Blockaid: Data Access Policy Enforcement for Web Applications (Extended Technical Report)

Wen Zhang[1] Eric Sheng[2,*] Michael Chang[1] Aurojit Panda[3] Mooly Sagiv[4] Scott Shenker[1,5]

[1]*UC Berkeley* [2]*Yugabyte* [3]*NYU* [4]*Tel Aviv University* [5]*ICSI*

Abstract

Modern web applications serve large amounts of sensitive user data, access to which is typically governed by data-access policies. Enforcing such policies is crucial to preventing improper data access, and prior work has proposed many enforcement mechanisms. However, these prior methods either alter application semantics or require adopting a new programming model; the former can result in unexpected application behavior, while the latter cannot be used with existing web frameworks.

Blockaid is an access-policy enforcement system that preserves application semantics and is compatible with existing web frameworks. It intercepts database queries from the application, attempts to verify that each query is policy-compliant, and blocks queries that are not. It verifies policy compliance using SMT solvers and generalizes and caches previous compliance decisions for better performance. We show that Blockaid supports existing web applications while requiring minimal code changes and adding only modest overheads.

1 Introduction

Many modern web applications use relational databases to store sensitive user data, access to which is governed by organizational or regulatory *data-access policies*. To enforce these policies, today's web developers wrap each database query within access checks that determine whether a user has access to the queried data. As an application can query the database at many call sites, getting access checks right at every call site is challenging, and erroneous or missing checks have exposed sensitive data in many production systems [4, 30, 37, 38, 47, 65].

Prior work has suggested a variety of languages, frameworks, and tools that simplify the enforcement of data-access policies. As we detail in §2, these approaches either (1) require applications be written using specialized web frameworks, hindering their adoption; or (2) transparently remove from query results any data that cannot be revealed, possibly resulting in unexpected application behavior (e.g., the user has no idea that there are missing results and reaches the wrong conclusion).

This paper proposes an alternative approach to enforcing data-access policies that meets four goals:

1. **Backwards compatibility**: Applies to applications built using common existing web frameworks.
2. **Semantic transparency**: Fully answers queries that comply with the policy and blocks queries that do not (rather than providing partial, and potentially misleading, results).
3. **Policy expressiveness**: Supports a wide range of policies.
4. **Low overhead**: Has limited impact on page load time.

We implement this approach in Blockaid, a system that enforces a data-access policy at runtime by intercepting SQL queries issued by the application, verifying that they comply with the policy, and blocking those that do not. We assume non-compliant queries are rare in production (having been mostly eliminated in testing), and focus on efficiently checking compliant queries. Blockaid expects the developer to insert access checks as usual; it merely ensures that the checks are adequate.

A Blockaid policy consists of SQL view definitions that specify what information can be accessed by a given user, although the application still issues queries against the base tables as usual (rather than against the views). Under this setting, a query is compliant if it *never* reveals—for any underlying dataset—more information than the views do, a well-studied property in databases called *query determinacy* [51].

While determinacy characterizes the compliance of one query in isolation, it is too restrictive in the context of web applications, which typically issue multiple queries when serving a request. In this setting, what queries can be allowed often depends on the result of previous queries in the same web request. Thus, we extend determinacy to take a *trace* of previous queries and their responses, a novel extension we call *trace determinacy*, and use that as the criterion for compliance.

To verify compliance, Blockaid frames trace determinacy as an SMT formula and checks it using SMT solvers. As we later explain, a solver returns an unsatisfiability proof when a query is compliant, and a test demonstrating a violation otherwise.

This basic method, while correct, is impractically slow as it invokes solvers on every query. Thus, we use a *decision cache* to record compliant queries (with traces) so that future

[*]Work done while at UC Berkeley.

occurrences need not be rechecked. But caching *exact* queries and traces would be ineffective: a query is usually specific to the user and page visited, and so is unlikely to occur many times.

Thus, to increase cache hit rate, we implement a novel generalization mechanism which, given a compliant query-trace pair, extracts a small set of assumptions on the query and trace that alone would guarantee compliance. These assumptions are cached in the form of a *decision template*, which will apply to all future query-trace pairs that meet those assumptions. Blockaid generates decision templates by progressively relaxing a query and trace while maintaining compliance, with the help of solver-generated unsat cores [8, § 11.8]. It does not cache noncompliance results, which we expect to be rare in production as they typically indicate application/policy bugs.

We applied Blockaid to three existing applications—diaspora* [25], Spree [64], and Autolab [5]—and found that it imposes an overhead of 2 % to 12 % to the median page load time when compliance decisions are cached.

Blockaid has some important limitations. It assumes that the application obtains all of its information through SQL queries visible to Blockaid or from a caching layer or file system mediated by Blockaid. It also supports only a subset of SQL and is at the mercy of solver performance and unsat-core size.

Blockaid is open source at `https://github.com/blockaid-project`.

2 Related Work

The subject of data-access control has been studied by many. We compare our approach to prior ones along our goals (§1).

Static verification. Several systems have been proposed to statically verify that application code can only issue compliant queries; examples include Swift [17], SELINKS [22], UrFlow [15], and STORM [42]. These systems incur no run-time overhead and can be more precise than Blockaid as they analyze source code. However, they typically require using a specialized language or framework like Jif [50] or Ur/Web [16], sacrificing compatibility with common web frameworks.

Query modification. A popular run-time approach is query modification [66]: replacing secret values returned by a query with placeholders (or dropping any rows containing secrets). This is implemented in commercial databases [13, 49] and academic works like Hippocratic databases [3], Jacqueline [73], Qapla [48], and multiverse databases [46]. While this approach allows programmers to issue queries without regard to policies, it lacks semantic transparency as it can alter query semantics in unexpected ways and return misleading results [32, 59, 70].

Furthermore, many of the query modification mechanisms use row- and cell-level policies (e.g., SQL Server RLS and DDM, Oracle VPD). As we discuss in §9, this row/cell-level format is less expressive than Blockaid's view-based scheme.

View-based access control. Many databases allow creating views and granting access to views and tables. Although identical in expressiveness to Blockaid, this mechanism requires queries to explicitly use view names instead of table names (like `Users`). This marks a significant deviation from regular web programming, as programmers must now sort out which views to use for each query. In contrast, Blockaid allows queries to be issued against the base tables directly.

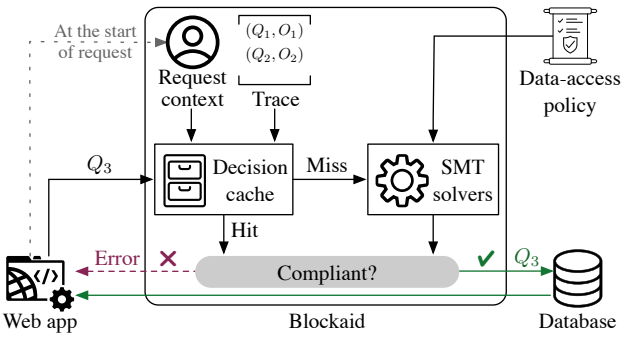


Figure 1: *An overview of Blockaid (for a single web request).*

While some prior work has studied view-based compliance of queries issued against base tables [10, 11], they only check single queries, while Blockaid checks a query in the context of a trace, a crucial feature for supporting web applications.

3 System Design

3.1 Application Assumptions and Threat Model

Blockaid targets web applications that store data in a SQL database. We assume that a user is logged in and that the current user's identifier is stored in a *request context*. The application can access the database and the request context when serving a request; each request is handled independently from others. We assume that the application authenticates the user correctly, and that the correct request context is passed to Blockaid (§3.2).

A *data-access policy* dictates, for a given request context, what information in the database is *accessible* and what is *inaccessible*. We treat the database schema and the policy itself as public knowledge and assume that the user cannot use side channels to circumvent policies. We enforce policies on database *reads* only, as done in prior work [2, 10–12, 33, 41, 46, 59, 62, 66, 70]. Ensuring the integrity of updates, while important, is orthogonal to our goal and is left to future work.

3.2 System Overview

Blockaid is a SQL proxy that sits between the application and the database (Figure 1). It takes as input (1) a database schema (including constraints), and (2) a data-access policy specified as database views (§4), and checks query compliance for each web request separately. For each web request, it maintains a *trace* of queries issued so far and their results; the trace is cleared when the request ends. Blockaid assumes that the results returned by queries in the trace are not altered till the end of the request.

When a web request starts, the application sends its request context to Blockaid. Then, every SQL query from the application traverses Blockaid, which attempts to verify that the query is *compliant*—i.e., it can be answered using accessible informa-

tion only. To do so, Blockaid checks the decision cache for any similar query has been determined compliant previously. If not, it encodes noncompliance as an SMT formula (§5) and checks its satisfiability using several SMT solvers in parallel (§7).

If a query is compliant, Blockaid forwards it to the database unmodified. In case of a cache miss, Blockaid also extracts and caches a decision template (§6). Finally, it appends the query and its result to the trace. If verification fails, Blockaid blocks the query by raising an error to the application.

Although our core design assumes that all sensitive information is stored in the relational database, Blockaid supports limited compliance checking for two other common data sources:

1. If the application stores database-derived data in a **caching layer** (e.g., Redis), the programmer can annotate a cache key pattern with SQL queries from which the value can be derived. Blockaid can then intercept each cache read and verify the compliance of the queries associated with the key.
2. If the application stores sensitive data in the **file system**, it can generate hard-to-guess names for these files and store the file names in a database column protected by the policy.

Blockaid's basic requirement is soundness: preventing the revelation of inaccessible information (formalized in §4.3). However, it may reject certain behaviors that do not violate the policy (§9), although such false rejections never arose in our evaluation (§8).

We end by emphasizing two aspects of Blockaid's operation:

1. Blockaid has *no visibility into or control over* the application (except by blocking queries). So it must assume that *any* data fetched by the application will be shown to the user.
2. Blockaid has *no access* to the database except by observing query results—it cannot issue additional queries of its own.

3.3 Application Requirements

For use with Blockaid, an application must:

1. Send the request context to Blockaid at the start of a request and signal Blockaid to clear the trace at the end;
2. Handle rejected queries cleanly (although a web server's default behavior of returning HTTP 500 often suffices); and,
3. Not query data that it does not plan on revealing to the user.

Existing applications often violate the third requirement. For example, when a user views an order on a Spree e-commerce site, the order is fetched from the database, and only then does Spree check, in application code, that the user is allowed to view it. To avoid spurious errors from Blockaid, such applications must be modified to fetch only data known to be accessible.

4 View-based Policy and Compliance

Throughout the paper, we will use as a running example a calendar application with the following database schema:

$$\begin{aligned} &\mathit{Users}(\underline{\mathit{UId}}, \mathit{Name}) \\ &\mathit{Events}(\underline{\mathit{EId}}, \mathit{Title}, \mathit{Duration}) \\ &\mathit{Attendances}(\underline{\mathit{UId}, \mathit{EId}}, \mathit{ConfirmedAt}) \end{aligned}$$

Listing 1: *Example policy view definitions V_1 to V_4 for the calendar application.* `?MyUId` *refers to the current user ID.*

1. `SELECT * FROM Users`
   Each user can view the information on all users.
2. `SELECT * FROM Attendances WHERE UId = ?MyUId`
   Each user can view their own attendance information.
3. `SELECT * FROM Events WHERE EId IN (SELECT EId FROM Attendances WHERE UId = ?MyUId)`
   Each user can view the information on events they attend.
4. `SELECT * FROM Attendances WHERE EId IN (SELECT EId FROM Attendances WHERE UId = ?MyUId)`
   Each user can view all attendees of the events they attend.

where primary keys are underlined. The request context consists of a parameter *MyUId* denoting the *UId* of the current user.

4.1 Specifying Policies as Views

A policy is a collection of SQL queries that, together, define what information a user is allowed to access. Each query is called a *view definition* and can refer to parameters from the request context. As an example, Listing 1 shows four view definitions, V_1–V_4; we denote this policy as $\mathcal{V} = \{V_1, V_2, V_3, V_4\}$.

Notationally, for a view V and a request context *ctx*, we write V^{ctx} to denote V with its parameters replaced with values in *ctx*. We often drop the superscript when the context is apparent.

4.2 Compliance to View-based Policy

Under a policy consisting of view definitions, Blockaid can allow an application query to go through *only if* it is certain that the query's result is *uniquely determined* by the views. In other words, an allowable query must be answerable using accessible information alone. If a query's output *might* depend on information outside the views, Blockaid must block the query.

Example 4.1. Let $MyUId = 2$. The following query selects the names of everyone whom the user attends an event with:

```
SELECT DISTINCT u.Name
FROM   Users u
       JOIN Attendances a_other
         ON a_other.UId = u.UId
       JOIN Attendances a_me
         ON a_me.EId = a_other.EId
WHERE a_me.UId = 2
```

Looking at Listing 1, this query can always be answered by combining V_4, which reveals the *UId* of everyone whom the user attends an event with, with V_1, which supplies the names associated with these *UId*'s. Hence, Blockaid allows it through.

This query above is allowed *unconditionally* because it is answerable using the views on *any* database instance. More commonly, queries are allowed *conditionally* based on what Blockaid has learned about the current database state, given the trace of prior queries and results in the same web request.

Example 4.2. Again, let $MyUId = 2$. Consider the following sequence of queries issued while handling one web request:

1. `SELECT * FROM Attendances WHERE UId = 2 AND EId = 5`
   `↪ (UId=2, EId=5, ConfirmedAt="05/04 1pm")`
2. `SELECT Title FROM Events WHERE EId = 5`

The application first queries the user's attendance record for Event #5—an unconditionally allowed query—and receives one row, indicating the user is an attendee. It then queries the title of said event. This is allowed because V_3 reveals the information on all events attended by the user. More precisely, the trace limits our scope to only databases where the user attends Event #5. Because the second query is answerable using V_3 *on all such databases*, it is conditionally allowed given the trace.

Context is important here: the second query cannot be safely allowed if it were issued in isolation.

Example 4.3. Suppose instead that the application issues the following query by itself:

```
SELECT Title FROM Events WHERE EId = 5
```

Blockaid must block this query because it is not answerable using $\mathcal{V}$ on a database where the user does not attend Event #5. Whether or not the user *actually* is an attendee of the event is irrelevant: The application, not having queried the user's attendance records, cannot be certain that the query is answerable using accessible information alone. This differs from alternative security definitions [32, 39, 74] where a policy enforcer can allow a query after inspecting additional information in the database that has not been fetched by the application.

Definition 4.4. A *trace* $\mathcal{T}$ is a sequence $(Q_1, O_1), \ldots, (Q_n, O_n)$ where each Q_i is a query and each O_i is a collection of tuples.

Such a trace denotes that the application has issued queries $Q_1, \ldots, Q_n$ and received results $O_1, \ldots, O_n$ from the database.

We now motivate the formal definition of query compliance given a trace (using colors to show correspondence between text and equations). Consider any two databases that are:

- Equivalent in terms of accessible data (i.e., they differ only in information outside the views), and
- Consistent with the observed trace (i.e., we consider only databases that *could* be the one the application is querying).

Blockaid must ensure that such two databases are indistinguishable to the user—by allowing only queries that produce the same result on both databases.

Definition 4.5. Let ctx be a request context, $\mathcal{V}$ be a set of views, and $\mathcal{T} = \{(Q_i, O_i)\}_{i=1}^{n}$ be a trace. A query Q is *ctx-compliant* to $\mathcal{V}$ given $\mathcal{T}$ if for every pair of databases D_1, D_2 that conform to the database schema and constraints,[1] and satisfy:

$$V^{ctx}(D_1) = V^{ctx}(D_2), \qquad (\forall V \in \mathcal{V}) \tag{1}$$
$$Q_i(D_1) = O_i, \qquad (\forall 1 \le i \le n) \tag{2}$$
$$Q_i(D_2) = O_i, \qquad (\forall 1 \le i \le n) \tag{3}$$

we have $Q(D_1) = Q(D_2)$. We will simply say *compliant* if the context is clear.

We call Definition 4.5 *trace determinacy* because it extends the classic notion of query determinacy [51, 61] with the trace. Query determinacy is undecidable even for conjunctive views and queries [27, 28]; trace determinacy must also be undecidable in the same scenario. Although several decidable cases have been discovered for query determinacy [1, 51, 53], they are not expressive enough for our use case. A promising direction is to identify classes of views and queries that capture common web use cases and for which trace determinacy is decidable.

4.3 From Query Compliance to Noninterference

Blockaid's end goal is to ensure that an application's output depends only on information accessible to the user. In relation to this goal, query compliance (Definition 4.5) satisfies two properties, making it the right criterion for Blockaid to enforce:

1. **Sufficiency**: As long as *only compliant queries* from the application are let through, there is no way for an execution's outcome to be influenced by inaccessible information.
2. **Necessity**: Any enforcement system that makes per-query decisions based solely on the query and its preceding trace *cannot safely allow any non-compliant query* without the risk of the application revealing inaccessible information.

Before stating and proving these properties formally, let us first model our target applications, enforcement systems, and goals.

We model a web request handler as a program $\mathcal{P}(ctx, req, D)$ that maps a request context ctx, an HTTP request req, and a database D to an HTTP response.[2] A program that abides by a policy $\mathcal{V}$ satisfies a *noninterference* property [21, 29] stating that its output depends only on the inputs that the user has access to—namely, ctx, req, and $V^{ctx}(D)$ for each $V \in \mathcal{V}$. The formal definition follows from a similar intuition as Definition 4.5.

Definition 4.6. A program $\mathcal{P}$ satisfies *noninterference* under policy $\mathcal{V}$ if the following condition holds:

$$\mathrm{NI}_{\mathcal{V}}(\mathcal{P}) := \forall ctx, req, D_1, D_2.\ \left[\forall V \in \mathcal{V}.\, V^{ctx}(D_1) = V^{ctx}(D_2)\right] \implies \mathcal{P}(ctx, req, D_1) = \mathcal{P}(ctx, req, D_2).$$

An enforcement system must ensure that any program running under it satisfies noninterference. We now model such a system that operates under Blockaid's assumptions.

[1] We will henceforth use "schema" to mean both schema and constraints, and rely on the database and/or the web framework to enforce the constraints.

[2] For simplicity, we assume $\mathcal{P}$ is a pure function—deterministic, terminating, and side-effect free—although this assumption can be relaxed through standard means from information-flow control [34, § 2].

Definition 4.7. An *enforcement predicate* is a mapping from a request context, a query, and a trace to an allow/block decision:

$$E(ctx, Q, \mathcal{T}) \to \{\checkmark, \times\}.$$

Definition 4.8. Let $\mathcal{P}(ctx, req, D)$ be a program and E be an enforcement predicate. We define the program $\mathcal{P}$ *under enforcement using* E as a new program $\mathcal{P}^E(ctx, req, D)$ that simulates every step taken by the original program $\mathcal{P}$, except that it maintains a trace $\mathcal{T}$ and blocks any query Q issued by $\mathcal{P}$ where $E(ctx, Q, \mathcal{T}) = \times$ by immediately returning an error.

Note that $\mathcal{P}^E$ evaluates E only on traces in which every query has been previously allowed by E given its trace prefix.

Definition 4.9. Given a request context ctx, we say that a trace $\mathcal{T} = \{(Q_i, O_i)\}_{i=1}^n$ is *prefix E-allowed* if for all $1 \le i \le n$,

$$E(ctx, Q_i, \mathcal{T}[1..i-1]) = \checkmark.$$

Definition 4.10. A predicate E *correctly enforces* policy $\mathcal{V}$ if:

$$\forall \mathcal{P}.\ \mathrm{NI}_{\mathcal{V}}(\mathcal{P}^E).$$

We are ready to state the sufficiency-and-necessity theorem, whose proof is left to Appendix B. Like before, we use colors to link a statement to its explanation.

Theorem 4.11. *Let $\mathcal{V}$ be a set of views and E be a predicate.*

1. *Suppose $E(ctx, Q, \mathcal{T}) = \checkmark$ only when Q is ctx-compliant to $\mathcal{V}$ given $\mathcal{T}$. Then E correctly enforces $\mathcal{V}$.*
2. *Suppose E correctly enforces $\mathcal{V}$. Then for any request context ctx, query Q, and prefix E-allowed trace $\mathcal{T}$ such that $E(ctx, Q, \mathcal{T}) = \checkmark$, Q is ctx-compliant to $\mathcal{V}$ given $\mathcal{T}$.*

To unpack, Theorem 4.11 says: (1) as long as an enforcement predicate ensures query compliance, it correctly enforces the policy on applications (i.e., sufficiency); and (2) for a predicate to correctly enforce the policy, it must ensure query compliance (i.e., necessity). Thus, query compliance can be regarded as the "projection" of application noninterference onto Blockaid's lens, making it the ideal criterion to enforce.

5 Compliance Checking with SMT

Having defined view-based policy and compliance, we now introduce how Blockaid verifies compliance using SMT solvers.

5.1 Translating Noncompliance to SMT

Blockaid verifies query compliance by framing *noncompliance* (i.e, the negation of Definition 4.5) as an SMT formula and checking its satisfiability—a query is compliant if and only if the formula is *unsatisfiable*. We use a straightforward translation based on Codd's theorem [20], which states, informally, that relational algebra under set semantics is equivalent in expressiveness to first-order logic (FOL). Relational algebra has five operators—projection, selection, cross product, union, and difference—and tables are interpreted as *sets* of rows (i.e., no duplicates). Under this equivalence, tables are translated to predicates in FOL, and operators are implemented using existential quantifiers, conjunctions, disjunctions, and negations.

Example 5.1. Let us translate into FOL the following query Q executed on a database D:

```
SELECT e.EId, e.Title
FROM   Events e, Attendances a
WHERE  e.EId = a.EId AND a.UId = 2
```

Let $E^D(\cdot,\cdot,\cdot)$ and $A^D(\cdot,\cdot,\cdot)$ be FOL predicates representing the *Events* and *Attendances* table in the database D in:

$$\mathsf{Q}^D(\mathsf{x}_e, \mathsf{x}_t) := \exists x_d, x_u, x'_e, x_c.\, E^D(\mathsf{x}_e, \mathsf{x}_t, x_d) \wedge A^D(x_u, x'_e, x_c) \wedge \mathsf{x}_e = x'_e \wedge x_u = 2.$$

$\mathsf{Q}^D(\mathsf{x}_e, \mathsf{x}_t)$ encodes the statement $(\mathsf{x}_e, \mathsf{x}_t) \in Q(D)$, i.e., that the row $(\mathsf{x}_e, \mathsf{x}_t)$ is returned by Q on database D. Note that Q^D is not a logical symbol, but merely a shorthand for the right-hand side.

Example 5.2. We now present the noncompliance formula for a single query Q with respect to $\mathcal{V}$ from §4.1. Let $\mathsf{V}_1^{D_i}, \ldots, \mathsf{V}_4^{D_i}$ and Q^{D_i} encode the views and query on database D_i ($i = 1, 2$) in FOL. The desired formula would then be the conjunction of:

$$\begin{aligned}
&\forall \mathbf{x}.\, \mathsf{V}_1^{D_1}(\mathbf{x}) \leftrightarrow \mathsf{V}_1^{D_2}(\mathbf{x}), && (V_1(D_1) = V_1(D_2)) \\
&\qquad\vdots \\
&\forall \mathbf{x}.\, \mathsf{V}_4^{D_1}(\mathbf{x}) \leftrightarrow \mathsf{V}_4^{D_2}(\mathbf{x}), && (V_4(D_1) = V_4(D_2)) \\
&\exists \mathbf{x}.\, \mathsf{Q}^{D_1}(\mathbf{x}) \not\leftrightarrow \mathsf{Q}^{D_2}(\mathbf{x}), && (Q(D_1) \neq Q(D_2))
\end{aligned}$$

where $\mathbf{x}$ denotes a sequence of fresh variables. Database constraints and consistency with a trace can be encoded similarly.

5.2 Handling Practical SQL Queries

The encoding of relational algebra into logic, while straightforward, fails to cover real-world SQL due to two semantic gaps:

1. While the encoding assumes that relational algebra is evaluated under *set semantics*, in practice databases use a mix of set, bag, and other semantics when evaluating queries.[3]
2. SQL operations like aggregation and sorting have no corresponding operators in relational algebra.

For Blockaid to bridge these gaps, it must first assume that database tables contain no duplicate rows. This is generally the case for web applications as object-relational mapping libraries like Active Record [60] and Django [26] add a primary key for every table. Given this assumption, Blockaid rewrites complex SQL into *basic queries* that map directly to relational algebra.

5.2.1 Basic SQL Queries

Definition 5.3. A *basic* query is either a `SELECT-FROM-WHERE` query that never returns duplicate rows, or a `UNION` of `SELECT-FROM-WHERE` clauses (the `UNION` always removes duplicates).[4]

A basic query on duplicate-free tables maps to relational algebra under set semantics, and so can be directly translated to FOL. To ensure a `SELECT` query is basic, we check it against these sufficient conditions for returning no duplicate rows:

[3] For example, a SQL `SELECT` clause can return duplicate rows, but the `UNION` operator removes duplicates.

[4] The `MINUS` operator is not used in our applications and is omitted.

- It contains the `DISTINCT` keyword or ends in `LIMIT 1`; or
- It projects unique key column(s) from every table in `FROM`, e.g., **`SELECT`** `UId, Name` **`FROM`** `Users`; or
- It is constrained by uniqueness in its `WHERE` clause—e.g.:

```
SELECT e.EId
FROM   Events e, Attendances a
WHERE  e.EId = a.EId AND a.UId = 2
```

For this query to return multiple copies of x, the database must contain multiple rows of the form $Attendances(2, x, ?)$; this is ruled out by the uniqueness constraint on (UId, EId).

In our experience, policy views can typically be written as basic queries directly—e.g., for Listing 1 we can frame V_3 and V_4 as equivalent basic queries by replacing subqueries with joins and using the inner join transformation from §5.2.2.

5.2.2 Rewriting Into Basic Queries

When the application issues a query Q, Blockaid attempts to rewrite it into a basic query Q' and verify its compliance instead. Ideally, Q' would be equivalent to Q, but when this is not possible, Blockaid produces an *approximate* Q' that reveals *at least as much* information as Q does.[5] Such approximation preserves soundness but may sacrifice completeness, although it caused no false rejections in our evaluation. We now explain how to rewrite several types of queries encountered in practice.

Inner joins. A query of the form:

```
SELECT ... FROM R1
INNER JOIN R2 ON C1 WHERE C2
```

is equivalently rewritten as the basic query:

```
SELECT ... FROM R1, R2 WHERE C1 AND C2
```

Left joins on a foreign key. Consider a query of the form:

```
SELECT ... FROM R1
LEFT JOIN R2 ON R1.A = R2.B WHERE ...
```

If `R1.A` is a foreign key into `R2.B`, then every row in `R1` matches at least one row in `R2`. In this case, the left join can be equivalently written as an inner join, which is handled as above.

Order-by and limit. Blockaid adds any `ORDER BY` column as an output column and then discards the `ORDER BY` clause. It also discards any `LIMIT` clause but, when adding the query to the trace, uses a modified condition $O_i \subseteq D(Q_i)$ (instead of "=") to indicate that it may have observed a partial result.

Aggregations. Blockaid turns **`SELECT SUM`**`(A)` **`FROM`** `R` into **`SELECT`** `PK, A` **`FROM`** `R`, where `PK` is table `R`'s primary key. By projecting the primary key in addition to `A`, the rewritten query reveals the multiplicity of the values in `A`—necessary for computing **`SUM`**`(A)`—without returning duplicate rows.

Left joins that project one table. Left joins of the form:

```
SELECT DISTINCT A.* FROM A
LEFT JOIN B ON C1 WHERE C2
```

can be equivalently rewritten to the basic query:

```
(SELECT A.* FROM A
 INNER JOIN B ON C1 WHERE C2)
UNION
(SELECT * FROM A WHERE C3)
```

where `C3` is obtained by replacing each occurrence of `B.?` with `NULL` in `C2` and simplifying the resulting predicate.[6] The first subquery covers the rows in `A` with at least one match in `B`, and the second subquery covers those with no matches.

Feature not supported. The SQL features not supported include `GROUP BY`, `ANY`, `EXISTS`, etc., although they can also be formulated / approximated using basic queries. In the future we plan to leverage other formalisms [14, 18, 19, 67–69, 71] to model complex SQL semantics more precisely.

5.3 Optimizations and SMT Encoding

We end this section with several optimizations for compliance checking and some notes on the SMT encoding.

Strong compliance. We define a stronger notion of compliance, which we found SMT solvers can verify more efficiently.

Definition 5.4. A query Q is *strongly ctx-compliant* to policy $\mathcal{V}$ given trace $\{(Q_i, O_i)\}_{i=1}^{n}$ if for each pair of databases D_1, D_2 that conform to the schema and satisfy:

$$V^{ctx}(D_1) \subseteq V^{ctx}(D_2), \qquad (\forall V \in \mathcal{V}) \tag{4}$$

$$Q_i(D_1) \supseteq O_i, \qquad (\forall 1 \le i \le n) \tag{5}$$

we have $Q(D_1) \subseteq Q(D_2)$.

Theorem 5.5. *If Q is strongly compliant to $\mathcal{V}$ given trace $\mathcal{T}$, then Q is also compliant to $\mathcal{V}$ given $\mathcal{T}$.*

Proof. Let Q be strongly compliant to $\mathcal{V}$ given $\mathcal{T}$. To show that Q is also compliant, let D_1, D_2 be databases that satisfy Equations (1) to (3) from the compliance definition. These imply the strong compliance assumptions (Equations (4) and (5)), and so we have $Q(D_1) \subseteq Q(D_2)$. By symmetry, we also have $Q(D_2) \subseteq Q(D_1)$. Putting the two together, we conclude $Q(D_1) = Q(D_2)$, showing Q to be compliant to $\mathcal{V}$ given $\mathcal{T}$. □

For faster checking, Blockaid verifies strong compliance rather than compliance; by Theorem 5.5, soundness is preserved. However, there are scenarios where a query is compliant but *not* strongly compliant (see Appendix C); such queries will be falsely rejected by Blockaid. This did not pose a problem in practice as we found the two notions to coincide for every query encountered in our evaluation.

Fast accept. Given a view **`SELECT`** `C1, ..., Ck` **`FROM`** `R`, any query that references only columns `R.C1, ..., R.Ck` must be compliant and is accepted without SMT solving.

[5] It suffices to guarantee that Q can be computed from the result of Q'.

[6] As long as `C2` contains no negations, it is safe to treat a `NULL` literal as `FALSE` when propagating through or short-circuiting `AND` and `OR` operators.

Trace pruning. Queries that returns many rows can inflate the trace and slow down the solvers. Fortunately, often times only few of the rows matter to a later query's compliance. We thus adopt a trace-pruning heuristic: when checking a query Q, look for any previous query has returned over ten rows, and keep only those rows that contain the first occurrence of a primary-key value (e.g., user ID) appearing in Q. This heuristic is sound, but may need to be adapted for any application where our premise for pruning does not hold.

SQL types and predicates. To model SQL types, we use SMT's uninterpreted sorts, which we found to yield better performance than theories of integers, strings, etc. We support logical operators `AND` and `OR`, comparison operators <, <=, >, >=, and operators `IN`, `NOT IN`,[7] `IS NULL`, and `IS NOT NULL`. We model < as an uninterpreted relation with a transitivity axiom.

NULLs. We model `NULL` using a two-valued semantics of SQL [31, § 6] by (1) designating a constant in each sort as `NULL`, and (2) taking `NULL` into account when implementing SQL operators. For example, the SQL predicate `x=y` translates into the following SMT formula: $x = y \wedge x \neq null \wedge y \neq null$.

6 Decision Generalization and Caching

While SMT solvers can check a wide range of queries, doing so often takes 100s of milliseconds per query. As a page load can depend on tens of queries, this overhead can add up to *seconds*.

To alleviate this overhead, Blockaid aims to reduce solver calls by caching compliance decisions. Naively, once query Q is deemed compliant given trace $\mathcal{T}$, we could record $(Q, \mathcal{T})$ and allow future occurrences without re-invoking the solvers.

However, this proposal is unlikely to be effective because the number of distinct $(Q, \mathcal{T})$ pairs can be unbounded. For example, an application can issue as many queries of the form `SELECT * FROM Users WHERE UId = ?` as there are users in the system. Therefore, requiring an exact query-trace match for a cache hit would result in a low cache hit rate.

Fortunately, while an application can issue an unbounded number of distinct queries, it only exhibits a finite number of truly different behaviors. For example, the query sequences generated by requests for two different calendar events are likely identical in structure while differing only in parameters (e.g., event ID). If one sequence is compliant, we can *generalize* this knowledge to conclude that the other is also compliant.

This generalization problem is the central challenge we tackle in this section: Given a query's compliance with respect to a trace, how to abstract this knowledge into a *decision template* such that (1) any query (and its trace) that matches this template is compliant, and (2) the template is general enough to produce matches on similar requests. Such a template, once cached, will apply to an entire class of traces and queries.

Decision templates are designed to cache compliant queries only. Our techniques do not extend to non-compliant queries, which are expected to be rare in production as they typically indicate bugs in the application or the policy.

Let us start with an example of a decision template.

6.1 Example

Suppose a user with $UId = 1$ requests Event #42 in the calendar application, resulting in the application issuing a sequence of SQL queries. Consider the third query, shown in Listing 2a. As we explained in Example 4.2, Query #3 is compliant because Query #2 has established that the user attends the event.

Blockaid aims to abstract this query (with trace) into a decision template that applies to another user viewing a different event. Listing 2b shows such a template; the notation says: If each query-output pair above the line has a match in a trace $\mathcal{T}$, then any query of the form below the line is compliant given $\mathcal{T}$. This particular template states: after it is determined that user x attends event y, user x can view event y for any x and y.

Compared with the concrete query and trace, this template (1) omits Query #1, which is immaterial to the compliance decision; and (2) replaces the concrete values with parameters. Occurrences of `?0` here constrain the event ID fetched by the query to equal the previously checked event ID. We use `*` to denote a fresh parameter, i.e., any arbitrary value is allowed.

We now dive into how Blockaid extracts such a decision template from a concrete query and trace. But before we do so, let us first define what a decision template is, what it means for a template to have a match, and what makes a "good" template.

6.2 Definitions and Goals

For convenience, from now on we will denote a trace as a set of query-*tuple* pairs $\{(Q_i, t_i)\}_{i=1}^{n}$, where each t_i is *one* of the rows returned by Q_i. A query that returns multiple rows is represented as multiple such pairs. This change of notation is permissible because under strong compliance (Definition 5.4), we no longer take into account the *absence* of a returned row.

Definition 6.1. We say a trace $\mathcal{T} = \{(Q_i, t_i)\}_{i=1}^{n}$ is *feasible* if there exists a database D such that $t_i \in Q_i(D)$ for all $1 \leq i \leq n$.

Definition 6.2. A *decision template* $\mathcal{D}[\mathbf{x}, \mathbf{c}]$, where $\mathbf{c}$ denotes variables from the request context and $\mathbf{x}$ a sequence of variables disjoint from $\mathbf{c}$, is a triple $(Q_\mathcal{D}, \mathcal{T}_\mathcal{D}, \Phi_\mathcal{D})$ where:

- $Q_\mathcal{D}$ is the *parameterized query*, whose definition can refer to variables from $\mathbf{x} \cup \mathbf{c}$;
- $\mathcal{T}_\mathcal{D}$ is the *parameterized trace*, whose queries and tuples can refer to variables from $\mathbf{x} \cup \mathbf{c}$; and
- $\Phi_\mathcal{D}$, the *condition*, is a predicate over $\mathbf{x} \cup \mathbf{c}$.

We will often denote a template simply by $\mathcal{D}$ if the variables are either unimportant or clear from the context.

As we later explain, $\Phi_\mathcal{D}$ represents any extra constraints that a template imposes on its variables (e.g., `?0` < `?1`).

Definition 6.3. A *valuation* ν over a collection of variables $\mathbf{y}$ is a mapping from $\mathbf{y}$ to constants (including `NULL`), extended to objects that contain variables in $\mathbf{y}$. For example, given a parameterized query Q, $\nu(Q)$ denotes Q with each occurrence of variable $y \in \mathbf{y}$ substituted with $\nu(y)$.

[7] We only support `IN` and `NOT IN` with a list of values, not with a subquery.

Listing 2: *An example query with trace from the calendar application and a decision template generated from it.*

(a) *Example query with trace ($UId = 1$).*

```
1. SELECT * FROM Users WHERE UId = 1
   ↪ (UId=1, Name="John Doe")
2. SELECT * FROM Attendances
   WHERE UId = 1 AND EId = 42
   ↪ (UId=1, EId=42, ConfirmedAt="05/04 1pm")
---
3. SELECT * FROM Events WHERE EId = 42
```

(b) *The decision template generated by Blockaid.*

```
1. SELECT * FROM Attendances
   WHERE UId = ?MyUId AND EId = ?0
   ↪ (UId = ?MyUId, EId = ?0, ConfirmedAt = *)
---
✓ SELECT * FROM Events WHERE EId = ?0
```

Definition 6.4. Let $\mathcal{D}[\mathbf{x},\mathbf{c}] = (Q_\mathcal{D}, \mathcal{T}_\mathcal{D}, \Phi_\mathcal{D})$ be a decision template, *ctx* be a request context, $\mathcal{T}$ be a trace, and Q be a query. We say that $\mathcal{D}$ *matches* $(Q,\mathcal{T})$ *under ctx* if there exists a valuation ν over $\mathbf{x}\cup\mathbf{c}$ such that:

- $\nu(\mathbf{c}) = ctx$,
- $\nu(Q_\mathcal{D}) = Q$,
- $(\nu(Q_j), \nu(t_j)) \in \mathcal{T}$ for all $(Q_j, t_j) \in \mathcal{T}_\mathcal{D}$, and
- $\nu(\Phi_\mathcal{D})$ holds.

Example 6.5. Listing 2b can be seen as a stylized rendition of a decision template $\mathcal{D}[\mathbf{x},\mathbf{c}]$ where $\mathbf{x} = (x_0, x_1)$—x_0 denoting `?0` and x_1 denoting the occurrence of `*`—and $\mathbf{c} = (MyUId)$; $Q_\mathcal{D}$ and $\mathcal{T}_\mathcal{D}$ are as shown below and above the line; and $\Phi_\mathcal{D}$ is the constant $\top$, meaning the template imposes no additional constraints on the variables.[8] Under the request context $MyUId = 1$, this template matches the query and trace in Listing 2a via the valuation $\{x_0 \mapsto 42, x_1 \mapsto$ `"05/04 1pm"`$, MyUId \mapsto 1\}$.

We are interested only in templates that imply compliance.

Definition 6.6. A decision template $\mathcal{D}$ is *sound* with respect to a policy $\mathcal{V}$ if for every request context *ctx*, whenever $\mathcal{D}$ matches $(Q,\mathcal{T})$ under *ctx*, Q is strongly *ctx*-compliant to $\mathcal{V}$ given $\mathcal{T}$.

Blockaid can verify that a template is sound via the following theorem derived from strong compliance (Definition 5.4):

Theorem 6.7. *A decision template $\mathcal{D}[\mathbf{x},\mathbf{c}] = (Q_\mathcal{D}, \mathcal{T}_\mathcal{D}, \Phi_\mathcal{D})$ is sound with respect to a policy $\mathcal{V}$ if and only if:*

$$\forall \mathbf{x},\mathbf{c},D_1,D_2.\quad \left.\begin{array}{r}\Phi_\mathcal{D}\\ \forall V\in\mathcal{V}.V(D_1)\subseteq V(D_2)\\ \forall (Q_i,t_i)\in\mathcal{T}_\mathcal{D}.t_i\in Q_i(D_1)\end{array}\right\} \implies Q_\mathcal{D}(D_1)\subseteq Q_\mathcal{D}(D_2).$$

For a compliant query Q (with trace $\mathcal{T}$) that misses the cache, there often exist many sound templates that match $(Q,\mathcal{T})$. But all such templates are not equal—we prefer the more *general* ones, those that match a wider range of *other* queries and traces.

Definition 6.8. A template $\mathcal{D}_1$ is *at least as general as* a template D_2 if for every query Q and feasible trace $\mathcal{T}$, if $\mathcal{D}_2$ matches $(Q,\mathcal{T})$, $\mathcal{D}_1$ also matches $(Q,\mathcal{T})$.

[8] Technically, this template requires $MyUId \neq$ `NULL` $\wedge\, x_0 \neq$ `NULL`. We omitted this condition in Listing 2b because we assume the user ID parameter and the *Attendances* table's *EId* column are both non-NULL.

Thus, Blockaid aims to generate a decision template that (1) is sound, (2) matches $(Q,\mathcal{T})$, and (3) is general enough for practical purposes. We now explain how this is achieved.

6.3 Generating Decision Templates

Blockaid starts from the trivial template $D_0 = (Q,\mathcal{T},\top)$, which is sound but not general, and generalizes it in two steps:

1. Minimize the trace $\mathcal{T}$ to retain only those (Q_i,t_i) pairs that are required for Q's compliance (§6.3.1).
2. Replace each constant in the trace and query with a fresh variable, and then generate a weak condition Φ over the variables that guarantees compliance (§6.3.3).

6.3.1 Step One: Trace Minimization

Blockaid begins by finding a minimal sub-trace of $\mathcal{T}$ that preserves compliance. It removes each $(Q_i,t_i)\in\mathcal{T}$ and, if Q is no longer compliant, adds the element back. For example, for Listing 2a this step removes Query #1. Denote the resulting minimal trace by $\mathcal{T}_\text{min}$ and let decision template $\mathcal{D}_1 = (Q,\mathcal{T}_\text{min},\top)$.

Proposition 6.9. *$\mathcal{D}_1$ is sound, matches $(Q,\mathcal{T})$, and is at least as general as $\mathcal{D}_0$.*

As an optimization, Blockaid starts the minimization from the sub-trace that the solver has actually used to prove compliance. It extracts this information from a solver-generated *unsat core* [8, § 11.8]—a subset of clauses in the formula that remains unsatisfiable even with all other clauses removed. If we attach *labels* to the clauses we care about, a solver will identify all labels in the unsat core when it proves the formula unsatisfiable.

To get an unsat core, Blockaid uses the following formula:

$$[LQ_i]\qquad \begin{array}{ll} V^{ctx}(D_1)\subseteq V^{ctx}(D_2), & (\forall V\in\mathcal{V})\\ t_i\in Q_i(D_1), & (\forall (Q_i,t_i)\in\mathcal{T})\\ Q(D_1)\not\subseteq Q(D_2), & \end{array}$$

where the clause asserting the i^{th} trace entry is labeled LQ_i. If Q is compliant, the solver returns as the unsat core a set S of labels. Blockaid ignores any $(Q_i,t_i)\in\mathcal{T}$ for which $LQ_i\notin S$.

6.3.2 Interlude: Model Finding for Satisfiable Formulas

A common operation in template generation is to remove parts of a formula and re-check satisfiability. A complication arises

when the formula turns satisfiable—while solvers are adept at proving unsatisfiability, they often fail on satisfiable formulas.[9]

To solve these formulas faster, we observe that they are typically satisfied by databases with small tables. We thus construct SMT formulas to directly seek such "small models" by representing each table not as an uninterpreted relation, but as a conditional table [35] whose size is bounded by a small constant.

A conditional table generalizes a regular table by (1) allowing variables in its entries, and (2) associating with each row with a *condition*, i.e., a Boolean predicate for whether the row exists. For example, a *Users* table with a bound of 2 appears as:

UId	*Name*	Exists?
$x_{u,1}$	$x_{n,1}$	b_1
$x_{u,2}$	$x_{n,2}$	b_2

where each entry and condition is a fresh variable, signifying that the table is not constrained in any way other than its size.

Queries on condition tables are evaluated via an extension of the relational algebra operators [35, § 7]. This allows queries to be encoded into SMT without using quantifiers or using relation symbols for tables.[10] For example, the query `SELECT Name FROM Users WHERE UId = 5` can be written as:

$$\mathsf{Q}(\mathsf{x}_n) := \bigvee_{i=1}^{2} (x_{u,i} = 5 \wedge x_{n,i} = \mathsf{x}_n \wedge b_i)\,.$$

We found that such formulas could be solved quickly by Z3.

After Blockaid generates an unsat core as described in §6.3.1, it switches to using bounded formulas (i.e., ones that use conditional tables instead of uninterpreted relations) for the remainder of template generation. Blockaid sets a table's bound to one plus the number of rows required to produce the sub-trace induced by the unsat core;[11] it relies on the solvers to produce small unsat cores to keep formula sizes manageable.

Care must be taken because using bounded formulas breaks soundness—a query compliant on small tables might not be on larger ones. Therefore, after a decision template is produced Blockaid verifies its soundness on the unbounded formula, and if this fails, increments the table bounds and retry.

6.3.3 Step Two: Find Value Constraints

Taking the template $\mathcal{D}_1 = (Q, \mathcal{T}_{\min}, \top)$ from Step 1, Blockaid generalizes it further by abstracting away the constants. To do so, Blockaid *parameterizes* $\mathcal{T}_{\min}$ and Q by replacing each occurrence of a constant with a fresh variable. We use a superscript "p" to denote the parameterized version of a query, tuple, or trace. Listing 3a shows $\mathcal{T}^{\mathrm{p}}_{\min}$ and Q^{p} from our example. As an optimization, Blockaid assigns the same variable (e.g., `x0`) to locations that are guaranteed by SQL semantics to be equal.

[9] For example, finite model finders in CVC4 [58] and Vampire [56] often time out or run out of memory on tables with only tens of columns.

[10] To avoid using quantifiers in these formulas, we drop the transitivity axiom for the uninterpreted less-than relation (§5.3).

[11] If the bounds are too small for a database to produce the trace, the resulting formula will be unsatisfiable regardless of compliance.

Listing 3: *Parameterization and candidate atoms for Listing 2a.*

(a) *Parameterized trace $\mathcal{T}^{p}_{min}$ and query q^{p}.*

```
2. SELECT * FROM Attendances
   WHERE UId = x0 AND EId = x1
   ↪ (UId = x0, EId = x1, ConfirmedAt = x2)
3. SELECT * FROM Events WHERE EId = x3
```

(b) *Candidate atoms (with symmetric duplicates removed).*

Form `x = v`:
- `MyUId = 1`
- `x0 = 1`
- `x1 = 42`
- `x2 = "05/04 1pm"`
- `x3 = 42`

Form `x = x'`:
- `MyUId = x0`
- `x1 = x3`

Form `x < x'`:
- `MyUId < x1`
- `MyUId < x3`
- `x0 < x1`
- `x0 < x3`

Blockaid must now generate a condition Φ such that the resulting template $\mathcal{D}_2 = (Q^{\mathrm{p}}, \mathcal{T}^{\mathrm{p}}_{\min}, \Phi)$ meets our goals. It picks as Φ a conjunction of atoms from a set of *candidate atoms*. Let $\mathbf{x}$ denote all variables generated from parameterization, and let ν map $\mathbf{x}$ to the replaced constants and $\mathbf{c}$ to the current context *ctx*.

Definition 6.10. The set of *candidate atoms* is defined as:

$$C = \bigcup \begin{cases} \{\texttt{x = v} & \mid x \in \mathbf{x} \cup \mathbf{c}, v = \nu(x) \neq \texttt{NULL}\} \\ \{\texttt{x IS NULL} & \mid x \in \mathbf{x} \cup \mathbf{c}, \nu(x) = \texttt{NULL}\} \\ \{\texttt{x = x'} & \mid x, x' \in \mathbf{x} \cup \mathbf{c}, \nu(x) = \nu(x') \neq \texttt{NULL}\} \\ \{\texttt{x < x'} & \mid x, x' \in \mathbf{x} \cup \mathbf{c}, \nu(x) < \nu(x')\} \end{cases}.$$

(We write atoms in `monospace font` to distinguish them from mathematical expressions. Following SQL, the "=" in an atom implies that both sides are non-`NULL`.)

Note that all candidate atoms hold on Q and $\mathcal{T}_{\min}$. Blockaid now selects a subset that not only guarantees compliance, but also imposes relatively few restrictions on the variables.

Definition 6.11. With respect to Q^{p} and $\mathcal{T}^{\mathrm{p}}_{\min}$, a subset of atoms $C_0 \subseteq C$ is *sound* if the decision template $(Q^{\mathrm{p}}, \mathcal{T}^{\mathrm{p}}_{\min}, \bigwedge C_0)$ is sound. ($\bigwedge C_0$ denotes the conjunction of atoms in C_0.)

Definition 6.12. Let $C_1, C_2 \subseteq C$. We say that C_2 is *at least as weak as* C_1 (denoted $C_1 \preceq C_2$) if $\bigwedge C_1 \implies \bigwedge C_2$, and that C_2 is *weaker* than C_1 if $C_1 \preceq C_2$ but $C_2 \not\preceq C_1$.

Example 6.13. Listing 3b shows all the candidate atoms from Listing 3a (after omitting symmetric ones in the `x = x'` group). Consider the following two subsets of atoms:

$$C_1 = \{\texttt{MyUId = x0},\ \ \texttt{x1 = 42},\ \ \texttt{x3 = 42}\},$$
$$C_2 = \{\texttt{MyUId = x0},\ \ \texttt{x1 = x3}\}.$$

While both are sound, C_2 is preferred over C_1 as it is weaker and thus applies in more scenarios. In fact, C_2 is *maximally* weak: there exists no subset that is both sound and weaker than C_2.

Ideally, Blockaid would produce a maximally weak sound subset of C to use as the template condition, but finding one can be expensive. It thus settles for finding a subset that is weak enough for practical generalization. It does so in three steps.

First, as a starting point, Blockaid generates a minimal unsat core of the formula:

$$\begin{aligned} & V^{ctx}(D_1) \subseteq V^{ctx}(D_2), && (\forall V \in \mathcal{V}) \\ & t_i^{\mathrm{p}} \in Q_i^{\mathrm{p}}(D_1), && (\forall (t_i^{\mathrm{p}}, Q_i^{\mathrm{p}}) \in \mathcal{T}_{\min}^{\mathrm{p}}) \\ [LC_i] \quad & c_i, && (\forall c_i \in C) \\ & Q^{\mathrm{p}}(D_1) \not\subseteq Q^{\mathrm{p}}(D_2). \end{aligned}$$

Let C_{core} denote the atoms whose label appears in the unsat core. For example, $C_{\text{core}} = \{\texttt{MyUId = x0}, \texttt{x1 = 42}, \texttt{x3 = 42}\}$.

Second, it augments C_{core} with other atoms that are implied by it: $C_{\text{aug}} = \{ c \in C \mid \bigwedge C_{\text{core}} \implies c \}$. In our example,

$$\begin{aligned} C_{\text{aug}} &= C_{\text{core}} \cup \{\texttt{x1 = x3}\} \\ &= \{\texttt{MyUId = x0}, \quad \texttt{x1 = 42}, \quad \texttt{x3 = 42}, \quad \texttt{x1 = x3}\}. \end{aligned}$$

C_{aug} enjoys a closure property: if $C_0 \subseteq C_{\text{aug}}$ and $C_0 \preceq C_1$, then $C_1 \subseteq C_{\text{aug}}$. In particular, C_{aug} contains a maximally weak sound subset of C. Thus, Blockaid focuses its search within C_{aug}.

Finally, as a proxy for weakness, Blockaid finds a *smallest* sound subset of C_{aug}, denoted C_{small}, breaking ties arbitrarily. It does so using the MARCO algorithm [43, 44, 55] for minimal unsatisfiable subset enumeration, modified to enumerate from small to large and to stop after finding the first sound subset. In our example, the algorithm returns $C_{\text{small}} = \{\texttt{MyUId=x0}, \texttt{x1=x3}\}$ of cardinality two, which is also a maximally weak subset (even though this might not be the case in general).[12] Nevertheless, searching for a smallest sound subset has produced templates that generalize well in practice.

At the end, Blockaid produces the decision template:

$$\mathcal{D}_2[\mathbf{x}, \mathbf{c}] = \left(Q^{\mathrm{p}}, \mathcal{T}_{\min}^{\mathrm{p}}, \bigwedge C_{\text{small}}\right).$$

Proposition 6.14. *$\mathcal{D}_2$ is sound, matches $(Q, \mathcal{T})$, and is at least as general as $\mathcal{D}_1$.*

As an optimization, whenever $\bigwedge C_{\text{small}} \implies x = y$ for $x, y \in \mathbf{x} \cup \mathbf{c}$, Blockaid replaces x with y in the template. This is how, e.g., in Listing 2b `?0` appears in both the trace and the query.

6.3.4 Optimizations

We implement two optimizations that improve the performance of template generation and the generality of templates.

Omit irrelevant tables. Given trace $\mathcal{T}$ and query Q, we call a table *relevant* if (1) it appears in $\mathcal{T}$ or Q, or (2) the table appears on the right-hand side of a database constraint of the form $Q_1 \subseteq Q_2$, given that a relevant table appears on the left.[13] Blockaid sets the size bounds of irrelevant tables to zero, reducing formula size while preserving compliance.

Split `IN`. A query Q that contains "c `IN` $(x_1, x_2, \ldots, x_n)$" often produces a template with a long trace. If Q is a basic query that does not contain the `NOT` operator, it can be split into $q_1, \ldots, q_n$ where q_i denotes Q with the `IN`-construct substituted with $c = x_i$, such that $Q \equiv q_1 \cup \ldots \cup q_n$. If $q_1, \ldots, q_n$ are all compliant then so is Q, and so Blockaid checks the subqueries instead. This is usually fast because $q_2, \ldots, q_n$ typically match the decision template generated from q_1. If any q_i is not compliant, Blockaid reverts to checking Q as a whole.

This optimization also improves generalization. Suppose Q' has structure identical to Q but a different number of `IN` operands. It would not match a template generated from Q, but its split subqueries q_i' could match the template from q_1.

6.4 Decision Cache and Template Matching

Blockaid stores decision templates in its *decision cache*, indexing them by their parameterized query using a hash map. When checking a query Q, Blockaid lists all templates whose parameterized query matches Q; for each such template, it uses recursive backtracking (with pruning optimizations) to search for a valuation that results in a match. This simple method proves efficient in practice as the templates tend to be small.

7 Implementation

We implemented Blockaid as a Java Database Connectivity (JDBC) driver that interposes on an underlying connection. It thus supports only applications on the JVM and runs within the web server, although our design allows it to reside elsewhere (e.g., in the database). The JDBC driver accepts custom commands that (1) set the request context, (2) clear the context and the trace, and (3) check an application cache read.

Blockaid parses SQL using Apache Calcite [9] and caches parser outputs. To check compliance, it uses Z3's Java binding [23] to generate formulas in SMT-LIB 2 format [7] and invokes an ensemble of solvers in parallel. Our ensemble consists of Z3 [24] (v4.8.12) and CVC5 [6] (v0.3) using default configurations, and Vampire [40] (v4.6.1) using six configurations from its CASC portfolio.[14] The ensemble is killed as soon as any solver finishes. If a query is not compliant, or all solvers time out after 5 s, Blockaid throws a Java `SQLException`.

To generate decision templates, Blockaid uses the same ensemble to produce the initial unsat core (§6.3.1), but kills the ensemble only when a solver returns a small core of up to 3 labels (subject to timeout). It uses only Z3 on bounded formulas.

Our prototype does not verify that queries return no duplicate rows and does not look at any `ORDER BY` columns. We manually ensured that queries in our evaluation return no duplicates and do not reveal inaccessible information through `ORDER BY`.

[12] For example, $\{\texttt{x < y}, \texttt{x < z}\}$ is strictly weaker than $\{\texttt{x < y}, \texttt{y < z}\}$ even though the two sets have the same cardinality.

[13] Every constraint encountered in our evaluation can be written in the form $Q_1 \subseteq Q_2$, including primary-key, foreign-key, and integrity constraints.

[14] https://github.com/vprover/vampire/blob/master/CASC/Schedules.cpp.

Table 1: Summary of schemas, policies, and code changes.

	diaspora*	Spree	Autolab
Schema & Policy			
# Tables modeled	35 / 52	46 / 93	17 / 28
# Constraints	108	122	51
# Policy views	108	84	57
# Cache key patterns	0	11	3
Code Changes (LoC)			
Boilerplate	12	17	12
Fetch less data	6	26	38
SQL feature	1	3	5
Parameterize queries	0	18	32
File system checking	0	0	9
Total	19	64	96

8 Evaluation

We use Blockaid to enforce data-access policies on three existing open-source web applications written in Ruby on Rails:

- **diaspora*** [25]: a social network with 850 k users.
- **Spree** [64]: an e-commerce app used by 50+ businesses.
- **Autolab** [5]: a course management app used at 20 schools.

For each application, we devised a data-access policy, modified its code to work with Blockaid, and measured its performance.

In summary: Blockaid imposes overheads of 2 %–12 % to median page load time when compliance decisions are cached; the decision templates produced by Blockaid generalize to other entities (users, etc.); and no query was falsely rejected in our benchmark. Instructions for reproducing our experiments can be found in Appendix A.

8.1 Constraints, Policies, and Annotations

Table 1 summarizes the constraints and policies for database tables queried in our benchmark, including any necessary application-level constraints (e.g., a reshared post is always public in diaspora*). Spree and Autolab use the Rails cache, and we annotate their cache key patterns with queries (§3.2).

Once a policy is given, transcribing it into views was straightforward. The more arduous task lied in divining the intended policy for an application, by studying its source code and interacting with it on sample data. This effort was complicated by edge cases in policies—e.g., a Spree item at an inactive location is inaccessible *except* when filtering for backorderable variants. Such edge cases had to be covered using additional views.

To give a sense of the porting effort, writing the Spree policy took one of us roughly a month. However, this process would be easier for the developer of a new application, who has a good sense of what policies are suitable and can create policies while building the application, amortizing the effort over time.

When writing the Autolab policy, we uncovered two access-check bugs in the application: (1) a persistent announcement (one shown on all pages of a course) is displayed regardless of whether it is active on the current date, and (2) an unreleased handout is hidden on its course page but can be downloaded from its assignment page. This experience corroborates the difficulty of making every access check airtight, especially for code bases that enjoy fewer maintenance resources.

8.2 Code Modifications

Our changes to application code fall into five categories:

1. **Boilerplate**: We add code that sends the request context to Blockaid at request start and clears the trace at request end.
2. **Fetch less data**: We modify code to not fetch potentially sensitive data unless it will be revealed to the user; some of these changes use the `lazy_column` gem [45].
3. **SQL features**: We modify some queries to avoid SQL features not supported by Blockaid (e.g., general left joins) without altering application behavior.
4. **Parameterize queries**: We make some queries parameterized so that Blockaid can effectively cache their parsing results. Most changes are mechanical rewrites of queries with comparisons, as idiomatic ways of writing comparisons [54] cause query parameters to be filled within Rails.
5. **File system checking**: Autolab uses files to store submissions; the file name are always accessible but the content is inaccessible during an exam. We modify it to store the submission content under a randomly generated file name and restrict access to the file name in the database (§3.2).

The code changes are summarized also in Table 1, which omits configuration changes, adaptations for JRuby, and experiment code. The changes range from 19 to 96 lines of code.

8.3 Experiment Setup and Benchmark

We deploy each application on an Amazon EC2 c4.8xlarge instance running Ubuntu 18.04. Because our prototype only supports JVM applications (§7), we run the applications using JRuby [36] (v9.3.0.0), a Ruby implementation atop the JVM (we use OpenJDK 17). In Rails's database configuration, we turn on `prepared_statements` so that Rails issues parameterized queries in the common case.[15] The applications run atop the Puma web server over HTTPS behind NGINX (which serves static files directly), and stores data in MySQL (and, if applicable, Redis) on the same instance. To reduce variability, all measurements are taken from a client on the same instance.

For each application, we picked five page loads that exercise various behaviors (Table 2). Each page load can fetch multiple URLs, some common among many pages (e.g., D9, which is the notifications URL). All queries issued are compliant, and all experiments are performed with the Rails cache populated.

8.4 Page Load Times

We start by measuring page load times (PLTs) using a headless Chrome browser (v96) driven by Selenium [63]. PLTs are reported as the time elapsed between `navigationStart` and `loadEventEnd` as defined by the `PerformanceTiming` interface [72]. The one exception is the Autolab "Submission" page,

[15] In case a Rails query is not fully parameterized (e.g., due to the use of raw SQL), it gets parameterized by Blockaid as described in §6.3.3.

Table 2: *Application benchmark. For a page we list the page URL followed by other URLs fetched (URLs for assets are excluded). When compliance decisions are cached, Blockaid incurs up to* 12 *% overhead to the median PLT over the modified applications.*

			Page Load Time (median / P95; default unit: ms)			
	URLs	**Description**	Original	Modified	Cached	No cache
diaspora*						
Simple post	D1, D2, D9	View a simple post shared with the user.	169 / 173	169 / 175	174 / 179	2.5 s / 2.6 s
Complex post	D3, D4, D9	View a public post with 30 votes and comments.	171 / 178	171 / 178	176 / 183	2.6 s / 2.7 s
Prohibited post	D5	Attempt to view an unauthorized post.	32 / 34	32 / 34	33 / 35	262 / 285
Conversation	D6, D9	View a conversation (5 messages).	253 / 258	255 / 262	260 / 267	2.1 s / 2.2 s
Profile	D7, D8, D9	View someone's profile (basic info and 3 posts).	142 / 148	145 / 152	150 / 156	1.3 s / 1.4 s
Spree						
Account	S1, S6–S8	View the user's account information.	74 / 80	76 / 83	78 / 84	588 / 611
Available item	S2, S6–S8	View a product for sale.	122 / 133	115 / 167	122 / 173	4.4 s / 4.4 s
Unavailable item	S3	Attempt to view a product no longer for sale.	20 / 22	21 / 23	22 / 24	350 / 371
Cart	S4, S6–S8	View the current shopping cart (3 items).	116 / 131	118 / 132	124 / 137	7.6 s / 7.7 s
Order	S5, S6–S8	View a summary and status of a previous order.	160 / 170	164 / 174	173 / 182	39 s / 39 s
Autolab						
Homepage	A1	View a summary of 3 courses enrolled.	56 / 61	59 / 64	65 / 70	1.4 s / 1.6 s
Course	A2, A3	View summary of one course (15 assignments).	84 / 96	87 / 101	97 / 116	3.9 s / 4.1 s
Assignment	A4	View a quiz (incl. 3 submissions and grades).	97 / 110	103 / 118	115 / 138	3.5 s / 3.6 s
Submission	A5	Download a previous homework submission.	22 / 26	26 / 31	27 / 33	1.1 s / 1.2 s
Gradesheet	A6	Instructor views grades for 51 enrollees.	456 / 474	474 / 493	504 / 530	72 s / 73 s

a file download, for which we report Chrome's download time instead. Since the client is on the same VM as the server, these experiments reflect the best-case PLT, as clients outside the instance / cloud are likely to experience higher network latency.

We report PLTs under four settings: *original* (unmodified application), *modified* (modified à la §8.2), *cached* (modified application under Blockaid with every query hitting the decision cache), and *no cache* (decision caching disabled). For the first three, we perform 3000 warmup loads before measuring the PLT of another 3000 loads. For *no cache*, where each run takes longer, we use 100 warmup loads and 100 measurement loads.

Table 2 shows that when compliance decisions are cached, Blockaid incurs up to 12 % overhead to median PLT over the modified application (and up to 17 % overhead to P95). With caching disabled, Blockaid incurs up to 236× higher median PLT. Compared with the original applications, the modified versions result in up to 6 % overhead to median PLT for all pages but Autolab's "Submissions", which suffers a 19 % overhead. (The P95 overhead is up to 7 % for all but two pages with up to 26 % overhead.) We will comment on these overheads in the next subsection, where we break down the pages into URLs.

8.5 Fetch Latency

To better understand page load performance, we separate out the individual URLs fetched by each page (Table 2), omitting URLs for assets, and measure the latency of fetching each URL (not including rendering time). The median latencies are shown in Figure 2. In addition to the four settings from §8.4, it includes performance under a "cold cache", where the decision cache is enabled but cleared at the start of each load (100 warmup runs followed by 100 measurements). When all compliance decisions are cached, Blockaid incurs up to 10 % of overhead (median 7 %) over "modified". In contrast, it incurs 7×–422× overhead on a cold decision cache, and 7×–310× overhead if the decision cache is disabled altogether.

For most URLs, "cold cache" is slower than "no cache" due to the extra template-generation step. Two exceptions are D4 and A6, where many structurally identical queries are issued, and so the performance gain from cache hits *within each URL* offsets the performance hit from template generation.

Compared to the original, the modified diaspora* and Spree are up to 5 % slower (median 2 %), but Autolab is up to 21 % slower (median 8 %). Autolab routinely reveals partial data on objects that are not fully accessible. For example, a user can distinguish among the cases where (1) a course doesn't exist, (2) a course exists but the user is not enrolled, and (3) the user is enrolled but the course is disabled. The original Autolab fetches the course in one SQL query but we had to split it into multiple—checking whether the course exists, whether it is disabled, etc.—and return an error immediately if one of these checks fails.

In one instance (S2), the modified version is 11 % faster than the original because we were able to remove queries for potentially inaccessible data that is never used in rendering the URL.

8.6 Solver Comparison

When a query arrives, Blockaid invokes an ensemble of solvers to check compliance when decision caching is disabled, and to generate a decision template on a cache miss when caching is enabled. The *winner*, in the no-cache case, is the first solver to return a decision; and in the cache-miss case, the first to return a

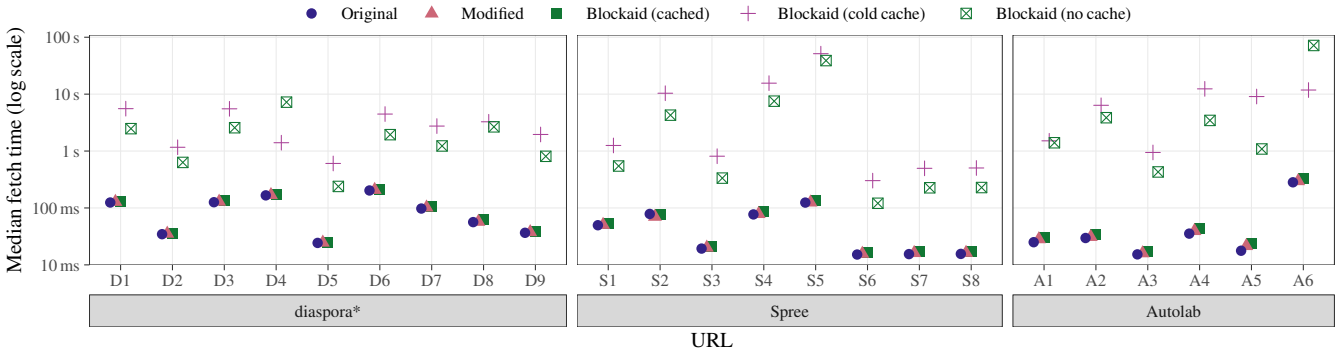


Figure 2: *URL fetch latency (median). With all compliance decisions cached, Blockaid incurs up to* 10 *% overhead over "modified".*

Listing 4: *Two (abridged) decision templates generated for the same parameterized query from Spree.* `Token` *is a Spree request context parameter identifying the current (possibly guest) user, and* `NOW` *is a built-in parameter storing the current time.*

(a) *This template doesn't fully generalize.*

```
SELECT * FROM products WHERE id IN (*, *, *)
↪ (id = ?1, available_on < ?NOW,
    discontinue_on IS NULL, deleted_at IS NULL, *)

SELECT * FROM variants WHERE id IN (*, *, *)
↪ (id = ?2, deleted_at IS NULL,
    discontinue_on IS NULL, product_id = ?1, *)
```

```
SELECT a.* FROM assets a
JOIN variants mv ON a.viewable_id = mv.id
JOIN variants ov ON mv.product_id = ov.product_id
WHERE mv.is_master AND mv.deleted_at IS NULL
  AND a.viewable_type = 'Variant' AND ov.id = ?2
```

(b) *This template does fully generalize.*

```
SELECT * FROM orders WHERE ...
↪ (id = ?0, token = ?Token, *)

SELECT * FROM line_items WHERE order_id = ?0
↪ (variant_id = ?1, *)
```

```
SELECT a.* FROM assets a
JOIN variants mv ON a.viewable_id = mv.id
JOIN variants ov ON mv.product_id = ov.product_id
WHERE mv.is_master AND mv.deleted_at IS NULL
  AND a.viewable_type = 'Variant' AND ov.id = ?1
```

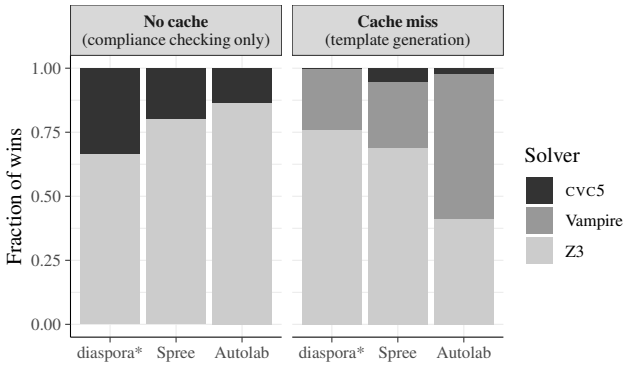


Figure 3: *Fraction of wins by each solver. "Vampire" covers a portfolio of six configurations (§7).*

small enough unsat core (§7), assuming the query is compliant.

Figure 3 shows, in the fetch latency experiments (§8.5), the fraction of wins by each solver in the two cases. In the no-cache case, the wins are dominated by Z3 followed by CVC5, with none for Vampire. In the cache-miss case, however, Vampire wins a significant portion of the time. This is because Z3 and CVC5 often finish quickly but with large unsat cores, causing Blockaid to wait till Vampire produces a smaller core.

8.7 Template Generalization

We found that the generated decision templates typically generalize to similar requests. The rest generalize in more restricted scenarios, and none is tied to a particular user ID, post ID, etc.

To illustrate how Blockaid might produce a template that fails to generalize fully, consider a query from Spree's cache key annotations (Listing 4). This query fetches assets for product variants in the user's order. (Here, the asset of a variant belongs to its product's "master variant".) Listing 4a shows a template that fails to generalize fully, for three reasons.

First, due to the queries with the `IN` operator in its premise (above the horizontal line), this template applies only when an order has exactly three variants. The `IN`-splitting optimization from §6.3.4 only applies to the query being checked, and we plan to handle such queries in the premise in future work.

Second, this template constrains the variant to be "not discontinued", which is defined as `discontinue_on IS NULL` or `discontinue_on >= NOW`. But because disjunctions are not supported in decision templates, Blockaid picked only the condition that matches the current variant (`IS NULL`).

Third, in this example there are multiple justifications for

this query's compliance, and Blockaid happened to pick one that does not always hold in a similar request. The policy states that a variant's asset can be viewed if it is not discontinued, or if it is part of the user's order.[16] This particular variant in the user's order happens to not be discontinued, and the template captures the former justification for viewing the asset. However, it does not apply to variants in the order that *are* discontinued; indeed, for such variants, Blockaid produces the template in Listing 4b, which generalizes fully. We could address this issue by finding multiple decision templates for every query.

Incidentally, inspecting decision templates has helped us expose overly permissive policies. When writing the Autolab policy, we missed a join condition in a view, a mistake that became apparent when Blockaid generated a template stating that an instructor for one course can view assignments for *all* courses. Although manual inspection of templates is not required for using Blockaid, doing so can help debug overly broad policies, whose undesirable consequences are often exposed by the general decision templates produced by Blockaid.

9 Additional Issues

Comparison to row- and cell-level policy. Several commercial databases (such as SQL Server [49] and Oracle [52]) implement *row- and/or cell-level* data-access policies, which specify accessible information at the granularity of rows or cells.

Such policies are less expressive than the view-based ones supported by Blockaid. For example, suppose we wish to allow each user to view everyone's timetables (i.e., the start and end times of the events they attend). Querying someone's timetable requires joining the *Events* and *Attendances* tables on the *EId* column, which must then be treated as visible by a cell-level policy. But this inevitably reveals meeting attendee information as well. Instead, we can implement this policy using a view:

```
SELECT UId, StartTime, EndTime
FROM   Events e
JOIN   Attendances a ON e.EId = a.EId
```

which lists the times of events attended without revealing *EId*.

False rejections. Even though false rejections of compliant queries never occurred in our evaluation, they remain a possibility for several reasons, including: (1) approximate rewriting into basic queries, which is incomplete; (2) our use of strong compliance; and (3) solver timeouts. Developers can reduce the chance of false rejections by running an application's end-to-end test suite under Blockaid before deployment, and manually examining any rejected query to determine whether it is due to a false positive, a bug in the code, or a misspecified policy.

Off-path deployment. If an operator is especially worried about false rejections affecting a website's availability, we can modify Blockaid to log potential violations instead of blocking any queries. We can even move Blockaid off-path by having the application stream its queries to Blockaid to be checked asynchronously, further reducing its performance impact.

What if Blockaid could issue its own queries? Suppose Blockaid can issue extra queries—but only ones answerable using the views, lest the decision itself reveal sensitive data—when checking compliance. Blockaid can now safely allow more queries from the application. For example, faced with the formerly non-compliant single query from Example 4.3:

```
SELECT Title FROM Events WHERE EId = 5
```

Blockaid can now *ask* whether the user attends Event #5 and if so, allow the query. In fact, under this setup the "necessary-and-sufficient" condition for application noninterference (in the sense of §4.3) becomes instance-based determinacy [39,59,74], a criterion less stringent than trace determinacy.

We decided against this design alternative for two reasons. First, it seems nontrivial to check instance-based determinacy efficiently: Blockaid must either figure out a small set of queries to ask, a difficult problem, or fetch all accessible information, an expensive task. Second, Blockaid is designed for conventional applications that do not *rely on* an enforcer for data-access compliance. These applications should not be issuing queries that fail trace determinacy but pass instance-based determinacy: Such queries can, in Blockaid's absence, reveal inaccessible information on *another* database and typically indicate application bugs. Thus, Blockaid is right to flag them.

Theoretically optimal templates. While decision templates produced by Blockaid are general enough in practice, they might not be *maximally general* among all sound templates that match the query and trace being checked. For one thing, the template condition might not be maximally weak (§6.3.3). For another, a maximally general template can have a *longer* trace than the concrete one, a possibility Blockaid never explores.

Fundamentally, our template generation algorithm is limited by its *black-box access* to the policy: It interacts with the policy solely by checking template soundness using a solver. Producing maximally general templates might require opening up this black box and having the policy guide template generation more directly, a path we plan to explore in future work.

10 Conclusion

Blockaid enforces view-based data-access policies on web applications in a semantically transparent and backwards compatible manner. It verifies policy compliance using SMT solvers and achieves low overhead using a novel caching and generalization technique. We hope that Blockaid's approach will help rule out data-access bugs in real-world applications.

Acknowledgments

We are grateful to Alin Deutsch and Victor Vianu for the many discussions about query determinacy, and to Nikolaj Bjørner, Alvin Cheung, Vivian Fang, and members of the Berkeley NetSys Lab for their help with the project. We also thank the anony-

[16] This is to allow users to view past purchases that are since discontinued.

mous reviewers and our shepherd Malte Schwarzkopf for their helpful comments. This research was funded in part by NSF grants 1817116 and 2145471, and gifts from Intel and VMware.

References

[1] Foto N. Afrati. Determinacy and query rewriting for conjunctive queries and views. *Theor. Comput. Sci.*, 412(11):1005–1021, 2011.

[2] Rakesh Agrawal, Paul Bird, Tyrone Grandison, Jerry Kiernan, Scott Logan, and Walid Rjaibi. Extending relational database systems to automatically enforce privacy policies. In *ICDE*, pages 1013–1022. IEEE Computer Society, 2005.

[3] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. Hippocratic databases. In *VLDB*, pages 143–154. Morgan Kaufmann, 2002.

[4] Warwick Ashford. Facebook photo leak flaw raises security concerns, March 2015. https://www.computerweekly.com/news/2240242708/Facebook-photo-leak-flaw-raises-security-concerns.

[5] Autolab Project. https://autolabproject.com/.

[6] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A versatile and industrial-strength SMT solver. In *TACAS*, 2022.

[7] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017.

[8] Clark W. Barrett and Cesare Tinelli. Satisfiability modulo theories. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 305–343. Springer, 2018.

[9] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J. Mior, and Daniel Lemire. Apache calcite: A foundational framework for optimized query processing over heterogeneous data sources. In *SIGMOD*, pages 221–230. ACM, 2018.

[10] Gabriel Bender, Lucja Kot, and Johannes Gehrke. Explainable security for relational databases. In *SIGMOD*, pages 1411–1422. ACM, 2014.

[11] Gabriel Bender, Lucja Kot, Johannes Gehrke, and Christoph Koch. Fine-grained disclosure control for app ecosystems. In *SIGMOD*, pages 869–880. ACM, 2013.

[12] Alexander Brodsky, Csilla Farkas, and Sushil Jajodia. Secure databases: Constraints, inference channels, and monitoring disclosures. *IEEE Trans. Knowl. Data Eng.*, 12(6):900–919, 2000.

[13] Kristy Browder and Mary Ann Davidson. The virtual private database in Oracle9iR2. *Oracle Technical White Paper*, 2002.

[14] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. Optimizing database-backed applications with query synthesis. In *PLDI*, pages 3–14. ACM, 2013.

[15] Adam Chlipala. Static checking of dynamically-varying security policies in database-backed applications. In *OSDI*, pages 105–118. USENIX Association, 2010.

[16] Adam Chlipala. Ur: statically-typed metaprogramming with type-level record computation. In *PLDI*, pages 122–133. ACM, 2010.

[17] Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. Secure web applications via automatic partitioning. In *SOSP*, page 31–44. ACM, 2007.

[18] Shumo Chu, Brendan Murphy, Jared Roesch, Alvin Cheung, and Dan Suciu. Axiomatic foundations and algorithms for deciding semantic equivalences of SQL queries. *Proc. VLDB Endow.*, 11(11):1482–1495, 2018.

[19] Shumo Chu, Konstantin Weitz, Alvin Cheung, and Dan Suciu. HoTTSQL: proving query rewrites with univalent SQL semantics. In *PLDI*, pages 510–524. ACM, 2017.

[20] E. F. Codd. Relational completeness of data base sublanguages. In *Database Systems*. Prentice-Hall, 1972.

[21] Ellis S. Cohen. Information transmission in computational systems. In *SOSP*, pages 133–139. ACM, 1977.

[22] Brian J. Corcoran, Nikhil Swamy, and Michael W. Hicks. Cross-tier, label-based security enforcement for web applications. In *SIGMOD*, pages 269–282. ACM, 2009.

[23] Leonardo de Moura. Z3 for Java. https://leodemoura.github.io/blog/2012/12/10/z3-for-java.html.

[24] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[25] Diaspora Foundation. The diaspora* project. https://diasporafoundation.org/.

[26] Django Software Foundation. Models | Django documentation | Django. https://docs.djangoproject.com/en/3.2/topics/db/models/.

[27] Tomasz Gogacz and Jerzy Marcinkowski. The hunt for a red spider: Conjunctive query determinacy is undecidable. In *30th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2015, Kyoto, Japan, July 6-10, 2015*, pages 281–292. IEEE Computer Society, 2015.

[28] Tomasz Gogacz and Jerzy Marcinkowski. Red spider meets a rainworm: Conjunctive query finite determinacy is undecidable. In *PODS*, pages 121–134. ACM, 2016.

[29] Joseph A. Goguen and José Meseguer. Security policies and security models. In *1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982*, pages 11–20. IEEE Computer Society, 1982.

[30] Matthew Green. Twitter post: Piazza offers anonymous posting, but does not hide each user's total number of posts, October 2017. https://twitter.com/matthew_d_green/status/925053953330634753.

[31] Paolo Guagliardo and Leonid Libkin. A formal semantics of SQL queries, its validation, and applications. *Proc. VLDB Endow.*, 11(1):27–39, 2017.

[32] Marco Guarnieri and David A. Basin. Optimal security-aware query processing. *Proc. VLDB Endow.*, 7(12):1307–1318, 2014.

[33] Raju Halder and Agostino Cortesi. Fine grained access control for relational databases by abstract interpretation. In *ICSOFT*, volume 170, pages 235–249. Springer, 2010.

[34] Daniel Hedin and Andrei Sabelfeld. A perspective on information-flow control. In Tobias Nipkow, Orna Grumberg, and Benedikt Hauptmann, editors, *Software Safety and Security - Tools for Analysis and Verification*, volume 33 of *NATO Science for Peace and Security Series - D: Information and Communication Security*, pages 319–347. IOS Press, 2012.

[35] Tomasz Imielinski and Witold Lipski Jr. Incomplete information in relational databases. *J. ACM*, 31(4):761–791, 1984.

[36] JRuby – the Ruby programming language on the JVM. https://www.jruby.org.

[37] Eddie Kohler. Hide review rounds from paper authors • kohler/hotcrp@5d53abc, March 2013. https://github.com/kohler/hotcrp/commit/5d53ab.

[38] Eddie Kohler. Download PC review assignments obeys paper administrators • kohler/hotcrp@80ff966, March 2015. https://github.com/kohler/hotcrp/commit/80ff96.

[39] Paraschos Koutris, Prasang Upadhyaya, Magdalena Balazinska, Bill Howe, and Dan Suciu. Query-based data pricing. In *PODS*, pages 167–178. ACM, 2012.

[40] Laura Kovács and Andrei Voronkov. First-order theorem proving and Vampire. In *CAV*, volume 8044 of *Lecture Notes in Computer Science*, pages 1–35. Springer, 2013.

[41] Kristen LeFevre, Rakesh Agrawal, Vuk Ercegovac, Raghu Ramakrishnan, Yirong Xu, and David J. DeWitt. Limiting disclosure in hippocratic databases. In *VLDB*, pages 108–119. Morgan Kaufmann, 2004.

[42] Nico Lehmann, Rose Kunkel, Jordan Brown, Jean Yang, Niki Vazou, Nadia Polikarpova, Deian Stefan, and Ranjit Jhala. STORM: refinement types for secure web applications. In *OSDI*, pages 441–459. USENIX Association, 2021.

[43] Mark H. Liffiton and Ammar Malik. Enumerating infeasibility: Finding multiple MUSes quickly. In *CPAIOR*, volume 7874 of *Lecture Notes in Computer Science*, pages 160–175. Springer, 2013.

[44] Mark H. Liffiton, Alessandro Previti, Ammar Malik, and João Marques-Silva. Fast, flexible MUS enumeration. *Constraints An Int. J.*, 21(2):223–250, 2016.

[45] Jorge Manrubia. jorgemanrubia/lazy_columns: Rails plugin that adds support for lazy-loading columns in active record models, 2015. https://github.com/jorgemanrubia/lazy_columns.

[46] Alana Marzoev, Lara Timbó Araújo, Malte Schwarzkopf, Samyukta Yagati, Eddie Kohler, Robert Morris, M. Frans Kaashoek, and Sam Madden. Towards multiverse databases. In *HotOS*, 2019.

[47] Mark Maunder. Vulnerability in WordPress Core: Bypass any password protected post. CVSS score: 7.5 (High), June 2016. https://www.wordfence.com/blog/2016/06/wordpress-core-vulnerability-bypass-password-protected-posts/.

[48] Aastha Mehta, Eslam Elnikety, Katura Harvey, Deepak Garg, and Peter Druschel. Qapla: Policy compliance for database-backed systems. In *USENIX Security*, pages 1463–1479. USENIX Association, 2017.

[49] Microsoft. Row-level security - SQL Server, 2021. https://docs.microsoft.com/en-us/sql/relational-databases/security/row-level-security.

[50] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *POPL*, pages 228–241. ACM, 1999.

[51] Alan Nash, Luc Segoufin, and Victor Vianu. Views and queries: Determinacy and rewriting. *ACM Trans. Database Syst.*, 35(3):21:1–21:41, 2010.

[52] Oracle. Using Oracle Virtual Private Database to control data access. https://docs.oracle.com/database/121/DBSEG/vpd.htm.

[53] Daniel Pasaila. Conjunctive queries determinacy and rewriting. In *ICDT*, pages 220–231. ACM, 2011.

[54] Alex Piechowski. Rails: How to use greater than/-less than in Active Record where statements, 2019. https://piechowski.io/post/how-to-use-greater-than-less-than-active-record/.

[55] Alessandro Previti and João Marques-Silva. Partial MUS enumeration. In *AAAI*. AAAI Press, 2013.

[56] Giles Reger, Martin Suda, and Andrei Voronkov. Finding finite models in multi-sorted first-order logic. In *SAT*, volume 9710 of *Lecture Notes in Computer Science*, pages 323–341. Springer, 2016.

[57] Raymond Reiter. On closed world data bases. In *Symposium on Logic and Data Bases*, Advances in Data Base Theory, pages 55–76, New York, 1977. Plemum Press.

[58] Andrew Reynolds, Cesare Tinelli, Amit Goel, Sava Krstic, Morgan Deters, and Clark W. Barrett. Quantifier instantiation techniques for finite model finding in SMT. In *CADE*, volume 7898 of *Lecture Notes in Computer Science*, pages 377–391. Springer, 2013.

[59] Shariq Rizvi, Alberto O. Mendelzon, S. Sudarshan, and Prasan Roy. Extending query rewriting techniques for fine-grained access control. In *SIGMOD*, pages 551–562. ACM, 2004.

[60] Ruby on Rails Guides. Active Record basics. https://edgeguides.rubyonrails.org/active_record_basics.html.

[61] Luc Segoufin and Victor Vianu. Views and queries: determinacy and rewriting. In *PODS*, pages 49–60. ACM, 2005.

[62] Jie Shi, Hong Zhu, Ge Fu, and Tao Jiang. On the soundness property for SQL queries of fine-grained access control in DBMSs. In *ICIS*, pages 469–474. IEEE Computer Society, 2009.

[63] Software Freedom Conservancy. SeleniumHQ: Browser automation, 2021. https://www.selenium.dev/.

[64] Spree Commerce - a headless open-source ecommerce platform. https://spreecommerce.org/.

[65] Ben Stock. Search leaks hidden tags • Issue #135 • kohler/hotcrp, June 2018. https://github.com/kohler/hotcrp/issues/135.

[66] Michael Stonebraker and Eugene Wong. Access control in a relational data base management system by query modification. In *ACM Annual Conference*, pages 180–186. ACM, 1974.

[67] Margus Veanes, Pavel Grigorenko, Peli de Halleux, and Nikolai Tillmann. Symbolic query exploration. In *ICFEM*, volume 5885 of *Lecture Notes in Computer Science*, pages 49–68. Springer, 2009.

[68] Margus Veanes, Nikolai Tillmann, and Jonathan de Halleux. Qex: Symbolic SQL query explorer. In *LPAR-16*, volume 6355 of *Lecture Notes in Computer Science*, pages 425–446. Springer, 2010.

[69] Chenglong Wang, Alvin Cheung, and Rastislav Bodík. Synthesizing highly expressive SQL queries from input-output examples. In *PLDI*, pages 452–466. ACM, 2017.

[70] Qihua Wang, Ting Yu, Ninghui Li, Jorge Lobo, Elisa Bertino, Keith Irwin, and Ji-Won Byun. On the correctness criteria of fine-grained access control in relational databases. In *VLDB*, pages 555–566. ACM, 2007.

[71] Yuepeng Wang, Isil Dillig, Shuvendu K. Lahiri, and William R. Cook. Verifying equivalence of database-driven applications. *Proc. ACM Program. Lang.*, 2(POPL):56:1–56:29, 2018.

[72] Zhiheng Wang. Navigation timing. W3C recommendation, W3C, December 2012. https://www.w3.org/TR/2012/REC-navigation-timing-20121217.

[73] Jean Yang, Travis Hance, Thomas H. Austin, Armando Solar-Lezama, Cormac Flanagan, and Stephen Chong. Precise, dynamic information flow for database-backed applications. In *PLDI*, pages 631–647. ACM, 2016.

[74] Zheng Zhang and Alberto O. Mendelzon. Authorization views and conditional query containment. In *ICDT*, volume 3363 of *Lecture Notes in Computer Science*, pages 259–273. Springer, 2005.

A Artifact Appendix

Abstract

Our artifact includes our Blockaid implementation, which is compatible with applications that can run atop the JVM and connect to a database via JDBC (§7). We also provide the three applications we used for our evaluation—modified according to §8.2—as well as the data-access policy we wrote for each. Finally, we provide a setup for reproducing the evaluation results from §8.

Table 3: *Where artifact contents are hosted.*

Content	Location	Branch / tag / release
Artifact README	https://github.com/blockaid-project/artifact-eval	`main` branch
Blockaid source	https://github.com/blockaid-project/blockaid	`main` branch (latest version)
		`osdi22ae` branch (AE version)[a]
Experiment launcher	https://hub.docker.com/repository/docker/blockaid/ae	`latest` tag
Launcher source	https://github.com/blockaid-project/ae-launcher	`main` branch
VM image	https://github.com/blockaid-project/ae-vm-image	`osdi22ae` release
Experiment scripts	https://github.com/blockaid-project/experiments	`osdi22ae` branch
Applications		
diaspora*	https://github.com/blockaid-project/diaspora	`blockaid` branch[b]
Spree	https://github.com/blockaid-project/spree	`bv4.3.0-orig` branch (original)[c]
		`bv4.3.0` branch (modified)[d]
Autolab	https://github.com/blockaid-project/Autolab	`bv2.7.0-orig` branch (original)[c]
		`bv2.7.0` branch (modified)[d]
Policies for applications	https://github.com/blockaid-project/app-policies	`main` branch

[a] The "AE version" is the version of Blockaid used in artifact evaluation.
[b] The same diaspora* branch is used for both baseline and Blockaid measurements. The code added for Blockaid is gated behind conditionals that check whether Blockaid is in use.
[c] "(original)" denotes the original application modified only to run on top of JRuby.
[d] "(modified)" denotes the "(original)" code additionally modified to work with Blockaid (§8.2).

Scope

This artifact can be used to run the main experiments from this paper: the page load time (PLT) measurements (§8.4) and the fetch latency measurements (§8.5 and §8.6) on the three applications. From these experiments, it generates Table 2 (with URLs and descriptions omitted), Figure 2, and Figure 3. Because the full experiment can be time- and resource-consuming (taking roughly 15 hours on six Amazon EC2 c4.8xlarge instances), the experiment launcher can be configured to take fewer measurement rounds at the expense of accuracy.

Our Blockaid implementation can also be used to enforce data-access policies on new applications, as long as they have been modified to satisfy our requirements (§3.3), run atop the JVM, and connect to the database using JDBC (§7).

Contents

This artifact consists of our Blockaid implementation, the three applications used in our evaluation (with modifications described in §8.2), the data-access policy we wrote for each, and scripts and virtual machine image for running the experiments.

Hosting

See Table 3.

Requirements

The experiment launcher, which relies on Docker, launches experiments on Amazon EC2 and so requires an AWS account. By default, it uses six c4.8xlarge instances—to run the PLT and fetch latency experiments for the three applications simultaneously. However, it can be configured to launch fewer instances at a time (e.g., to run the experiments serially, using one instance at a time).

B Proof: From Query Compliance to Application Noninterference

Proof of Theorem 4.11. We prove each part separately:

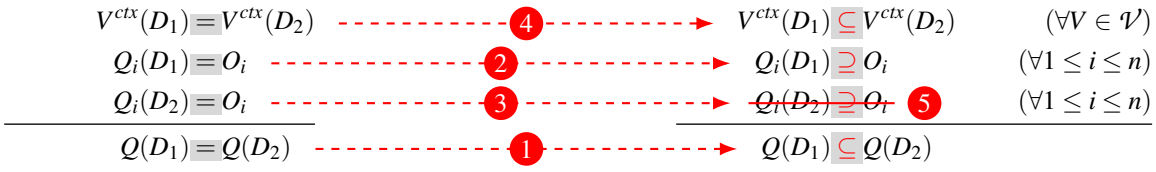
Part 1. Suppose $E(ctx, Q, \mathcal{T}) = \checkmark$ only when Q is ctx-compliant to $\mathcal{V}$ given $\mathcal{T}$. Pick any $\mathcal{P}$, ctx, and req, and let D_1 and D_2 be databases such that $V^{ctx}(D_1) = V^{ctx}(D_2)$ for all $V \in \mathcal{V}$.

Consider executions $\mathcal{P}^E(ctx, req, D_1)$ and $\mathcal{P}^E(ctx, req, D_2)$. We will show that the two executions coincide, by induction on the number of steps taken by $\mathcal{P}$. This will imply that $\mathcal{P}^E(ctx, req, D_1) = \mathcal{P}^E(ctx, req, D_2)$, finishing the proof.

Base case. Because $\mathcal{P}$ is assumed to be deterministic, so is $\mathcal{P}^E$, and so the two executions start off with the same program state.

Inductive step. Suppose the two executions coincide after $\mathcal{P}$ has taken i steps. Consider the $i+1$st step taken on both sides:

- Suppose this step is a query Q to the database. Let $\mathcal{T}$ denote the (same) trace maintained by the two executions so far. If $E(ctx, Q, \mathcal{T}) = \text{✗}$, then both executions terminate with an error. Otherwise, Q must be ctx-compliant to $\mathcal{V}$ given $\mathcal{T}$. By assumption, $V^{ctx}(D_1) = V^{ctx}(D_2)$ for all $V \in \mathcal{V}$; and by the construction of $\mathcal{P}^E$, $Q_i(D_1) = Q_i(D_2) = O_i$ for every $(Q_i, O_i) \in \mathcal{T}$. Therefore, we must have $Q(D_1) = Q(D_2)$, and so the two executions end up in the same program state after this step.
- If this step is *not* a database query, then its behavior depends only on $\mathcal{P}$'s program state and inputs ctx and req, all of which are the same across the two executions at this time.



(a) *Compliance (Definition 4.5).* ***(b)*** *Strong compliance (Definition 5.4).*

Figure 4: *The definition of compliance is turned into that of strong compliance in five steps. Dashed arrows for Steps 1–4 denote modifications; the solid line (strikethrough) for Step 5 denotes removal.*

Part 2. Suppose that E correctly enforces $\mathcal{V}$. Pick any ctx, Q, and prefix E-allowed $\mathcal{T} = \{(Q_i, O_i)\}_{i=1}^{n}$ such that $E(ctx, Q, \mathcal{T}) = \checkmark$. Consider the following program $\mathcal{P}$:

procedure $\mathcal{P}(ctx, req, D)$
  for $i \leftarrow 1..n$ **do**
    issue $Q_i(D)$ and discard the result
  return $Q(D)$

To show that Q is ctx-compliant to $\mathcal{V}$ given $\mathcal{T}$, let D_1 and D_2 be databases such that:

$$V^{ctx}(D_1) = V^{ctx}(D_2), \qquad (\forall V \in \mathcal{V}) \qquad (6)$$
$$Q_i(D_1) = O_i, \qquad (\forall 1 \le i \le n) \qquad (7)$$
$$Q_i(D_2) = O_i. \qquad (\forall 1 \le i \le n) \qquad (8)$$

Let req be a request, and consider executions $\mathcal{P}^E(ctx, req, D_1)$ and $\mathcal{P}^E(ctx, req, D_2)$. Because $\mathcal{T}$ is prefix E-allowed, neither execution ends in a policy violation error. Therefore,

$$\mathcal{P}^E(ctx, req, D_1) = \mathcal{P}(ctx, req, D_1) = Q(D_1),$$
$$\mathcal{P}^E(ctx, req, D_2) = \mathcal{P}(ctx, req, D_2) = Q(D_2).$$

Furthermore, because E correctly enforces $\mathcal{V}$, Equation (6) implies that $\mathcal{P}^E(ctx, req, D_1) = \mathcal{P}^E(ctx, req, D_2)$. We thus have $Q(D_1) = Q(D_2)$, concluding Q to be ctx-compliant to $\mathcal{V}$ given $\mathcal{T}$. □

C From Compliance to Strong Compliance

To understand when strong compliance fails to coincide with compliance, let us look at Figure 4, which illustrates how we modified the definition of compliance (Definition 4.5) into that of strong compliance (Definition 5.4) in five steps.

Step 1 does not affect the truthfulness of the formula since D_1 and D_2 are symmetric. Steps 2–3 adopt an *open-world assumption* (OWA) [57], treating every query as returning partial results. Under this assumption, a trace can no longer represent the *nonexistence* of a returned row; this can cause Blockaid may falsely reject a query. However, such cases never arose in our evaluation. The OWA also proves convenient during decision template generation (§6.3.1) when Blockaid computes a minimal sub-trace (which, by necessity, represents partial information) that guarantees strong compliance.

To see how Step 4 affects the definition, suppose there are no database dependencies and the trace is empty (so Steps 2–3 are irrelevant). In this scenario, compliance holds iff $\mathcal{V}$ *determines* Q [51, 61], while strong compliance states that Q has a *monotonic* rewriting using $\mathcal{V}$. There are cases where determinacy holds but no monotonic rewriting exists; e.g., Nash et al. [51, § 5.1] present an example in terms of conjunctive queries.

Finally, in Step 5 we drop the condition that D_2 be consistent with the trace. We can show by induction that this condition is redundant as long as each query in the trace is strongly compliant given the trace before it (which is the case in Blockaid).