhead substantially for the more complex queries Q2 and Q3.

Our experiment inflates Qapla's true overheads to some extent, because the rewritten query may require accessing tables that are not mentioned in the original query to ensure compliance. HotCRP accesses these same tables in a separate query to perform the filtering in its own code. To understand this further, we measure the overheads for traces of queries corresponding to user actions in the next experiment.

5.2 Action overhead and latency

A user task in HotCRP and APPLY typically involves multiple actions, such as logging in, clicking on a url to visit a page, and clicking on a button to save a form. For each action, the application in turn issues several SQL queries to get the required data for the response and for policy compliance checks. In this section, we measure the overhead for the sequence of SQL queries involved in several application user tasks. We recorded the SQL queries issued for each of the tasks, and replay the query trace with and without Qapla.

We measured the overhead for executing the query traces and the client-perceived latency overhead under various configurations of the baseline and Qapla. **Base** is the baseline system without Qapla. **Qapla** is Qapla and $\textbf{Qapla}_{\textbf{t-cache}}$ is Qapla with the template cache enabled. In all configurations, the query cache of the backend DBMS was enabled.

5.2.1 HotCRP

In HotCRP, we measured four user tasks: **H1**: As an author, view reviews for a submission (resulting in two actions). **H2**: As a PC member, search for a paper with a keyword, and add a comment (resulting in four actions). **H3**: As PC chair, search for a paper with a keyword, and declare a conflict with a PC member (resulting in five actions). **H4**: As PC chair, invoke the automatic review assignment for all submissions (resulting in three actions).

Task trace execution overhead. First, we measured the average time for executing the traces for tasks H1-H4 on MySQL and the commercial DBMS, respectively, under the three configurations and across 1000 trials (all standard deviations are below 5%).

With MySQL, the relative overheads of $\textbf{Qapla}_{\textbf{t-cache}}$ are 6x, 4.7x, 5.4x, and 7.8x for the tasks, respectively. With the commercial DBMS, the relative overheads of $\textbf{Qapla}_{\textbf{t-cache}}$ are 2.5x, 6.5x, 3.8x, and 2.9x. The results for $\textbf{Qapla}_{\textbf{t-cache}}$ show that Qapla's query template cache is effective in reducing the overhead resulting from Qapla's query parsing and rewriting. The template cache hit rates for each action are 25%, 71%, 82%, and 99%, respectively, yielding a reduction in Qapla's overhead of up to 22%, relative to **Qapla**, for H4 with the commercial DBMS. In the case of H1, we observe a net increase in overhead, because the cost of maintaining the template cache cannot be offset due to the low hit rate.

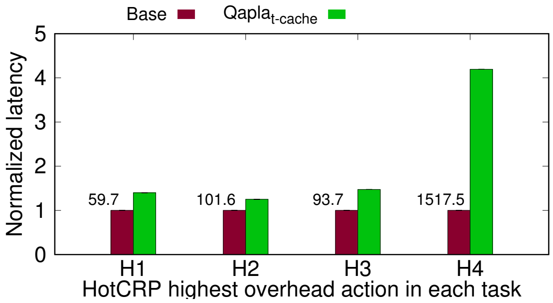

Figure 4: HotCRP client latency of highest overhead action in each task. MySQL, normalized to **Base**. The labels show **Base** absolute latency numbers in milliseconds.

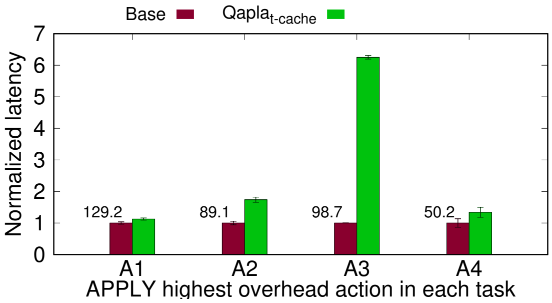

Figure 5: APPLY client latency of highest overhead action, MySQL, normalized to **Base**. The labels show **Base** absolute latency numbers in milliseconds.

Client-side latency. To measure the client-perceived latencies from the perspective of a Web client, we executed each task with a client-side driver that issues HTTP requests to HotCRP for each action involved in performing the task manually. The driver fetches the static HTML pages (but excludes dynamic content such as css, javascripts) from HotCRP and stores them locally. Thus, the experiment includes the overheads of executing PHP code, including database queries, and sending the HTML pages over the network. The template cache as well as the database query cache were flushed after each iteration of a task to fully expose worst-case latency overheads.

Figure 4 shows the average latency, across 1000 trials, of the action with the highest relative overhead in **Qapla$_{\textbf{t-cache}}$** (all standard deviations are below 0.05%). The latency overheads for the actions are 40%, 25%, 47%, and 320%, respectively. The latency could be reduced further by removing the redundant post-filtering queries in HotCRP.

Most of the latency is due to the PHP execution (including database queries), while the network overhead is minimal (0.2ms on average). All the actions are performed in less than 150ms, except the assignment page generated in H4, which takes 1.5 seconds in **Base** and 6.4 seconds in **Qapla$_{\textbf{t-cache}}$**. The assignment algorithm invokes about 3780 queries for the given set of papers and reviewers, while the remaining actions invoke less than 45 queries [3]. H4 is a task used by the PC chair(s) only, and normally only a few times per conference, depending on the number of reviewing rounds.

5.2.2 APPLY

In APPLY, we measured the following tasks: **A1**: As an applicant, view the status of a submitted application (resulting in 3 actions). **A2**: As the faculty member in charge of post-doc applications, mark the status of multiple applications to reject, and send rejection emails to the marked applications (resulting in 7 actions). **A3**: As a faculty member, search for an applicant by name, and request recommendation letters from the applicant's recommenders (resulting in 7 actions). **A4**: As a student reviewing doctorate applications, see a list of doctorate applications currently under review, and view the details of a single application (resulting in 4 actions).

Task execution trace overhead. With MySQL, the relative overheads of **Qapla$_{\textbf{t-cache}}$** are 5.35x, 5.4x, 5.2x, and 4.5x for A1-A4, respectively. With the commercial DBMS, the relative overheads are 4.2x, 3x, 3.3x, and 3.3x, respectively.

Client-side latency. Figure 5 shows the average latency, across 100 trials, of the action with the highest relative overhead in **Qapla$_{\textbf{t-cache}}$** (all standard deviations are below 12%). The latency overheads are low except for an action in A3: 12.5%, 74%, 6.25x, and 34%, respectively. The high overhead in action A3 is due to a single query with very high runtime, which is the cause of nearly all the overhead. On investigating the query behavior, we found that the performance overhead is due to the MySQL query optimizer's inability to deal with a specific query pattern, possibly because this pattern is unlikely to occur in hand-written queries. When we ran the same query on the commercial DBMS, the overheads came down to approximately 50%.

5.3 HotCRP submission throughput

For most HotCRP actions, latency is the metric of interest, as it affects user-perceived delays. Right before a submission deadline, however, throughput is also a measure of interest, because many authors re-submit a final revision of their submission within the last minutes before a deadline. To examine the performance under such conditions, we measured the number of submissions per second HotCRP can sustain with and without Qapla.

In this experiment, clients concurrently upload submissions of size 356KB, which is close to the average submission size in the past HotCRP conference deploy-

[3] The USENIX Security'17 version of our paper mentioned 200 queries, which was the total number queries involved in the entire user task, and not the individual actions.

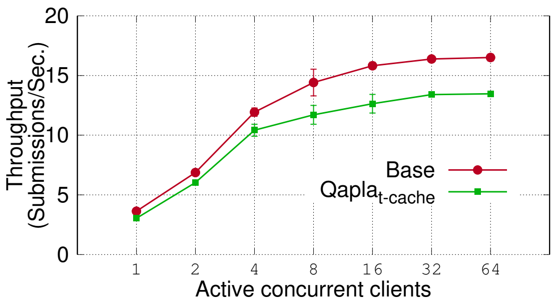

Figure 6: Submission throughput

ment. We varied the number of concurrent clients from 1 to 64. 32 clients were sufficient to saturate the CPU. Prior to the experiment, we cached the entire conference database (∼880MB) in memory. Figure 6 shows the number of submissions per second our HotCRP installation can sustain for different numbers of concurrently connected clients. The results were averaged across 3 runs, each of 120 seconds. The error bars show the standard deviation across 3 runs. The overheads are moderate (below 20.2%), and can be compensated by provisioning a somewhat faster server.

5.4 Trace replay

In the next experiment, we check if HotCRP-Qapla can correctly execute all the actions performed in a HotCRP conference deployment across the various review stages, and produce the same output as the unmodified HotCRP. We use a trace derived from the anonymized logfile of the past HotCRP deployment. The logfile contains over 10,000 log entries that correspond to HotCRP database updates. From it, we constructed a trace by inspecting the HotCRP codebase to determine the set of SELECT queries that typically precede a specific update. For example, submitting a review for a submission must have been preceded by viewing the submission page. Since update queries are not subject to policy checks in Qapla, they are not of interest to our experiment and were not included in the trace. Table 4 shows the actions performed for each log entry.

The trace consists of actions corresponding to four phases: submissions, review, discussion, and post notification stage. We replayed the entire trace against the original HotCRP and HotCRP-Qapla and compared the outputs. Because the trace is read-only, we replayed it against the final state of the HotCRP database at the end of the conference review period. As a result, several policies were not exercised the way they would be in a real deployment and, consequently, the outputs of approximately 27% of the actions differed with and without Qapla enforcement (e.g., withdraw link enabled or not, papers may have been withdrawn at a later stage of the conference). Most of these actions were in the first phase. We verified separately that the relevant policies are enforced as expected.

Log entry	High-level task reads	Count
Create/update account	User logs in, visits her profile	1090
Register, update, submit, or withdraw paper	User logs in, visits the submission page	2082
Added primary/none reviewer	Chair logs in, visits the paper's reviewers assignment/conflicts	1335
Set paper lead-/shepherd	Chair logs in, visits the paper's page	126
Save/submit/delete review/comment	Reviewer logs in, visits the paper's page	3279
Download paper(s)	Reviewer logs in, visits the paper's page, downloads the paper	2582
Send accept/reject notification	Chair logs in, sends decisions to contact authors	2

Table 4: Trace actions for HotCRP

We found that approximately 3% of action outputs differed for other reasons. These reasons are: (i) some non-compliant queries we have not yet modified (e.g., chair unable to make assignments to conflict papers), (ii) policies that are more restrictive than HotCRP assumes (e.g., conflicted PC members unable to download the paper), and (iii) missing policies (e.g., external reviewers not considered).

5.5 Native DBMS access control

As discussed in Section 6, some production DBMS systems support fine-grained access control over tables and views to a limited extent. In this section, we compare using Qapla to enforcing policies directly in our commercial DBMS, which unlike MySQL has some support for fine-grained access control. More precisely, this database supports the equivalent of our single-column policies through a special configuration mechanism. We specified many of the HotCRP policies through this mechanism. However, as our work on HotCRP and APPLY demonstrates, applications often require richer policies (such as link and aggregate policies), which cannot be expressed using the DBMS's policy mechanism. To enforce these policies, we had to create additional views on all HotCRP tables, restrict access to those views and update all queries, whether compliant or not, to use views rather than the underlying tables.

We ran the experiments from Section 5.2 to compare the performance of the DBMS access control mechanism with that of Qapla. Figure 7 shows the average latency for HotCRP actions, across 100 trials, normalized to **Base**. The error bars show the standard deviation. Qapla policy enforcement overhead is lower than

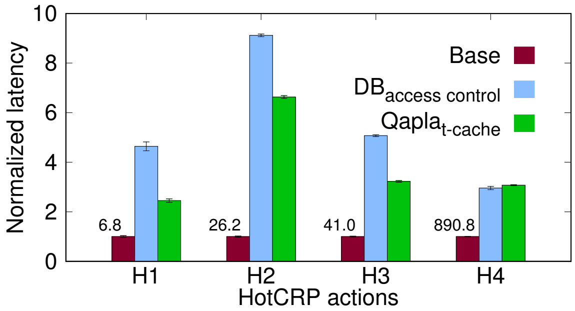

Figure 7: HotCRP action latency with policies enforced using a commercial DBMS's native support for fine-grained access control, normalized to **Base**. Numerical labels indicate absolute **Base** latency in milliseconds.

the overhead of enforcing policies through the DBMS access control for most actions.

The results show that using the native support for fine-grained access control in the commercial DBMS is less efficient than Qapla's policy enforcement. Moreover, to get this level of performance from the commercial DBMS, we had to carefully tune its cache configuration for this experiment. Qapla, on the other hand, achieves better performance with both MySQL and the commercial DBMS, has a DBMS-independent policy language, and does not require the use of views and the resulting changes to compliant queries.

5.6 Fault injection experiments

To verify that Qapla is effective at preventing accidental data leaks, we manually reviewed HotCRP's change logs for bugs that caused data leaks and other policy violations [2]. We are confident that Qapla can prevent any data leaks that are related to missing or incorrect filtering code in HotCRP, which appear to account for the majority of cases.

As a sanity check, we reproduced two sample bugs HotCRP had in the past. One bug notified authors about changes to PC-only fields during response periods. Another bug allowed PC members to search for papers based on their acceptance status and learn of the acceptance of their papers prematurely. We simulated these bugs by making changes to the policy check functions implemented in HotCRP, or by removing the invocations of these functions at certain places in the application. We executed user actions on the buggy HotCRP application with and without Qapla and manually examined the outputs. We verified that Qapla prevents the data from being revealed to unauthorized parties.

There is one class of data leaks that Qapla cannot prevent by itself, namely when a policy depends on incorrect data recorded in the database. For instance, if HotCRP failed to record the conflicts declared by users correctly in the database, Qapla could not prevent the associated leak. We have not found instances of such bugs in HotCRP's change logs, but it is possible that such bugs might occur.

6 Related work

Database access control. The database community has explored fine-grained access and disclosure control within databases, using SQL conditions [28], queries against restricted authorization views [34], and data-derived security views [13]. A formal framework for the design of database access control is presented by Guarnieri *et al.* [26]. In contrast to these systems, Qapla's goal is to provide a portable policy layer that works with existing DBMSs and applications, without relying on any support for policies within the DBMSs.

DataLawyer [40] is a database middleware system that analyzes and rejects non-compliant queries to a relational DBMS. Policies are stated as SQL queries on the database and a usage log, which contains provenance information. DataLawyer supports rich policies, motivated, for instance, by medical databases. Since policies are associated with the entire database, each query must be checked against all policies, each requiring a separate query. Qapla policies are more restricted (e.g., they cannot refer to provenance), but Qapla is much more efficient because policies are indexed by columns. Also Qapla policies are expressed directly as filter conditions, making them easy to write and understand.

In the context of link policies, DiMon [16], its extension D^{2}Mon [38] and Biskup's work [14] enforce access policies by relying on an explicit, complete specification of information that a querier can infer from past queries. These systems deny a query when the query would allow the inference of policy-prohibited information. Qapla's approach is complementary and easier to implement and use; we require the specification of only access rules, abstracting away the inferences those accesses would allow. If indeed a complete specification of possible inferences were to exist, it could be used to assist the policy designer understand the consequences of Qapla policies.

Turan *et al.* [39] present an algorithm to partition a database schema such that two pieces of data that should not be linked (according to a policy) lie in separate logical subschemas. This could be a useful optimization in a Qapla deployment. However, it cannot be used for policies where, of three columns, any two may be linked together, but all three may not be linked simultaneously.

IVD [31] is an authorization system deployed in Facebook, which automatically learns write access control rules on their graph database system from production logs, and enforces them at runtime. Qapla's focus, on the other hand, is on read access control and link policies in relational DBMSs.

Access control in production DBMSs. Current production DBMSs support access control at various levels of granularity. However, the extent of support and the language used to express policies varies among DBMSs and, as far as we know, no DBMS can support all of Qapla's policies without requiring changes to either the schema or queries (including queries that are policy compliant). Qapla enforces fine-grained policies without requiring changes to the schema or policy compliant queries, and requires no support for such policies in the backend DBMS. Moreover, as shown in Section 5.5, Qapla's overhead is lower than a commercial database's native support for fine-grained policies.

Oracle VPD [11] provides extensive support for cell-level access control on tables and views. However, a policy on a table cannot depend on the results of a query on the table itself. Such policies occur in our applications. For instance, the first clause in policy C1 in Table 1 checks that the user is the chair, which is defined using the table that the policy protects. Such policies can be enforced in VPD only by either changing the schema or creating additional views. The use of views, in general, also requires changing queries to use the views instead of the underlying tables. On the other hand, automatic query rewriting as in Qapla is transparent to applications that issue policy compliant queries.

IBM DB2 [9] and SQL Server [10] require a combination of row-level (data-dependent) access control and column masking policies to specify fine-grained policies, which can obscure the policy specification. PostgreSQL [7] has support for row-level policies, but they apply to all columns of a table uniformly. A policy on a subset of columns requires the creation of a view containing only those columns. MySQL and MariaDB do not support data-dependent access control. Fine-grained access control in these DBMSs requires creating a separate view for every group of users with the same privilege, or creating stored procedures and granting privileges to users to execute the procedures [3, 8].

In all production DBMSs we know of, enforcing link policies requires creating a separate view for each policy. Transformation and aggregation policies require separate views or stored procedures. As mentioned above, creating additional views or using stored procedures requires significant changes even to applications that issue only policy-compliant queries.

Database interposition. Interposing on database queries to improve security is a common technique. Perhaps most closely related to our work is CLAMP [33], which has the same goals as Qapla. However, CLAMP's architecture and policy language are different from Qapla's. In CLAMP, when a user initiates a session, the enforcement framework performs user authentication, instantiates a logical view of the database restricted only to data that the user can access (based on applicable policies), and isolates a fresh instance of the application in a virtual machine, restricting it to only communicate with the authenticated user and giving it access to only the logical view of the database via query interposition (as in Qapla). CLAMP's design supports a stronger threat model than Qapla's current prototype—CLAMP isolates user sessions from each other and from the reference monitor, and does not rely on the application to authenticate the user (see Section 3.5)—but the expressiveness of policies, which is really the focal point of our work, is limited in CLAMP. CLAMP only supports per-table policies, which specify the rows that each user has access to. Support for finer policies that differentiate columns of a table from each other or take into account linking, transformation and aggregation is missing in CLAMP. Qapla can be strengthened with CLAMP's isolation and authentication techniques in a straightforward manner.

Diesel [24] is a framework for applying the principle of least privilege on relational databases. Diesel policies specify subsets of a database that each application *module* can access. For example, a policy may specify that a user-facing module can only access the Users table, but not administrative tables, thus limiting damage in the event of a user session compromise. This is very different from Qapla's (and CLAMP's) goal of specifying what data each *user* can access. Nonetheless, Diesel also relies on query interposition (as in Qapla) to enforce its policies.

Passe [15] hardens the web framework Django to isolate application modules from each other. Like Diesel, it uses query interposition to enforce least privilege on data accessible to each module. Unlike Diesel, but like CLAMP and Qapla, Passe's policies are sensitive to the authenticated user. However, Passe's policies are fundamentally different from those of Qapla, CLAMP and Diesel—they enforce data-dependency relations on query parameters. For example, a Passe policy may enforce that the third parameter of the second query made by a specific application module is always a value returned for the first query of the module. Moreover, Passe's policies are not specified by administrators. Instead, they are learnt by automated testing in an offline phase. This learning can have both false positives (it may learn a policy that is too restrictive) and false negatives (it may not learn a required policy). Due to the very different nature of Passe's policies, it is not possible to directly compare their expressiveness to that of Qapla's policies.

Policy languages. EPAL [12] specifies enterprise privacy policies in terms of user categories, data categories, purposes, actions, obligations, and conditions. Qapla relies on authentication-based access control instead of purpose-based access control. Also, Qapla uses SQL syntax to specify policy languages, similar to [19, 28]. SQL is a natural choice to specify policies for database-backed applications, since it enables specifying complex policies on query operators easily, and developers are already familiar with it.

CMS confidentiality. CoCon is a new conference management system whose confidentiality properties were verified formally in the Isabelle proof assistant [27]. Qapla on the other hand, is a general, language-independent runtime compliance layer for database queries, which we have used to enforce compliance in an existing and widely used conference management system, HotCRP.

Privacy in statistical databases. Differential privacy [23] and privacy-preserving queries [32, 17] are focused on statistical databases, where only statistical information, but no information about individual records, should be revealed. Qapla instead focuses on applications that require access to specific database records, subject to fine-grained policies.

Information Flow Control. UrFlow [19], Hails [25], Jacqueline [41], DBTaint [22], RESIN [42], LabelFlow [18] and Nemesis [21] use language-based techniques to enforce information flow control in web applications written in specific languages. In contrast, Qapla can be ported to any language easily but it enforces access policies, not information flow control. Qapla can be integrated with a language-based technique to control information flow with fine-grained policies.

IFDB [36] enforces authorization policies by modifying the PostgreSQL database engine, as well as the application environments in PHP and Python. For enforcing column policies, IFDB relies on declassifying views. Row policies are specified with secrecy and integrity labels, which are associated with database records. IFDB enforces row policies by tracking the labels through the application process and stored procedures. Qapla specifies all policies using one mechanism. Qapla's enforcement uses query rewriting and is database-agnostic.

Sif [20], SeLINQ [35], and Li *et al.* [29] assign labels or security types to database columns, and use security-typed programming languages to write restricted query interfaces to the database and the application code. However, these systems cannot enforce data-dependent policies. Furthermore, some of these systems [35, 29] rely on programming applications in languages that integrate database query mechanisms. While the current prototype of Qapla focuses on applications using SQL to query databases, it can be easily extended to protect applications using other programming paradigms for database queries. Qapla does not impose any restrictions on the programming language for the applications themselves.

7 Conclusion

We have presented and evaluated Qapla, a system that ensures compliance with confidentiality policies in database-backed systems. Fine-grained access policies are stated in a SQL-like language separate from application code, and may refer to user id, time, tables, columns, rows, as well as query operators like aggregation, group by, and join. Qapla adds a reference monitor to the database adapter, which intercepts and rewrites queries to ensure compliance.

Qapla reliably prevents a large class of data confidentiality breaches due to application bugs. Qapla's declarative specification of applicable policies, separate from application code and associated with the database schema, eases the task of specifying, enforcing and auditing confidentiality policy. The system's policy language and enforcement is independent of the DBMS used as a backend.

Acknowledgements

We would like to thank our shepherd, Mathias Payer, and the anonymous reviewers for their valuable feedback. The work was supported in part by the European Research Council (ERC Synergy imPACT 610150) and the German Science Foundation (DFG CRC 1223).

References

[1] Django. https://www.djangoproject.com/.

[2] HotCRP release news. http://read.seas.harvard.edu/~kohler/hotcrp/news.html.

[3] Implementing row level security in MySQL. https://www.sqlmaestro.com/en/resources/all/row_level_security_mysql/.

[4] MySQL Workbench. http://mysqlworkbench.org/.

[5] PHP Data Objects (PDO). http://php.net/manual/en/intro.pdo.php.

[6] PHP MySQL Improved Extension. http://php.net/manual/en/book.mysqli.php.

[7] PostgreSQL 9.5.3 Documentation. https://www.postgresql.org/docs/current/static/ddl-rowsecurity.html.

[8] Protect Your Data: Row-level Security in MariaDB 10.0. https://mariadb.com/blog/protect-your-data-row-level-security-mariadb-100.

[9] Row and Column Access Control Support in IBM DB2 for i. http://www.redbooks.ibm.com/redpapers/pdfs/redp5110.pdf.

[10] SQL Server 2016 Technical Documentation. https://msdn.microsoft.com/en-us/library/dn765131.aspx?f=255&MSPPError=-2147217396.

[11] The Virtual Private Database in Oracle9iR2. http://www.cgisecurity.com/database/oracle/pdf/VPD9ir2twp.pdf, January 2002.

[12] ASHLEY, P., HADA, S., KARJOTH, G., POWERS, C., AND SCHUNTER, M. Enterprise Privacy Authorization Language (EPAL 1.2). http://www.w3.org/Submission/2003/SUBM-EPAL-20031110, 2003.

[13] Bender, G. M., Kot, L., Gehrke, J., and Koch, C. Fine-grained Disclosure Control for App Ecosystems. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD), 2013.*

[14] Biskup, J. History-dependent inference control of queries by dynamic policy adaption. In *Proceedings of the Annual IFIP WG 11.3 Conference Data and Applications Security and Privacy (DBSec), 2011.*

[15] Blankstein, A., and Freedman, M. J. Automating isolation and least privilege in web services. In *Proceedings of the IEEE Symposium on Security and Privacy (SP), 2014.*

[16] Brodsky, A., Frakas, C., and Jajodia, S. Secure databases: constraints, inference channels, and monitoring disclosures. *IEEE Transactions on Knowledge and Data Engineering 12* (2000).

[17] Chen, R., Akkus, I. E., and Francis, P. SplitX: High-performance Private Analytics. In *Proceedings of the ACM SIGCOMM, 2013.*

[18] Chinis, G., Pratikakis, P., Ioannidis, S., and Athanasopoulos, E. Practical Information Flow for Legacy Web Applications. In *Proceedings of the Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS), 2013.*

[19] Chlipala, A. Static Checking of Dynamically-varying Security Policies in Database-backed Applications. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI), 2010.*

[20] Chong, S., Vikram, K., and Myers, A. C. SIF: Enforcing Confidentiality and Integrity in Web Applications. In *Proceedings of the USENIX Security Symposium, 2007.*

[21] Dalton, M., Kozyrakis, C., and Zeldovich, N. Nemesis: Preventing Authentication & Access Control Vulnerabilities in Web Applications. In *Proceedings of the USENIX Security Symposium, 2009.*

[22] Davis, B., and Chen, H. DBTaint: Cross-application Information Flow Tracking via Databases. In *Proceedings of the USENIX conference on Web Application development (WebApps), 2010.*

[23] Dwork, C. Differential Privacy. In *Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP), 2006.*

[24] Felt, A. P., Finifter, M., Weinberger, J., and Wagner, D. Diesel: Applying privilege separation to database access. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS), 2011.*

[25] Giffin, D. B., Levy, A., Stefan, D., Terei, D., Mazières, D., Mitchell, J. C., and Russo, A. Hails: Protecting Data Privacy in Untrusted Web Applications. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI), 2012.*

[26] Guarnieri, M., Marinovic, S., and Basin, D. A. Strong and Provably Secure Database Access Control. *CoRR abs/1512.01479* (2015).

[27] Kanav, S., Lammich, P., and Popescu, A. A Conference Management System with Verified Document Confidentiality. In *Proceedings of the International Conference on Computer Aided Verification (CAV), 2014.*

[28] LeFevre, K., Agrawal, R., Ercegovac, V., Ramakrishnan, R., Xu, Y., and DeWitt, D. Limiting Disclosure in Hippocratic Databases. In *Proceedings of the International Conference on Very Large Data Bases (VLDB), 2004.*

[29] Li, P., and Zdancewic, S. Practical Information-flow Control in Web-Based Information Systems. In *Proceedings of the IEEE Workshop on Computer Security Foundations (CSFW)* (2005).

[30] Litton, J., Vahldiek-Oberwagner, A., Elnikety, E., Garg, D., Bhattacharjee, B., and Druschel, P. Lightweight contexts: An os abstraction for safety and performance. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2016.*

[31] Marinescu, P. D., Perry, C., Pomarole, M., Yuan, T., Tague, P., and Papagiannis, I. IVD: Automatic learning and enforcement of authorization rules in online social networks. In *Proceedings of the IEEE Symposium on Security and Privacy (SP), 2017.*

[32] McSherry, F. D. Privacy Integrated Queries: An Extensible Platform for Privacy-preserving Data Analysis. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD), 2009.*

[33] Parno, B., McCune, J. M., Wendlandt, D., Andersen, D. G., and Perrig, A. Clamp: Practical prevention of large-scale data leaks. In *Proceedings of the 2009 30th IEEE Symposium on Security and Privacy (SP), 2009.*

[34] Rizvi, S., Mendelzon, A., Sudarshan, S., and Roy, P. Extending Query Rewriting Techniques for Fine-grained Access Control. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD), 2004.*

[35] Schoepe, D., Hedin, D., and Sabelfeld, A. SeLINQ: Tracking Information Across Application-database Boundaries. *SIGPLAN Not. – Volume 49,9, Aug 2014.*

[36] Schultz, D., and Liskov, B. IFDB: Decentralized Information Flow Control for Databases. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys), 2013.*

[37] Sweeney, L. K-anonymity: A model for protecting privacy. *International Journal of Uncertainty Fuzziness Knowledge-Based Systems – Volume 10, 5, 2002.*

[38] Toland, T. S., Frakas, C., and Eastman, C. M. The inference problem: Maintaining maximal availability in the presence of database updates. *Computers and Security 29*, 1.

[39] Turan, U., and Toroslu, I. H. Privacy preserving secure decomposition algorithm for attribute based access control mechanism. *CoRR abs/1402.5742* (2014).

[40] Upadhyaya, P., Balazinska, M., and Suciu, D. Automatic Enforcement of Data Use Policies with DataLawyer. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD), 2015.*

[41] Yang, J., Hance, T., Austin, T. H., Solar-Lezama, A., Flanagan, C., and Chong, S. Precise, dynamic information flow for database-backed applications. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2016.*

[42] Yip, A., Wang, X., Zeldovich, N., and Kaashoek, M. F. Improving application security with data flow assertions. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP), 2009.*