Sensitive-data Protection for Today's Web Applications

by

Wen Zhang

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Scott Shenker, Chair
Assistant Professor Aurojit Panda
Professor Sylvia Ratnasamy
Associate Professor Alvin Cheung

Summer 2025

Sensitive-data Protection for Today’s Web Applications

Abstract

Sensitive-data Protection for Today's Web Applications

by

Wen Zhang

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Scott Shenker, Chair

As web applications increasingly handle sensitive user data, protecting that data from unauthorized access is more critical than ever. Yet, despite decades of research on access control, data leaks remain prevalent—not due to a lack of solutions, but because existing solutions are difficult to adopt by today's deployed applications. Two key challenges hinder adoption: (1) many solutions require nonstandard programming models that are incompatible with mainstream web frameworks, and (2) developers must manually define access-control policies—a time-consuming and error-prone task, particularly for legacy applications that lack such policies.

If we want to solve the societal problem of sensitive-data protection, we must meet today's applications where they are. This dissertation focuses on developing access-control techniques that can be easily applied to existing applications. We will present two systems: Blockaid, which performs fine-grained access control on existing web applications with minimal modification, and Ote, which aids in policy creation by extracting implicit policies embedded in legacy code. By supporting today's applications without requiring a redesign, our approach aims to bring practical data protection to real-world deployments.

To my family.

Contents

List of Figures

List of Tables

Acknowledgments

My first and deepest thanks go to my advisor, **Scott Shenker**. Scott was the reason I chose Berkeley, and looking back, I certainly made the right choice. He is, simply put, the best advisor I could have asked for. From Scott, I learned to look beyond low-hanging fruit and instead to "take a step back", ask fundamental questions, relentlessly seek simplicity and clarity, and find the *right way* to solve a problem—the way that changes how people think. I always feel comfortable walking into Scott's office to discuss any idea that pops into my head, whether related to my research or in a completely different area, whether well-formed or (as is often the case) half-baked. Even when I have no idea what I'm talking about, Scott always humors me, listens patiently to my ramblings, and skillfully finds the nuggets of gold within. His mentorship transcends any single research topic and has emboldened me to venture into any new area that interests me.

I owe tremendous gratitude to **Aurojit Panda**, my unofficial second advisor. Panda is a walking encyclopedia of computer science (and many other things). No matter what subject I bring up—a thorny research problem, a question about a random paper I've read, or a new industry technology—Panda always has something intelligent to offer. I have had the privilege of working with him since my first day at Berkeley, and I am still amazed by his generosity with his time and knowledge. Most of all, working with Panda has made research infinitely more enjoyable.

I am also deeply grateful to my other committee members, **Sylvia Ratnasamy** and **Alvin Cheung**. Over the years, Sylvia has given me invaluable feedback on my research. My only regret is not having the opportunity to work with her more closely, but whenever I needed help, she was always there for me. Alvin brought unique expertise from both database and programming-language research. He would patiently dissect my nascent ideas and sharpen them into something concrete—I have learned a great deal from him.

I first met **Natacha Crooks** when I took her class on distributed systems and have since had the privilege of collaborating with her. As a newcomer to distributed systems, I always have countless basic questions—or worse, vague confusion that I cannot even articulate. Natacha would patiently listen, help me frame my thoughts, and guide me to the precise answers I am seeking. I also thank Natacha for her endless encouragement as I navigate my career path.

I began working with **Mooly Sagiv** at the start of the Blockaid project. He singlehandedly introduced me to both database theory and the amazing capabilities of SMT solvers. Whenever I was stuck, Mooly could point the way towards progress, whether by sharpening a definition or by leveraging a suitable tool. Even after our paper was published, Mooly continued to support my career, making time to meet even when he was busiest with his company. I learned so much from Mooly and truly appreciate his guidance.

I interned with **Irene Zhang** at Microsoft Research in the summer of 2019; we started the Persimmon project then and extended our collaboration beyond the internship. Irene skillfully located three servers with Intel® Optane™ DC Persistent Memory, which were essential for our experiments. But more importantly, she introduced me to the world of datacenter computing and taught me the valuable skill of thinking about problems at a high level and presenting the big picture.

Throughout my PhD, I have been fortunate to collaborate with many people at Berkeley: Dev Bali, Eric Sheng, Jamison Kerney, Matei Zaharia, Micah Murray, Michael Alan Chang, Narek Galstyan, Peter Xiang Gao, Rishabh Iyer, Sam Son, Silvery Fu, Zhihong Luo, Jiwon Park, and Shadaj Laddad. I have learned a lot from each of them, and I thank them for putting up with my incessant questions.

I give special thanks to **Silvery Fu**. Silvery and I started as graduate students in the NetSys Lab at the same time; he has always been there for me when I need someone to talk to, *and* even when I don't realize I do. I have found in Silvery a true friend, and I am confident that our friendship will extend well beyond graduate school.

I have enjoyed the company of many others—friends from the NetSys Lab: Aisha Mushtaq, Akshay Narayan, Alexander Krentsel, Amin Tootoonchian, Amy Ousterhout, Anwar Hithnawi, Chang Lan, Christopher Branner-Augmon, Edward Oakes, Emily Marx, Emmanuel Amaro, Ethan J. Jackson, Hannah B. Pasandi, Lloyd Brown, Murphy McCauley, Radhika Mittal, Sarah McClure, Tenzin Samten Ukyab, Tess Despres, Yotam Harchol, and Ziming Mao; and friends from the RISE-Lab / Sky Computing Lab and beyond: Geng Zhao, Zongheng Yang, Sam Kumar, Conor & Laura Power, David Chu, Eyal Sela, Jaewan Hong, Jean-Luc Watson, Jenny Huang, Julien Piet, Justin Wong, Peter Schafhalter, Samyu Yagati, Shishir Patil, Shu Liu, Stephanie Wang, Tian Xia, Tianjun Zhang, Wenshuo Guo, and many more!

I am grateful to our lab's administrative staff—Ivan Ortega, Jon Kuroda, Kailee Truong, Kattt Atchley, Tram Vu, Angie Goodwin, Boban Zarkovich, Dave Schonenberg, and Shane Knapp—who have skillfully kept the lab running smoothly.

I began my research journey as an undergraduate at Stanford University, where I had the great fortune to work with **Elliott Slaughter** and **Alex Aiken**. Elliott and Alex were the reason why I chose to pursue a PhD. They introduced me to the fun and excitement of research and encouraged me to apply to graduate school, believing in my potential even when I did not. I am profoundly grateful to them for setting me on a path that has turned out to be so rewarding.

* * *

I would not be here without my parents, **Yan Lang** and **Zhuo Zhang**, who have unconditionally loved and supported me from the beginning, consistently put my needs before their own, and kept my best interests in mind in everything they did. They also had the incredible foresight to recognize the importance of mastering both computer technology and the English language early in my education, which has certainly prepared me well for writing this Computer Science dissertation in English.

Last but certainly not least, I thank my wife, **Vivian Fang**. She has been my constant companion on this journey, possessing the remarkable ability to pull me away from my desk—out of the apartment, even—for much-needed breaks that I didn't realize I needed. Proud as I am of the work in this dissertation, the most precious thing I have gained during my PhD is undoubtedly my relationship with Vivian. I thank her for being a loyal and loving partner, for adding color to my life, and for bringing into our lives our two cats, **Natto** and **Mini**, who have provided immeasurable joy and support throughout this journey.

Chapter 1

Introduction

We use many web applications in our everyday lives, applications that store and serve sensitive user data. Students log into university portals to check grades, patients access healthcare dashboards to review medical records, and billions of people rely on messaging platforms to stay connected with friends and family. In every case, users expect that their personal data remains confidential and is revealed only to authorized parties.

Protecting such data is therefore a matter of both social importance and, in many domains, legal obligation. Governments have long regulated the disclosure of information deemed particularly sensitive—e.g., FERPA for education records and HIPAA for medical records. But even outside strictly regulated settings, platform operators face contractual, reputational, and ethical pressure to avoid unauthorized disclosures.

Yet data leaks continue unabated, suggesting that the status quo for data-protection in web applications is woefully inadequate. To explain why, we will define the problem (§1.1), discuss why the prevailing approaches for data protection are insufficient (§1.2), and highlight where past research falls short (§1.3). We will then outline our contributions—two complementary systems that, taken together, form a holistic solution for protecting sensitive data in web applications today (§1.4).

1.1 The Problem

Figure 1.1 shows the simplified architecture of a typical web application. A user interacts with the application—say, a calendar—backed by a database holding records for *all* users. The browser sends an HTTP request to the application, which issues a series of queries to the database and uses the results to construct a response to send back to the user. Both the application and the database are controlled by the web application's *operator*, whereas the user is free to craft arbitrary requests.

Under this setting, we set out to tackle one problem:

> **How should the operator ensure that the user sees only the data they are allowed to see?**

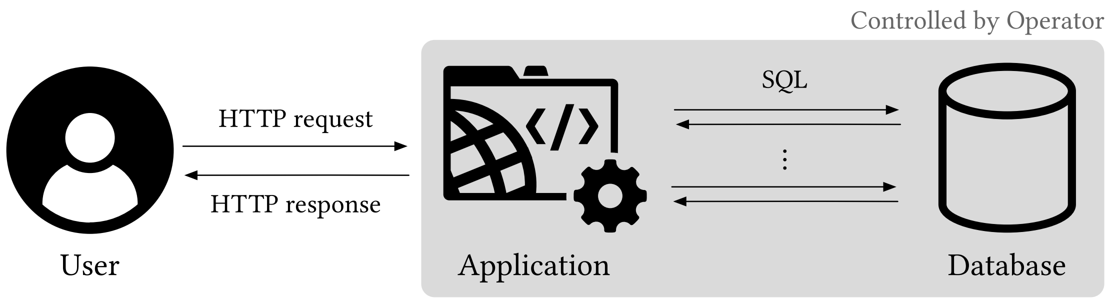


Figure 1.1: A simplified architecture of a typical web application.

For example, the user should be able to see the calendar events that they are invited to, but not the private events of others.

Remark 1.1. Sensitive-data protection is a broad domain, under which there are many other problems that are just as important but we will not be addressing in this dissertation. For example, we will not try to protect the user's data *from the operator*, for which many good cryptographic techniques have been developed. And we will not try to prevent the identification of personal records from aggregate statistics, for which differential privacy [49] is a good solution. Rather, we are solely taking the perspective of the operator, and making sure that the user is shown the right data.

1.2 The Status Quo

At first glance, this problem doesn't seem hard to solve: Since the user interacts only with the application layer, why not just implement the application logic to reveal only the allowed data?

Indeed, today's status quo is to limit data disclosure within the application code. There are two common code patterns for achieving this:

Query filters To serve a *general request* for data, the developer carefully crafts a SQL query to return only data that the user is allowed to see. For example, in our calendar application, to implement the HTTP endpoint `/all_events` the developer may write the query:

```
SELECT *
FROM   Events
       JOIN Attendance
         ON Attendance.EId = Events.EId
WHERE  Attendance.UId = ?MyUId
```

This query returns only those events that the current user is attending.

Access checks To serve a request for a *specific data item*, the developer may write an if statement to check if the user is allowed to see it. For example, to implement the HTTP endpoint `/event/{eid}`, which displays the details of an event, the developer may write the code:

```
if not curr_user.is_attending(eid):
  raise Http404("Event not found")
event = Event.find(eid)
return format_html(event)
```

Here, the if block raises an error when there is no event that the current user is attending with the requested event ID.

This approach is effective, as long as the developer is careful to put the appropriate filters and checks in every place they are needed. The problem is that if the developer makes a single mistake, a data leak may ensue. For example, if the developer forgot the access check in the `/event/{eid}` endpoint, then a user would be able to access another user's data simply by requesting an event with an arbitrary ID.

This example may look contrived, but in the real world, such mistakes happen all the time:

- Fiserv, a top provider for banking solutions, allowed a customer to view other customers' personal details by simply making an HTTP request for a notification ID belonging to someone else [124].
- The U.S. Postal Service exposed an API that let any logged-in user query the account details for any other user [75].
- OpenEMR, a medical records portal, contained a defective access check that allowed a user to access other patients' medical profiles [163].
- HotCRP, a conference management system, had a bug that leaked hidden paper tags, not on the main webpage, but in the search autocomplete dropdown [134].

In fact, such mistakes are so prevalent that "broken access control" is listed as the top web-security risk in the OWASP Top 10 [103].

To be clear, the developers are likely not being malicious in these cases. They are likely just making mistakes—mistakes that are very likely to occur in any software above a certain size.

1.3 Past Research

Sensitive-data exposure is not a new problem. There has been decades of research on access control for database applications—of which web applications is a prominent class—trying to address this problem. The foundational work in this area was laid out in the database literature [135], but solutions applying database access control to *applications* have appeared in many research communities [32, 80, 81, 90, 92, 97, 151]. These solutions typically work as follows:

1. A human writes an access-control *policy* defining what data the user is allowed to access.
2. Then, an *enforcement* mechanism ensures that the policy is respected by the application.

Unfortunately, these solutions suffer from a common drawback: They are difficult to apply to *existing web code bases*, due to two key issues:

1. **Past solutions are incompatible with today's web programming model.** Broadly speaking, past solutions fall into a few categories (which we discuss in detail in §2.2):

   View-based authorization Many databases allow administrators to define *views*, which are virtual tables denoting subsets of the database to be revealed. The user is then allowed to query only the views, not the underlying tables. But to apply this approach to an existing web application, we would have to rewrite every SQL query to use the views instead, a significant undertaking.

   Content filtering The database transparently modifies the application's query results to remove any information that the user is not allowed to see. But such modification can cause an existing application's queries to return misleading or wrong results [119,142], and can easily break the application's functionality.

   Static verification Once a developer implements the web application in a specialized language, they can use a verifier to catch access-control bugs at compile time. Again, this approach is not design to work with existing applications, which are overwhelmingly written in mainstream languages like PHP and Ruby, which are not amenable to such verification.

2. **Past solutions require writing a policy from scratch.** As we explain in §4.2.1, writing a policy for an existing code base is far from trivial, and the requirement of writing a policy from scratch is a major barrier to adoption for access control.

Thus, most prior research on sensitive-data protection targets either greenfield applications—those built from scratch using novel technology—or existing applications that are extensively rewritten. But today's web is powered by large, entrenched codebases that cannot feasibly be rewritten from the ground up. These legacy systems are the ones that people rely on daily, and arguably the ones most in need of robust data protection. Yet, they are largely overlooked by the access-control literature. It should be clear that sensitive data protection is far too urgent a problem to demand a complete rewrite of the modern web to achieve.

1.4 Our Contributions

The work presented in this dissertation has one goal:

To bring data protection to today's installed base of web applications.

Our solution consists of two systems, addressing the two issues mentioned above:

1. **Blockaid:** an access-control enforcer that is compatible with today's web programming model. Blockaid checks SQL queries issued by the application using a novel criterion called *trace determinacy*, which works with the way queries are issued by existing web applications. The key challenge is to check this criterion *fast*; to do so, we developed a generalization caching solution on top of SMT solving. We describe Blockaid in Chapter 2, and an associated theoretical result in Chapter 3.

2. **Ote:** a tool that helps humans create a policy for a legacy web application by extracting the policy embedded in its code. The main challenge is that web-application code is often written in dynamic languages, which makes it difficult to analyze. Our approach is to adopt concolic execution, a technique from software testing, which proved effective in exploring code paths through the parts of the application that affect what data is being queried. Ote then generalizes the individual queries encountered into a policy for human review. We describe Ote in Chapter 4.

In building these systems, we take a first step in providing a holistic solution for protecting sensitive data in web applications today. We point out several avenues for future research in Chapter 5.

1.5 Previous Publications

Blockaid was published at OSDI '22 [159]. Ote and our result on query determinacy were posted as arXiv preprints in 2024 [155, 157]. Some of the future directions we discuss in Chapter 5 were presented at HotOS '23 [158]. My previous work on Kappa [156] (published at SoCC '20) and Persimmon [160] (published at OSDI '20) are not included in this dissertation.

Chapter 2

Blockaid: Access-control Enforcement

2.1 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 [10, 57, 71, 72, 91, 134].

Prior work has suggested a variety of languages, frameworks, and tools that simplify the enforcement of data-access policies. As we detail in §2.2, 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).

In this chapter, we propose an alternative approach to enforcing data-access policies that meets four goals:

Backwards compatibility Applies to applications built using common existing web frameworks.

Semantic transparency Fully answers queries that comply with the policy and blocks queries that do not (rather than providing partial, and potentially misleading, results).

Policy expressiveness Supports a wide range of policies.

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 policy in Blockaid is specified using SQL view definitions defining what information that 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 and only if it *never* reveals—for any underlying dataset—more information than the views do, a well-studied property in databases called *query determinacy* [99].

While determinacy does characterize 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 data is accessible often depends on the result of previous queries in the same web request. We thus 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 unsatisifiability 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 for every query. Thus, we use a *decision cache* to record compliant queries (with traces) so that future occurrences need not be rechecked. But caching exact queries and traces would be ineffective: a query is often 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 [15, § 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* [46], Spree [133], and Autolab [29]—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 only supports 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`.

2.2 Related Work

The subject of data-access control has been studied by many. We compare our approach to prior ones along our goals (§2.1).

Static verification. Several systems have been proposed to statically verify that application code can only issue compliant queries; examples include Swift [34], SELinks [40], UrFlow [32], and Storm [81]. 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 [98] and Ur/Web [33], sacrificing compatibility with common web frameworks.

Query modification. A popular run-time approach is query modification [135]: replacing secret values returned by a query with placeholders (or dropping any rows containing secrets). This is implemented in commercial databases [24,93] and academic works like Hippocratic databases [6], Jacqueline [151], Qapla [92], and multiverse databases [90]. 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 [59, 119, 142].

Furthermore, query modification mechanisms typically use row- and cell-level policies (e.g., SQL Server RLS and DDM, Oracle VPD). As we discuss in §2.9, this row-/cell-level format is less expressive than Blockaid's view-based scheme.

View-based access control. Many databases allow administrators to create views and grants 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 regular web programming, as programmers must now sort out which views grant them information for each query. In contrast, Blockaid allows queries to be issued against the base tables directly.

While some prior work has studied view-based compliance of queries issued against base tables [18, 19], they only check single queries, while Blockaid checks a query in the context of a trace, a crucial feature for supporting web applications.

2.3 System Design

2.3.1 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 (§2.3.2).

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 [5, 18, 19, 23, 60, 80, 90, 119, 130, 135, 142]). Ensuring the integrity of updates, while important, is orthogonal to our goal and is left to future work.

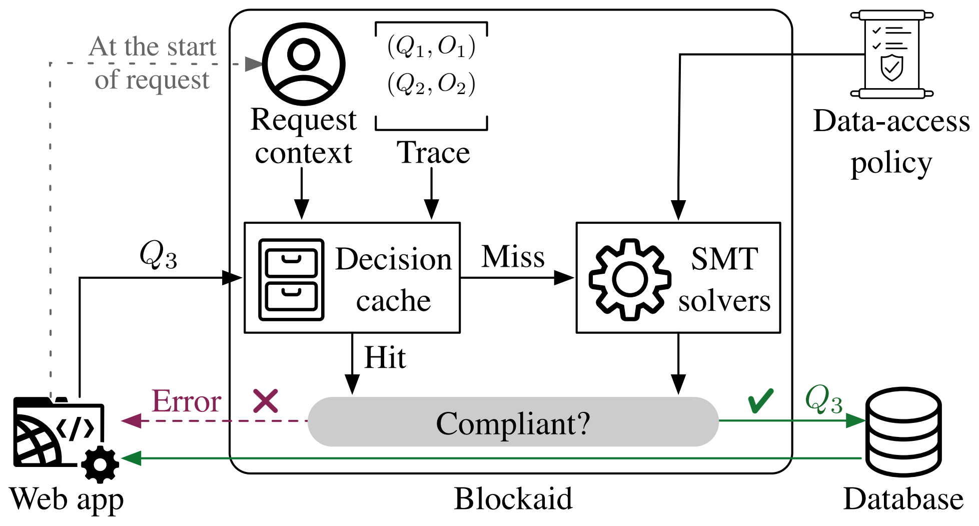


Figure 2.1: An overview of Blockaid (for a single web request).

2.3.2 System Overview

Blockaid is a SQL proxy that sits between the application and the database (Figure 2.1). It takes as input (1) a database schema (including constraints), and (2) a data-access policy specified as database views (§2.4), 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 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 information 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 (§2.5) and checks its satisfiability using several SMT solvers in parallel (§2.7).

If a query is compliant, Blockaid forwards it to the database unmodified. In case of a cache miss, it also extracts and caches a decision template (§2.6). 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 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 check 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 §2.4.3). But it may reject certain behaviors that do not violate the policy (§2.9), although we never encountered this in our evaluation.

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.

2.3.3 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.

2.4 View-based Policy and Compliance

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

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

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

Listing 2.1: 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.

2.4.1 Specifying Policies as Views

A policy is a collection of SQL queries that, together, define what information a user is allowed access. Each query is called a *view definition* and can refer to parameters from the request context. As an example, Listing 2.1 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.

2.4.2 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 2.1. Let $\mathit{MyUId} = 2$. 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 2.1, this query can always be answered using V_4, which reveals the *UId* of everyone whom the user attends an event with, and 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 from the trace of prior queries and results in the same web request.

Example 2.2. Again, let *MyUId* $= 2$. Consider the following sequence of queries issued while handling a 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 were issued in isolation.

Example 2.3. 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 [59, 73, 162] where a policy enforcer can allow a query after inspecting additional information in the database that has not been fetched by the application. ◀

Definition 2.4. 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 2.5. 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:

$$V^{ctx}(D_1) = V^{ctx}(D_2), \qquad (\forall V \in \mathcal{V}) \tag{2.1}$$

$$Q_i(D_1) = O_i, \qquad (\forall 1 \leq i \leq n) \tag{2.2}$$

$$Q_i(D_2) = O_i, \qquad (\forall 1 \leq i \leq n) \tag{2.3}$$

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

We call Definition 2.5 *trace determinacy* as it extends the classic notion of query determinacy [99, 126] with the trace. Query determinacy is undecidable even for conjunctive views and queries [54, 55]; trace determinacy must also be undecidable in the same scenario. Although several decidable cases have been discovered for query determinacy [2, 99, 105], 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.

2.4.3 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 2.5) 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 to be influenced by inaccessible information.
2. **Necessity**: Any enforcement system that makes per-query decisions based solely on the query and the trace *cannot safely allow any non-compliant query* without the risk of the application revealing inaccessible information.

[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.

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 $\mathcal{P}(\mathit{ctx}, \mathit{req}, D)$ 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 [39, 56] stating that its output depends only on the inputs that the user has access to—namely, ctx, req, and $V^{\mathit{ctx}}(D)$ for each $V \in \mathcal{V}$. The formal definition follows from a similar intuition as Definition 2.5.

Definition 2.6. A program $\mathcal{P}$ satisfies *noninterference* under policy $\mathcal{V}$ if the following condition holds:

$$\begin{aligned}\mathrm{NI}_{\mathcal{V}}(\mathcal{P}) \quad \triangleq \quad & \forall \mathit{ctx}, \mathit{req}, D_1, D_2. \\ & \left[\forall V \in \mathcal{V}.\, V^{\mathit{ctx}}(D_1) = V^{\mathit{ctx}}(D_2)\right] \\ & \qquad \implies \mathcal{P}(\mathit{ctx}, \mathit{req}, D_1) = \mathcal{P}(\mathit{ctx}, \mathit{req}, D_2).\end{aligned}$$

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

Definition 2.7. An *enforcement predicate* is a mapping from a request context, a query, and a trace to an allow/block decision:

$$E(\mathit{ctx}, Q, \mathcal{T}) \to \{\checkmark, \text{✗}\}.$$

Definition 2.8. Let $\mathcal{P}(\mathit{ctx}, \mathit{req}, D)$ be a program and E be an enforcement predicate. We define the program $\mathcal{P}$ *under enforcement using* E as a new program $\mathcal{P}^E(\mathit{ctx}, \mathit{req}, D)$ 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 $E(\mathit{ctx}, Q, \mathcal{T}) = \text{✗}$ by immediately returning an error.

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

Definition 2.9. 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 $1 \le i \le n$,

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

Definition 2.10. 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. Like before, we use colors to link statements to their explanation.

[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 [62, § 2].

Theorem 2.11. Let $\mathcal{V}$ be a set of views and E be a predicate.

1. Suppose $E(ctx, Q, \mathcal{T}) = ✓$ 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 $E(ctx, Q, \mathcal{T}) = ✓$, Q is ctx-compliant to $\mathcal{V}$ given $\mathcal{T}$.

To unpack, Theorem 2.11 says: (1) as long as an enforcement predicate ensures query compliance, it correctly enforces the policy on an application (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.

Proof of Theorem 2.11. We prove the two parts separately.

Part 1. Suppose $E(ctx, Q, \mathcal{T}) = ✓$ 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 allow us to conclude 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 $E(ctx, Q, \mathcal{T}) = ✗$, 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.

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 $E(ctx, Q, \mathcal{T}) = ✓$. 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
        end for
        return $Q(D)$
    end procedure
```

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

$$V^{ctx}(D_1) = V^{ctx}(D_2), \qquad (\forall V \in \mathcal{V}) \tag{2.4}$$

$$Q_i(D_1) = O_i, \qquad (\forall 1 \leq i \leq n) \tag{2.5}$$

$$Q_i(D_2) = O_i. \qquad (\forall 1 \leq i \leq n) \tag{2.6}$$

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 (2.4) 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 $\mathcal{T}$. $\square$

2.5 Compliance Checking with SMT

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

2.5.1 Translating Noncompliance to SMT

Blockaid verifies query compliance by framing *noncompliance* (i.e, the negation of Definition 2.5) 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 [38], 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 2.12. 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 $A^D(\cdot,\cdot)$ be FOL predicates representing the *Events* and *Attendances* table in the database D in:

$$\begin{aligned}\mathsf{Q}^D(\mathsf{x}_e,\mathsf{x}_t) \;&\triangleq\; \exists x_d, x_u, x'_e, x_c.\, E^D(\mathsf{x}_e,\mathsf{x}_t,x_d) \wedge A^D(x_u,x'_e,x_c)\\ &\wedge \mathsf{x}_e = x'_e \wedge x_u = 2.\end{aligned}$$

$\mathsf{Q}^D(\mathsf{x}_e,\mathsf{x}_t)$ encodes the statement $(\mathsf{x}_e,\mathsf{x}_t) \in Q(D)$, i.e., 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 2.13. We now present the noncompliance formula for a single query Q with respect to $\mathcal{V}$ from §2.4.1. 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 is 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. We can similarly encode database constraints and consistency with a trace. ◀

2.5.2 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 translation 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 aggregations 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 [122] and Django [48] add a primary key for every table. Given this assumption, Blockaid rewrites complex SQL into *basic queries* that map directly to relational algebra.

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

Basic SQL Queries

Definition 2.14. 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` 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:

- 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 $(\mathit{UId}, \mathit{EId})$.

In our experience, policy views can typically be written as basic queries directly—e.g., for Listing 2.1 we can frame V_3 and V_4 as equivalent basic queries by replacing subqueries with joins and using the inner join transformation from §2.5.2.

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
```

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

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

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. We add any `ORDER BY` column as an output column and then discard the `ORDER BY` clause. We also discard any `LIMIT` clause but, when adding this query to the trace, use a modified condition $O_i \subseteq D(Q_i)$ (instead of "$=$") to indicate that Blockaid may have observed a partial result.

Aggregations. We rewrite `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)
```

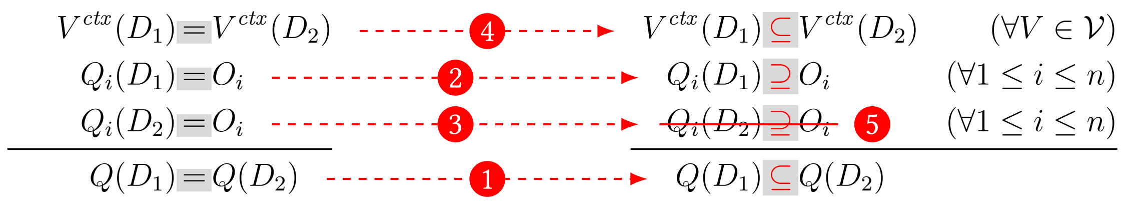
where `C3` is obtained by replacing each occurrence of `B.?` with `NULL` in `C2` and simplifying the resulting predicate.[6] The first sub-query covers the rows in `A` with at least one match in `B`, and the second sub-query covers those with no matches.

Feature not supported. The SQL features that we do not support include `GROUP BY`, `ANY`, `EXISTS`, etc., although they can also be formulated / approximated using basic queries. In the future we also plan to leverage other formalisms that model complex SQL semantics more precisely [31, 35, 36, 139–141, 144].

2.5.3 Optimizations and SMT Encoding

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

[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.


$V^{ctx}(D_1) = V^{ctx}(D_2)$
4
$V^{ctx}(D_1) \subseteq V^{ctx}(D_2)$
$(\forall V \in \mathcal{V})$
$Q_i(D_1) = O_i$
2
$Q_i(D_1) \supseteq O_i$
$(\forall 1 \le i \le n)$
$Q_i(D_2) = O_i$
3
~~$Q_i(D_2) \supseteq O_i$~~
5
$(\forall 1 \le i \le n)$
$Q(D_1) = Q(D_2)$
1
$Q(D_1) \subseteq Q(D_2)$

(a) Compliance (Definition 2.5). (b) Strong compliance (Definition 2.15).

Figure 2.2: **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.

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

Definition 2.15 (Strong compliance)**.** 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:

$$V^{ctx}(D_1) \subseteq V^{ctx}(D_2), \qquad (\forall V \in \mathcal{V}) \tag{2.7}$$

$$Q_i(D_1) \supseteq O_i, \qquad (\forall 1 \le i \le n) \tag{2.8}$$

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

Theorem 2.16. If Q is strongly compliant to $\mathcal{V}$ given trace $\mathcal{T} = \{(Q_i, O_i)\}_{i=1}^n$, then Q is 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 (2.1) to (2.3) from the compliance definition. These imply the strong compliance assumptions (Equations (2.7) and (2.8)), 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 $\mathcal{T}$. $\square$

For faster checking, Blockaid verifies strong compliance rather than compliance; by Theorem 2.16, soundness is preserved. However, there are scenarios where a query is compliant but *not* strongly compliant (see Remark 2.17); 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.

Remark 2.17. To understand when strong compliance fails to coincide with compliance, let us look at Figure 2.2, which illustrates how we modified the definition of compliance (Definition 2.5) into that of strong compliance (Definition 2.15) 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) [115], 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 (§2.6.3) 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 [99, 126], 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. [99, § 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).

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.

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 use a trace-pruning heuristic: when checking a query Q, look for any previous query has returned over ten rows, and only keep rows that contain the first occurrence of a primary-key value (e.g., user ID) that appears 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 function with a transitivity axiom.

NULLs. We model `NULL` using a two-valued semantics of SQL [58, § 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: $x = y \wedge x \neq \mathit{null} \wedge y \neq \mathit{null}$.

2.6 Decision Generalization and Caching

While SMT solvers can verify 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*.

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

Listing 2.2: An example query with trace from the calendar application and a decision template generated from it.

(a) Example query with trace ($\mathit{UId} = 1$).

```
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
```

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 approach is unlikely to be effective because the number of distinct $(Q, \mathcal{T})$ pairs can be unbounded. For example, the application can issue as many queries of the form:

```
SELECT * FROM Users WHERE UId = ?
```

as there are users in the system. 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.

2.6.1 Example

Suppose a user with $\mathit{UId} = 1$ requests Event #42 in the calendar application, resulting in the application issuing a sequence of SQL queries. Consider the third query, shown in Listing 2.2a. As we explained in Example 2.2, 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 2.2b 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 define what a decision template is, what it means for a template to have a match, and what makes a "good" template.

2.6.2 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 2.15), we no longer take into account the *absence* of a returned row.

Definition 2.18. 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 $1 \le i \le n$.

Definition 2.19. 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 obvious 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 2.20. 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 $\nu(y)$.

Definition 2.21. 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 2.22. Listing 2.2b 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 $\mathbf{c} = (\mathit{MyUId})$; $Q_\mathcal{D}$ 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 $\mathit{MyUId} = 1$, this template matches the query and trace in Listing 2.2a via the valuation $\{x_0 \mapsto 42, x_1 \mapsto$ `"05/04 1pm"`, $\mathit{MyUId} \mapsto 1\}$. ◀

We are interested only in templates that imply compliance.

Definition 2.23. 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 verifies that a template is sound via the following theorem derived from strong compliance (Definition 2.15):

Theorem 2.24. 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.

[8]Technically, this template requires $\mathit{MyUId} \neq$ `NULL` $\wedge\, x_0 \neq$ `NULL`. We omitted this condition in Listing 2.2b because we assume the user ID parameter and the *Attendances* table's *EId* column are both non-NULL.

Definition 2.25. 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 $(Q, \mathcal{T})$.

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.

2.6.3 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 (§2.6.3).
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 (§2.6.3).

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 2.2a 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 2.26. $\mathcal{D}_1$ 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* [15, § 11.8]—a subset of clauses in the formula that remains contradictory even when all other clauses are 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:

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

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 $LQ_i \notin S$.

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 [65] 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 can be evaluated via an extension of the relational algebra operators [65, § 7]. This allows queries to be translated into SMT without using quantifiers or relation symbols. For example, the query `SELECT Name FROM Users WHERE UId = 5` can be written as:

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

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

After Blockaid generates an unsat core as described in §2.6.3, 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;[10] 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 regular, unbounded formula, and if this fails, increments the table bounds and retry.

[9]For example, finite model finders in CVC4 [117] and Vampire [113] often time out or run out of memory on tables with only tens of columns.

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

Listing 2.3: Parameterization and candidate atoms for Listing 2.2a.

(a) Parameterized trace $\mathcal{T}^{\mathrm{p}}_{\mathrm{min}}$ 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`:	Form `x = x'`:	Form `x < x'`:
• `MyUId = 1`	• `MyUId = x0`	• `MyUId < x1`
• `x0 = 1`	• `x1 = x3`	• `MyUId < x3`
• `x1 = 42`		• `x0 < x1`
• `x2 = "05/04 1pm"`		• `x0 < x3`
• `x3 = 42`		

Step Two: Find Value Constraints

Taking the template $\mathcal{D}_1 = (Q, \mathcal{T}_{\mathrm{min}}, \top)$ from Step 1, Blockaid generalizes it further by abstracting away the constants. To do so, Blockaid *parameterizes* $\mathcal{T}_{\mathrm{min}}$ 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 2.3a shows $\mathcal{T}^{\mathrm{p}}_{\mathrm{min}}$ 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.

Blockaid must now generate a condition Φ such that the resulting template $\mathcal{D}_2 = (Q^{\mathrm{p}}, \mathcal{T}^{\mathrm{p}}_{\mathrm{min}}, \Phi)$ 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 2.27. The set of *candidate atoms* is defined as:

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

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

Definition 2.28. With respect to Q^{p} and $\mathcal{T}^{\mathrm{p}}_{\mathrm{min}}$, a subset of atoms $C_0 \subseteq C$ is *sound* if the decision template $(Q^{\mathrm{p}}, \mathcal{T}^{\mathrm{p}}_{\mathrm{min}}, \bigwedge C_0)$ is sound. ($\bigwedge C_0$ denotes the conjunction of atoms in C_0.)

Definition 2.29. 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 $C_2 \npreceq C_1$.

Example 2.30. Listing 2.3b shows all the candidate atoms from Listing 2.3a (after omitting symmetric ones in the `x = x'` group). Consider the following two subsets of atoms:

$$\begin{aligned} C_1 &= \left\{\texttt{MyUId = x0},\ \ \texttt{x1 = 42},\ \ \texttt{x3 = 42}\right\}, \\ C_2 &= \left\{\texttt{MyUId = x0},\ \ \texttt{x1 = x3}\right\}. \end{aligned}$$

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 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:

$$\begin{array}{lcr} & V^{ctx}(D_1) \subseteq V^{ctx}(D_2), & (\forall V \in \mathcal{V}) \\ & t^{\mathrm{p}}_i \in Q^{\mathrm{p}}_i(D_1), & (\forall (t^{\mathrm{p}}_i, Q^{\mathrm{p}}_i) \in \mathcal{T}^{\mathrm{p}}_{\mathrm{min}}) \\ [LC_i] & c_i, & (\forall c_i \in C) \\ & Q^{\mathrm{p}}(D_1) \nsubseteq Q^{\mathrm{p}}(D_2). & \end{array}$$

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

Second, Blockaid augments C_{core} with other atoms that are implied by it:

$$C_{\mathrm{aug}} \ \triangleq\ \left\{ c \in C \,\middle|\, \bigwedge C_{\mathrm{core}} \implies c \right\}.$$

In our example,

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

C_{aug} enjoys a closure property: if $C_0 \subseteq C_{\mathrm{aug}}$ and $C_0 \preceq C_1$, then $C_1 \subseteq C_{\mathrm{aug}}$. 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 [85, 86, 110] 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 $C_{\mathrm{small}} = \{\,\texttt{MyUId=x0}, \texttt{x1=x3}\,\}$ of cardinality two—which is also a maximally weak subset—even though this might not be the case in general.[11] Nevertheless, searching for a smallest sound subset has produced templates that generalize well in practice.

Blockaid thus produces the decision template:

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

Proposition 2.31. $\mathcal{D}_2$ is sound, matches $(Q, \mathcal{T})$, and is at least as general as $\mathcal{D}_1$.

As an optimization, whenever $\bigwedge C_{\mathrm{small}} \implies x = y$ for $x, y \in \mathbf{x} \cup \mathbf{c}$, Blockaid replaces x with y in the template. This is how, e.g., in Listing 2.2b `?0` appears in both the trace and the query.

Optimizations

We implement two optimizations that improve the performance of template generation and the generalizability 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.[12] 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 sub-queries 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 sub-queries q'_i could match the template from q_1.

2.6.4 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.

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

[12]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.

2.7 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 [17] and caches parser outputs. To check compliance, it uses Z3's Java binding [43] to generate formulas in SMT-LIB 2 format [14] and invokes an ensemble of solvers in parallel. Our ensemble consists of Z3 [44] (v4.8.12) and cvc5 [13] (v0.3) using default configurations, and Vampire [74] (v4.6.1) using six configurations from its CASC portfolio.[13] 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 (§2.6.3), 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`.

2.8 Evaluation

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

- **diaspora*** [46]: a social network with 850 k users.
- **Spree** [133]: an e-commerce app used by 50+ businesses.
- **Autolab** [29]: 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 generalize to other entities (users, etc.); and no query was falsely rejected in our benchmark. Reproduction instructions can be found in §2.8.8.

2.8.1 Constraints, Policies, and Annotations

Table 2.1 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

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

Table 2.1: 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

diaspora*). Spree and Autolab use the Rails cache, and we annotate their cache key patterns with queries (§2.3.2).

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.

2.8.2 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.

Table 2.2: **Application benchmark description and page-load time.** 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. The abbreviation "w/" stands for "with".

			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 w/ 30 votes & 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 a profile (basic info & 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	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 & status of a prior 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 course summary (15 assignments).	84 / 96	87 / 101	97 / 116	3.9 s / 4.1 s
Assignment	A4	View a quiz (w/ 3 submissions & grades).	97 / 110	103 / 118	115 / 138	3.5 s / 3.6 s
Submission	A5	Download a 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

2. **Fetch less data**: We modify some queries to not fetch potentially sensitive data unless it will be revealed to the user; some of these changes use the `lazy_column` gem [89].

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 [107] 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 (§2.3.2).

The code changes are summarized also in Table 2.1, which omits configuration changes, adaptations for JRuby, and experiment code. The changes range from 19 to 96 lines of code.

2.8.3 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, we run the applications using JRuby [68] (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.[14] 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 2.2). 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.

2.8.4 Page Load Times

We start by measuring the *page load time* (PLT) using a headless Chrome browser (v96) driven by Selenium [132]. The PLT is reported as the time elapsed between `navigationStart` and `loadEventEnd` as defined by the `PerformanceTiming` interface [145]. The one exception is the

[14]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 §2.6.3.

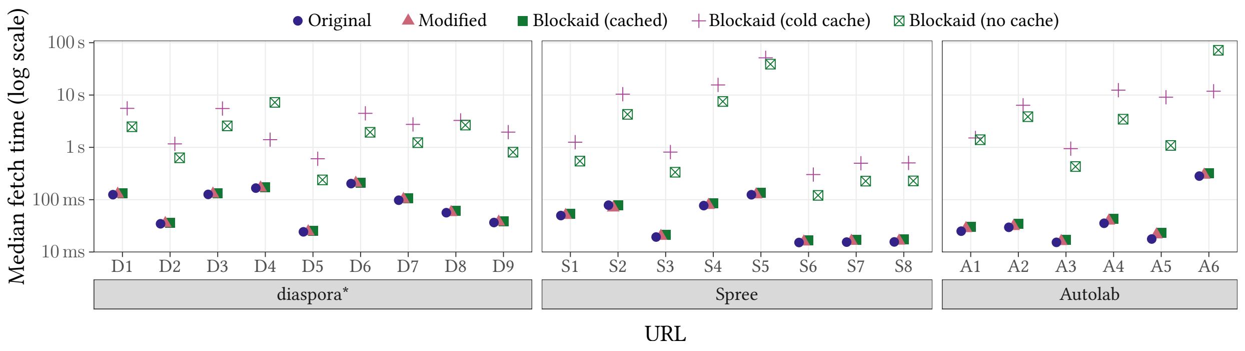


Figure 2.3: **URL fetch latency (median).** With all compliance decisions cached, Blockaid incurs up to 10 % overhead over “modified”.

Autolab “Submission” page, 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 §2.8.2), *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 2.2 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.

2.8.5 Fetch Latency

To better understand page load performance, we separate out the individual URLs fetched by each page (Table 2.2), omitting URLs for assets, and measure the latency of fetching each URL (not including rendering time). The median latencies are shown in Figure 2.3. In addition to the four settings from §2.8.4, 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. As a

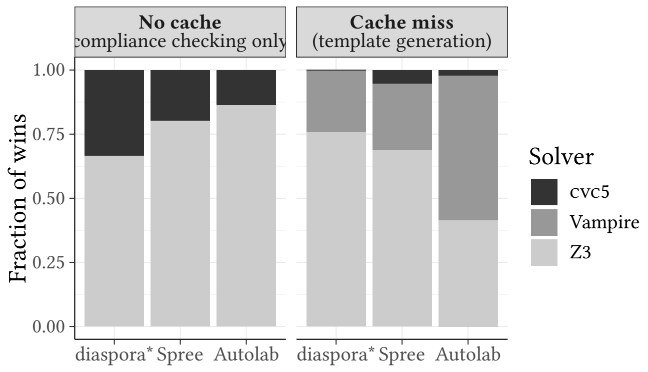


Figure 2.4: **Fraction of wins by each solver.** "Vampire" covers a portfolio of six configurations (§2.7).

result, 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.

2.8.6 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 small enough unsat core (§2.7), assuming the query is compliant.

Figure 2.4 shows the fraction of wins by each solver in the two scenarios. In the no-cache case, 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—Z3 and cvc5 often finish quickly but with large unsat cores, causing Blockaid to wait till Vampire produces a smaller core.

2.8.7 Template Generalization

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

Listing 2.4: **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
```

To illustrate how Blockaid might produce a template that does not generalize fully, consider a query from Spree's cache key annotations (Listing 2.4). This query fetches assets for product variants in the user's order; here, a variant's asset belongs to its product's "master variant". Listing 2.4a 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 only applies when an order has exactly three variants. The `IN`-splitting optimization from §2.6.3 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 Spree defines 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.[15] 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 a variant, Blockaid produces the template in Listing 2.4b, 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 manually inspecting templates is not required for using Blockaid, doing so can help debug overly broad policies, whose undesired consequences are often exposed by the general decision templates produced by Blockaid.

2.8.8 Artifact

We have open-sourced our artifact, which includes:

- Our implementation of Blockaid, which is compatible with applications that can run atop the JVM and connect to a database via JDBC (§2.7);
- The three applications we used for our evaluation—modified according to §2.8.2—as well as the policy we wrote for each; and,
- A setup for reproducing the evaluation results from §2.8.

Scope This artifact can be used to run the main experiments from this chapter: the page load time (PLT) measurements (§2.8.4) and the fetch latency measurements (§2.8.5 and §2.8.6) on the

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

Table 2.3: 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 (§2.8.2).

three applications. From these experiments, it generates Table 2.2 (with URLs and descriptions omitted), Figure 2.3, and Figure 2.4. 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 (§2.3.3), run atop the JVM, and connect to the database using JDBC (§2.7).

Contents This artifact consists of our Blockaid implementation, the three applications used in our evaluation (with modifications described in §2.8.2), the data-access policy we wrote for each, and scripts and virtual machine image for running the experiments.

Hosting See Table 2.3.

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).

2.9 Additional Issues

2.9.1 Comparison to row- and cell-level policy

Several commercial databases (such as SQL Server [93] and Oracle [102]) 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*.

2.9.2 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.

2.9.3 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 traces to Blockaid to be checked asynchronously, further reducing Blockaid's performance impact.

2.9.4 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 reveals 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 2.3:

```
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 §2.4.3) becomes *instance-based determinacy* [73, 119, 162], 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. Such 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.

2.9.5 Optimal templates

While decision templates produced by Blockaid are general enough in practice, they might not be *maximally general* among the templates that match the query and trace being checked. For one thing, the template condition might not be maximally weak (§2.6.3). 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.

2.10 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.

Chapter 3

A Decidable Case of Query Determinacy: Project-Select Views

3.1 Introduction

To enforce a view-based access-control policy, Blockaid checks trace determinacy (Definition 2.5), a property on views and queries which generalizes *query determinacy* under set semantics [159, § 4.2]. A set $\mathbf{V}$ of queries (which we will call "views") *determines* a query Q if and only if $\mathbf{V}(\mathbf{I}) = \mathbf{V}(\mathbf{I}')$ implies $Q(\mathbf{I}) = Q(\mathbf{I}')$ [99].[1]

Checking query determinacy is a hard problem—it is undecidable even for CQ (conjunctive query) views and CQ queries [54, 55]. It *is* shown to be decidable in simpler scenarios—for example, (1) for MCQ (monadic-conjunctive-query) views and CQ queries [99, Theorem 5.16], and (2) for a single path-query view and a CQ query [99, Theorem 5.20]. But these decidability results are too limited to be applied to view-based access-control for practical web applications.

Here, we discuss another case where query determinacy is decidable: for select-project views, and a select-project-join query with no self joins—as long as the selection predicates are not too complex. To be clear, this result is still quite limited, for the simple reason that real-world views often have joins. But it is a step forward in our search for a larger class of views and queries, encompassing more real-world use cases, for which query determinacy is decidable.

3.2 Setup

We consider all queries under set semantics, and we assume that all select-project-join queries are put into normal form [1, Proposition 4.4.2].

Fix a database schema consisting of relation names $R_1, R_2, \ldots, R_m$, and let $\mathbf{V}$ be a set of views. Since we're dealing only with project-select views, we partition $\mathbf{V}$ into $\mathbf{V}_1, \mathbf{V}_2, \ldots, \mathbf{V}_m$,

[1]We may think of $\mathbf{V}(I)$ as a database instance that, for each $V \in \mathbf{V}$, maps the relation name V to the relation $V(\mathbf{I})$. Then the formula $\mathbf{V}(\mathbf{I}) = \mathbf{V}(\mathbf{I}')$ means $\forall V \in \mathbf{V} : V(\mathbf{I}) = V(\mathbf{I}')$.

Table 3.1: Database notations.

m	Number of relations in the database schema
R_i	Name of relation
$\mathbf{I}$, $\mathbf{I}'$	Database instance (a mapping from relation names to relations)
$V_{i,j}$, Q	Query
$\mathbf{V}$, $\mathbf{V}_i$	Set of queries
U, $U_{i,j}$	Set of column names
θ, $\theta_{i,j}$	Predicate (used in selections)
t	Database tuple (a mapping from column names to values)
t_i, $t_{i,j}$	Tuple in relation R_i
$t[U]$	Sub-tuple $\{A \mapsto t(A) : A \in U\}$

Table 3.2: Other mathematical notations.

$\mathbf{t}$	Sequence of tuples $\langle t_1, \ldots, t_{\lvert\mathbf{t}\rvert} \rangle$
$\mathbf{t}_{m..n}$	Sub-sequence of tuples $\langle t_m, t_{m+1}, \ldots, t_n \rangle$
$\mathbf{t} \circ \mathbf{s}$	Concatenation of the sequence $\mathbf{t}$ and the sequence $\mathbf{s}$
$\mathbf{t}$ EXCEPT $i \mapsto s$	Sequence of tuples $\langle t_1, \ldots, t_{i-1}, s, t_{i+1}, \ldots, t_{\lvert\mathbf{t}\rvert} \rangle$

where each $\mathbf{V}_i$ consists of some number n_i of queries that refer only to the relation R_i. In other words, we denote:

$$\begin{aligned} \mathbf{V} &\triangleq \mathbf{V}_1 \cup \mathbf{V}_2 \cup \cdots \mathbf{V}_m, \\ \mathbf{V}_i &\triangleq \{V_{i,j} : 1 \leq j \leq n_i\}, && (1 \leq i \leq m) \\ V_{i,j} &\triangleq \pi_{U_{i,j}} \sigma_{\theta_{i,j}} R_i. && (1 \leq i \leq m, 1 \leq j \leq n_i) \end{aligned} \tag{3.1}$$

And let query Q be a project-select-join query with no self joins:

$$Q \triangleq \pi_U \sigma_\theta (R_1 \times R_2 \times \cdots \times R_m). \tag{3.2}$$

A summary of notations is found in Tables 3.1 and 3.2. For visual clarity, we will use lists bulleted by "$\wedge$" to denote the conjunction of a number of formulas [77].

3.3 Reducing determinacy to a logical formula

3.3.1 Statement of Theorem

In this setting, checking whether $\mathbf{V}$ determines Q can be reduced to checking the satisfiability of a logical formula.

Theorem 3.1. The set $\mathbf{V}$ of views determines query Q iff for every $1 \leq i \leq m$:

$$\forall\, \mathbf{t} : \theta(\mathbf{t}) \Rightarrow \bigvee_{j=1}^{n_i} (\theta_{i,j}(t_i) \wedge \forall\, t'_i : \Phi_{i,j}(t_i, t'_i) \Rightarrow \Psi_{i,j}(\mathbf{t}, t'_i)) \qquad (\star)$$

where the sub-formulas Φ and Ψ are defined as:

$$\begin{array}{rcl} \Phi_{i,j}(t_i, t'_i) & \triangleq & \wedge\ \theta_{i,j}(t'_i) \\ & & \wedge\ t'_i[U_{i,j}] = t_i[U_{i,j}], \\ \Psi_{i,j}(\mathbf{t}, t'_i) & \triangleq & \text{LET}\ \ \mathbf{s} \ \triangleq\ \mathbf{t}\ \text{EXCEPT}\ i \mapsto t'_i \\ & & \text{IN}\ \ \ \wedge\ \theta(\mathbf{s}) \\ & & \qquad \wedge\ \mathbf{s}[U] = \mathbf{t}[U]. \end{array}$$

Furthermore, finite and unrestricted determinacy coincide in this setting.

We present its proof in §3.3.2.

As a result of this theorem, for project-select views and a project-select-join query without self joins, we can check query determinacy by checking the validity of ($\star$) for every $1 \leq i \leq m$. That is, query determinacy is decidable as long as the validity of ($\star$) is decidable.

Observe that ($\star$) is constructed from selection predicates θ and $\theta_{i,j}$, equality atoms, and propositional connectives.

Corollary 3.2. For project-select views and a project-select-join query without self joins, if all selection predicates in the views and the query are in a first-order theory for which validity is decidable, then query determinacy is decidable.

For example, if the selection predicates consist only of equalities over variables and propositional connectives [22, § 9], then query determinacy is decidable. A straightforward way to check determinacy is by encoding ($\star$) into an SMT formula and calling an SMT solver. What's more, if the selection predicates are quantifier-free (which they typically are in practice), then the negation of ($\star$) is purely existentially quantified, so we can check determinacy simply by checking the satisfiability of a formula that is quantifier-free.

3.3.2 Proof of Theorem

Theorem 3.1 is somewhat tedious to prove. In an attempt to avoid mistakes and make the proof easier to check, we write the proof in the *hierarchically structured style* [79]. Explaining this style, Lamport [78, Appendix A] writes:

> A structured proof consists of a sequence of statements and their proofs; each of those proofs is either a structured proof or an ordinary paragraph-style proof. The j^{th} step in the current level i proof is numbered $\langle i \rangle j$. […] We recommend reading the proofs hierarchically, from the top level down. To read the proof of a long level i step, first read the level $i + 1$ statements that form its proof, together with the proof of the final Q.E.D. step (which is usually a short paragraph).

Proof of Theorem 3.1. By structured proof:

$\langle 1 \rangle 1.$ ASSUME: ($\star$) holds for every $1 \leq i \leq m$.
PROVE: $\mathbf{V}$ determines Q.

$\langle 2 \rangle 1.$ SUFFICES ASSUME: 1. NEW $(\mathbf{I}, \mathbf{I}')$
2. $\mathbf{V}(\mathbf{I}) = \mathbf{V}(\mathbf{I}')$
3. NEW $s \in Q(\mathbf{I})$
PROVE: $s \in Q(\mathbf{I}')$

PROOF: By definition of query determinacy, and by symmetry between $\mathbf{I}$ and $\mathbf{I}'$.

$\langle 2 \rangle 2.$ Choose $\mathbf{t}$ such that:

1. $\theta(\mathbf{t})$ holds,
2. $\mathbf{t}[U] = s$,
3. $t_i \in \mathbf{I}(R_i)$ for all $1 \leq i \leq m$.

PROOF: Such $\mathbf{t}$ exists by $\langle 2 \rangle 1.3$, and by definition of Q (3.2).

$\langle 2 \rangle 3.$ For every $0 \leq \ell \leq m$, there exists a sequence $\mathbf{t}^\ell$ of ℓ tuples, such that:

C1. $t_i^\ell \in \mathbf{I}'(R_i)$ for every $1 \leq i \leq \ell$,

C2. $\theta(\mathbf{t}^\ell \circ \mathbf{t}_{\ell+1..m})$ holds,

C3. $(\mathbf{t}^\ell \circ \mathbf{t}_{\ell+1..m})[U] = s$.

We will proceed by induction on ℓ.

$\langle 3 \rangle 1.$ There exists $\mathbf{t}^0$ which satisfies C1–C3 for $\ell = 0$.

PROOF: Take $\mathbf{t}^0$ to be the empty sequence. C1 holds vacuously; C2 follows from $\langle 2 \rangle 2.1$; and C3 follows from $\langle 2 \rangle 2.2$.

$\langle 3 \rangle 2.$ ASSUME: NEW $\mathbf{t}^{r-1}$ $(0 < r \leq m)$, $\mathbf{t}^{r-1}$ satisfies C1–C3 for $\ell = r - 1$.
PROVE: There exists $\mathbf{t}^r$ which satisfies C1–C3 for $\ell = r$.

$\langle 4 \rangle 1.$ There exists $1 \leq j \leq n_r$ such that:

$$\wedge\, \theta_{r,j}(t_r) \tag{3.3}$$

$$\wedge\, \forall\, t'_r : \Phi_{r,j}(t_r, t'_r) \Rightarrow \Psi_{r,j}\left((\mathbf{t}^{r-1} \circ \mathbf{t}_{r..m}), t'_r\right). \tag{3.4}$$

PROOF: By taking ($\star$) with $i = r$, instantiating the outermost universal quantifier using $\mathbf{t}^{r-1} \circ \mathbf{t}_{r..m}$, and applying C2 with $\ell = r - 1$.

$\langle 4 \rangle 2.$ $t_r[U_{r,j}] \in V_{r,j}(\mathbf{I})$

PROOF: $t_r \in \mathbf{I}(R_r)$ by $\langle 2 \rangle 2.3$; by (3.3); and by definition of $V_{r,j}$.

$\langle 4 \rangle 3.$ $t_r[U_{r,j}] \in V_{r,j}(\mathbf{I}')$

PROOF: By $\langle 4 \rangle 2$, noting that $V_{r,j}(\mathbf{I}) = V_{r,j}(\mathbf{I}')$ by $\langle 2 \rangle 1.2$.

$\langle 4 \rangle 4.$ There exists tuple $t'_r \in \mathbf{I}'(R_r)$ such that $\Phi_{r,j}(t_r, t'_r)$ holds.

PROOF: By $\langle 4 \rangle 3$, plugging in the definition of $V_{r,j}$ (3.1), there exists $t'_r \in \mathbf{I}'(R_r)$ such that:

$$\begin{aligned} &\wedge\, \theta_{r,j}(t'_r) \\ &\wedge\, t'_r[U_{r,j}] = t_r[U_{r,j}] \end{aligned}$$

which implies $\Phi_{r,j}(t_r, t'_r)$.

$\langle 4 \rangle 5.$ $\Psi_{r,j}\left((\mathbf{t}^{r-1} \circ \mathbf{t}_{r..m}), t'_r\right)$ holds.

PROOF: Instantiate universal quantifier in (3.4) with t'_r from $\langle 4 \rangle 4$.

DEFINE: $\mathbf{t}^r \ \triangleq\ \mathbf{t}^{r-1} \circ \langle t'_r \rangle$

$\langle 4\rangle 6$. $\mathbf{t}^r$ satisfies C1 for $\ell = r$.

PROOF: From $\langle 3\rangle 2$, $\mathbf{t}^{r-1}$ satisfies C1 for $\ell = r-1$; from $\langle 4\rangle 4$, $t'_r \in \mathbf{I}'(R_r)$.

$\langle 4\rangle 7$. $\mathbf{t}^r$ satisfies C2 for $\ell = r$.

PROOF: By $\langle 4\rangle 5$, plugging in definition for Ψ.

$\langle 4\rangle 8$. $\mathbf{t}^r$ satisfies C3 for $\ell = r$.

PROOF: By $\langle 4\rangle 5$, plugging in definition for Ψ; by $\langle 2\rangle 2.2$, $\mathbf{t}[U] = s$.

$\langle 4\rangle 9$. Q.E.D.

PROOF: By $\langle 4\rangle 6$, $\langle 4\rangle 7$, and $\langle 4\rangle 8$.

$\langle 3\rangle 3$. Q.E.D.

PROOF: By induction on ℓ, with $\langle 3\rangle 1$ as the base case and $\langle 3\rangle 2$ as the inductive step.

$\langle 2\rangle 4$. Q.E.D.

PROOF: By $\langle 2\rangle 3$, taking $\ell = m$; and by definition of Q (3.2).

$\langle 1\rangle 2$. ASSUME: For some $1 \le k \le m$, ($\star$) does not hold for $i = k$.

PROVE: There exist finite $\mathbf{I}, \mathbf{I}'$ such that $\mathbf{V}(\mathbf{I}) = \mathbf{V}(\mathbf{I}')$ but $Q(\mathbf{I}) \neq Q(\mathbf{I}')$.

$\langle 2\rangle 1$. Choose $\mathbf{t}$ such that $\theta(\mathbf{t})$ holds, but for every $1 \le j \le n_k$:

N1. $\theta_{k,j}(t_k)$ does not hold, or

N2. There exists $t'_{k,j}$ such that $\Phi_{k,j}(t_k, t'_{k,j})$ but $\neg\Psi_{k,j}(\mathbf{t}, t'_{k,j})$.

PROOF: Such $\mathbf{t}$ exists by the negation of ($\star$).

DEFINE: Database instances $\mathbf{I}, \mathbf{I}'$:

$$\begin{array}{rcll} \mathbf{I}(R_i) & \triangleq & \textsc{if}\ i = k\ \textsc{then} & \{t'_{k,j} : \theta_{k,j}(t_k), 1 \le j \le n_k\} \qquad \text{(as in N2)} \\ & & \textsc{else} & \{t_i\}, \\ \mathbf{I}'(R_i) & \triangleq & \textsc{if}\ i = k\ \textsc{then} & \mathbf{I}(R_k) \cup \{t_k\} \\ & & \textsc{else} & \mathbf{I}(R_i). \end{array}$$

$\langle 2\rangle 2$. ASSUME: NEW $V_{i,j} \in \mathbf{V}$

PROVE: $V_{i,j}(\mathbf{I}) = V_{i,j}(\mathbf{I}')$

$\langle 3\rangle 1$. CASE: $i \neq k$

PROOF: $V_{i,j}$ refers only to relation R_i by assumption, and $\mathbf{I}(R_i) = \mathbf{I}'(R_i)$ by the construction of $\mathbf{I}'$.

$\langle 3\rangle 2$. CASE: $i = k$

Since $V_{k,j}$ refers only to relation name R_k, we will treat $V_{k,j}$ as a function on the relation R_k—i.e., $V_{k,j}(\mathbf{I}) = V_{k,j}(\mathbf{I}(R_k))$.

$\langle 4\rangle 1$. $V_{k,j}(\mathbf{I}(R_k)) \subseteq V_{k,j}(\mathbf{I}'(R_k))$

PROOF: Because $\mathbf{I}(R_k) \subseteq \mathbf{I}'(R_k)$ by definition, and by monotonicity of project-select queries.

$\langle 4\rangle 2$. $V_{k,j}(\mathbf{I}'(R_k)) \subseteq V_{k,j}(\mathbf{I}(R_k))$

$\langle 5\rangle 1$. SUFFICES: $V_{k,j}(\{t_k\}) \subseteq V_{k,j}(\mathbf{I}(R_k))$

PROOF: Because $\mathbf{I}'(R_k) = \mathbf{I}(R_k) \cup \{t_k\}$ by construction.

$\langle 5\rangle 2$. CASE: $\theta_{k,j}(t_k)$ holds.

By the definition of $\mathbf{I}$, there exists $t'_{k,j} \in \mathbf{I}(R_k)$ such that $\Phi_{k,j}(t_k, t'_{k,j})$ holds. Expanding the definition of Φ, we have:

$$\wedge\ \theta_{k,j}(t'_{k,j})$$
$$\wedge\ t'_{k,j}[U_{k,j}] = t_k[U_{k,j}]$$

which implies $V_{k,j}(\{t_k\}) = V_{k,j}(\{t'_{k,j}\}) \subseteq V_{k,j}(\mathbf{I}(R_k))$.

$\langle 5\rangle 3.$ CASE: $\theta_{k,j}(t_k)$ does not hold.

PROOF: $V_{k,j}(\{t_k\}) = \emptyset \subseteq V_{k,j}(\mathbf{I}(R_k))$ by definition of $V_{k,j}$ (3.1).

$\langle 5\rangle 4.$ Q.E.D.

PROOF: $\langle 5\rangle 2$ and $\langle 5\rangle 3$ cover all the cases.

$\langle 4\rangle 3.$ Q.E.D.

PROOF: By $\langle 4\rangle 1$ and $\langle 4\rangle 2$.

$\langle 3\rangle 3.$ Q.E.D.

PROOF: $\langle 3\rangle 1$ and $\langle 3\rangle 2$ cover all the cases.

$\langle 2\rangle 3.$ $Q(\mathbf{I}) \neq Q(\mathbf{I}')$.

$\langle 3\rangle 1.$ $\mathbf{t}[U] \in Q(\mathbf{I}')$

PROOF: By construction of $\mathbf{I}'$, we have $t_i \in \mathbf{I}'(R_i)$ for all $1 \leq i \leq m$. By assumption, $\theta(\mathbf{t})$ holds.

$\langle 3\rangle 2.$ $\mathbf{t}[U] \notin Q(\mathbf{I})$

$\langle 4\rangle 1.$ SUFFICES ASSUME: 1. NEW $\mathbf{s} \in \mathbf{I}(R_1) \times \cdots \times \mathbf{I}(R_m)$

2. $\theta(\mathbf{s}) \wedge (\mathbf{s}[U] = \mathbf{t}[U])$

PROVE: FALSE

PROOF: By definition of Q (3.2).

$\langle 4\rangle 2.$ Choose $1 \leq j \leq n_k$ and tuple $t'_{k,j}$ such that:

1. $\mathbf{s} = \mathbf{t}$ EXCEPT $k \mapsto t'_{k,j}$,
2. $\neg\Psi_{k,j}(\mathbf{t}, t'_{k,j})$.

PROOF: Such j and $t'_{k,j}$ exist by the construction of $\mathbf{I}$.

$\langle 4\rangle 3.$ Q.E.D.

PROOF: $\langle 4\rangle 2.2$ contradicts $\langle 4\rangle 1.2$.

$\langle 3\rangle 3.$ Q.E.D.

PROOF: By $\langle 3\rangle 1$ and $\langle 3\rangle 2$.

$\langle 2\rangle 4.$ Q.E.D.

PROOF: By $\langle 2\rangle 2$ and $\langle 2\rangle 3$, noting that $\mathbf{I}$ and $\mathbf{I}'$ are finite.

$\langle 1\rangle 3.$ Finite and unrestricted determinacy coincide.

PROOF: If unrestricted determinacy does not hold, by $\langle 1\rangle 2$ there exists a finite counterexample, and so finite determinacy does not hold either.

$\langle 1\rangle 4.$ Q.E.D.

PROOF: By $\langle 1\rangle 1$, $\langle 1\rangle 2$, and $\langle 1\rangle 3$.

□

Chapter 4

Ote: Access-policy Extraction

4.1 Introduction

Protecting sensitive data from unauthorized access is a critical concern for today's web applications. Therefore, when building web applications, developers must determine what *access-control policy* the application should enact—for example, a university might want a policy that ensures a student's grades are visible only to the student and their instructors.

In today's applications, access-control policies are embedded in application code. Furthermore, in most cases they are spread across several functions and in the filter predicates of multiple database queries. This practice is error-prone: Missing or misspecified access checks have previously led to sensitive-data exposure [10, 57, 71, 72, 91, 134]. But more fundamentally, because the policy is never stated explicitly, it is difficult for anyone other than the application's developer to understand *what* policy is embedded in the code. Worse, as time passes, even the application development team is unlikely to remember the policy, and is unlikely to be able to reconstruct it from application code. While there have been research frameworks that require explicit policy specification [11, 151, 152], we aim to address legacy applications rather than requiring them to be rewritten in such frameworks.

This chapter tackles the task of *policy extraction*: extracting a web application's implicitly embedded access-control policy by summarizing its possible data accesses. A human then reviews the extracted policy to better understand the application's data accesses, ensuring they are within the bounds of intended data revelations. If not, the application likely has an access-check bug to be fixed. Once reviewed, the policy can stand alone as a specification for the application's data accesses and can optionally be enforced using an enforcer [90, 92, 159] to ensure continued compliance.

We present an approach for extracting policies from legacy web applications. Our approach begins by exploring execution paths through application code using concolic execution [53, 127], producing transcripts that record the conditions under which SQL queries are issued (§4.4). These transcripts are then merged and simplified to derive a policy that allows each recorded query to be issued under its conditions (§4.5).

A key challenge here is scalability: Web-backend code is often complex with many branches, making exhaustive path exploration infeasible. But we empirically observe that the logic governing query issuance typically involves only a few simple control- and data-flow operations (§4.4.1). We thus tailor concolic execution to track only those operations, reducing the number of paths to explore—and our implementation effort.

We implemented this approach in Ote, a policy-extraction tool for web applications written in Ruby on Rails. We then applied Ote to three real-world applications, two of which we had previously written policies for by hand. When we compared the extracted policies to the handwritten ones (§4.8.5), we identified several formerly unknown errors in the latter—including a few overly permissive views can reveal sensitive data to unauthorized users. This underscores the difficulty of understanding access-control logic in a complex legacy application and testifies to the utility of policy extraction in aiding this understanding. A further review of the extracted policies uncovered a subtle bug we had inadvertently introduced into application code that silently disabled an access check.

In terms of limitations: Due to its reliance on concolic execution, Ote cannot guarantee that the extracted policy covers all possible application queries or captures every condition under which the queries are issued (§4.3.2), and it does not scale to all web handlers (§4.8.4). In addition, Ote may require the user to specify certain input constraints and to provide hints to prune the exploration space (§4.3.1), supports only a subset of SQL (§4.3.2), and could benefit from a better user interface for policy auditing. Nevertheless, we were encouraged by the usefulness of the policies Ote has already managed to extract, and see this as a meaningful step towards better access control for existing web applications.

4.2 Motivation and Background

4.2.1 Why Policy Extraction?

We were prompted to tackle the policy-extraction problem by our earlier experience hand-crafting policies for existing web applications. A few years ago, while investigating externally enforced access control for web applications,[1] we took two open-source applications [29, 46], chose several representative URL endpoints, and tried our best to write down policies for data that should be allowed for their intended function.

This process was extremely tedious and time-consuming. We reviewed documentation, inspected database schemas, experimented with the applications using sample data, and read their source code.[2] Using this information and common sense, we formulated policies that we thought would allow the endpoints to function correctly while protecting sensitive data. Despite our best

[1]By "externally enforced", we mean having an enforcer program mediate the application's database queries, preventing accesses deemed unauthorized according to an access-control policy [80, 90, 92, 159].

[2]This process was especially hard for us because we were not the developers of these applications. But we suspect that for a large-enough codebase, any single developer would go through a similar (albeit lighter) process.

efforts, we often discovered errors in our drafts, including an omission that would have leaked sensitive data—one that we only later discovered by chance.

We started the policy-extraction effort to make it easier to write an accurate policy. After building Ote, we applied it to the same two applications and compared the extracted policies to our original, handwritten ones. This exercise (§4.8.5) uncovered even more errors in our handwritten policies, which turned out to be overly restrictive in some places (blocking legitimate accesses) and overly permissive in others (allowing unauthorized access). Having Ote would certainly have helped us write a more accurate policy with less effort from the start.

Additionally, while reviewing the extracted policy, we discovered in the application code a subtle bug that we had introduced years earlier when adapting the application for access control. This bug, caused by a misuse of an external library, silently rendered an access check a no-op (§4.8.5). This experience shows that policy extraction can uncover access-control issues even if the policy is not ultimately enforced.

4.2.2 Policy as SQL View Definitions

Before delving into how policy extraction works, we first describe how our access-control policies are specified.

We target web applications that store data in a relational database; when a user visits a page, the application issues SQL queries on the user's behalf and renders the page using the query results. In this context, a classic way to specify access-control policies is to use a list of *view definitions* [97,119,120], which are SQL `SELECT` statements—parameterized by session parameters like the current user ID—that define information in the database that a user is allowed to access.

Under a view-based policy, a query is allowed only if it can be fully answered using the views. This criterion extends to a program that (conditionally) issues multiple queries, by treating the program as "one big query" that returns the results of its constituent queries. These notions can be made precise based on *query determinacy* [99], but in the interest of space, we omit the formal definitions and offer an example instead.[3]

Example 4.1. Suppose a course-management site has a web request handler that displays a course's grade sheet (Listing 4.1). It ensures that the user is an instructor before fetching grades.

Note that the handler has access to both *session parameters* (user ID) and *request parameters* (course ID). Session parameters are trusted (e.g., set by an authentication mechanism), and may appear in the policy and dictate the extent of allowed data access. Request parameters are untrusted (e.g., parsed from an HTTP request) and must not appear in the policy.

A policy for this handler might look like Listing 4.2, where `MyUserId` denotes the user-ID session parameter. Notably, it does not reference the course-ID request parameter, instead allowing the handler's queries *for any course ID*. This policy precisely captures the information the handler can query. ◀

[3]We refer interested readers to the literature for both theoretical [3, 82, 99, 162] and practical [18, 19, 119, 159] treatments of this subject.

Listing 4.1: A handler that displays a course's grade sheet.

```
def view_grade_sheet(db, session, req):
  role = db.sql(
    "SELECT * FROM roles WHERE user_id = ? AND course_id = ?",
    session["user_id"], req["course_id"])
  if role is None:
    raise Http404
  if not role.is_instructor:
    raise Http403
  all_grades = db.sql("SELECT * FROM grades WHERE course_id = ?",
                      role.course_id)
  return format_html(all_grades, ...)
```

Listing 4.2: An example policy for the handler in Listing 4.1.

(V_1)
```
SELECT * FROM roles
WHERE user_id = MyUserId
```
A user can view their role (if any) in any course.

(V_2)
```
SELECT grades.* FROM roles, grades
WHERE roles.user_id = MyUserId
  AND roles.is_instructor
  AND grades.course_id = roles.course_id
```
An instructor for a course can view all grades.

Like prior work in database access control [5,18,19,80,90,119,130,159], we focus on extracting policies for database *reads* (SELECTs) only. Similar techniques can be used to extract conditions for other operations, although our policy language (§4.3.2) and algorithms (§4.5) would need to be extended.

4.3 Overview

Given an application, the ideal policy extractor would produce a view-based access-control policy that satisfies:

Completeness Allows all queries the application can issue.

Tightness Reveals as little information as possible subject to completeness.

Conciseness Has a short representation in SQL.

For example, given Listing 4.1 (but written in a real web framework), we would like to extract the policy in Listing 4.2.

Policy extraction is a challenging task and Ote is not guaranteed to meet all three goals (see §4.3.2). Nevertheless, we show in §4.8 that Ote produces policies that are useful in practice.

4.3.1 Workflow

Before discussing how we approach these goals, we first describe Ote's workflow from a user's perspective (Figure 4.1).

Suppose a user wants to extract a policy from a web application. We assume that the application is written in a supported framework (Ruby on Rails) and that the user is familiar with the application's functionality. The user first (Figure 4.1 left):

1. Declares the handlers to be analyzed, with the names and types of request parameters; and,

2. Writes down the application's database constraints, with the help of a constraint-generation tool (§4.7.3).

Ote supports two general forms of database constraints: (1) a column (or set of columns) is unique, (2) a query Q_1's result is contained in another query Q_2's. These forms can express all constraints we encountered—including non-null, foreign-key, and domain constraints, as well as more advanced kinds. Ote accepts constraints written as a HOCON configuration [87] and provides shorthands for common constraint types.

In our experience (§4.8.1), a handler declaration requires a single line of code and a majority of the database constraints can be auto-generated. If path pruning is required, the user can add further constraints to restrict the input space.

The user then packages the application into a Docker container and invokes Ote, which (Figure 4.1):

1. Explores execution paths through the handlers via concolic execution, producing *transcripts* that record the branches taken and the SQL queries issued (§4.4);

2. For each individual handler, analyzes the transcripts to generate a preliminary policy allowing each query to be issued under its recorded conditions (§§ 4.5.1 to 4.5.3);

3. Merges the individual policies and prunes any redundant views, producing a final policy (§4.5.4).

Next, the user inspects the generated policy, using their domain knowledge of the application's privacy requirements:

- A policy that is too broad can indicate that the application may access unauthorized data (i.e., an access-control bug). The user can investigate the queries that yielded a too-broad view using inputs logged by concolic execution (§4.7.3) and modify the application if appropriate.

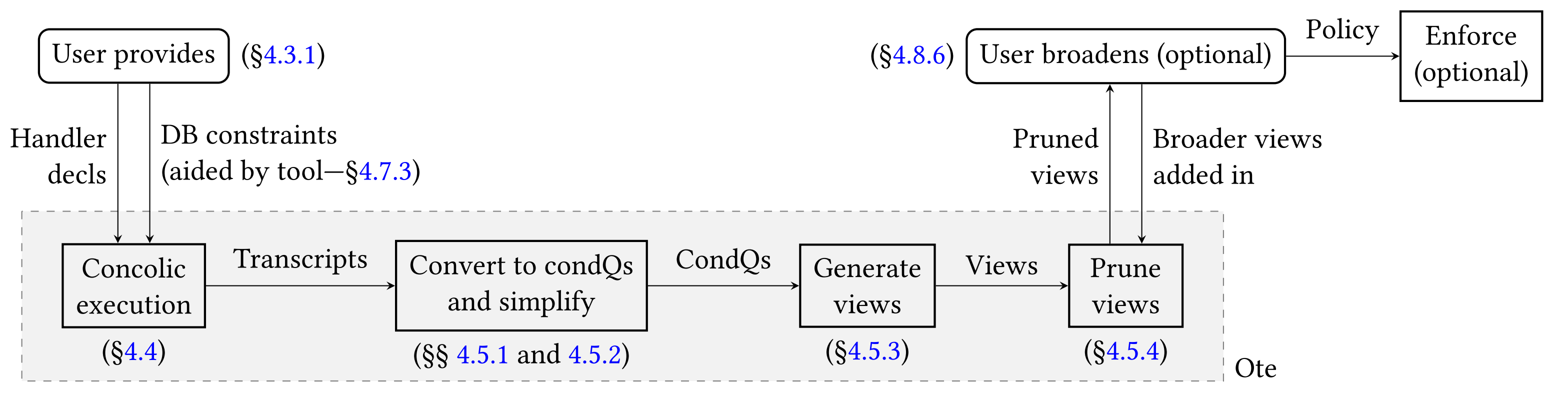


Figure 4.1: **Policy extraction workflow.** “CondQs” stands for conditioned queries (§4.5.1).

- A policy that is too tight can be *broadened* to reveal more data. This may simplify the policy—by removing filters or by replacing multiple views with a single broader one. Ote supports iterative policy broadening: The user adds a broader view into the policy, re-invokes pruning (Step 3) to remove now-redundant views, and repeats (§4.8.6).

Once satisfied, the user can optionally enforce the policy to ensure the application's current and future compliance.

4.3.2 Assumptions and Scope

Queries At its core, Ote supports project-select-join queries in set semantics. These are queries that (1) always return distinct rows, and (2) have the form:

```
SELECT [DISTINCT] col1, col2, ...
FROM tbl1, tbl2, ... WHERE ...                    (PSJ)
```

Also supported are common queries that Ote can mechanically rewrite into this form, such as queries with inner joins or of the form `SELECT 1 FROM tbl1, tbl2, ... WHERE ... LIMIT 1`.

We found that the distinct-rows assumption does not limit utility, having never encountered queries that may return duplicate rows in our evaluation. But applications do issue queries more complex than PSJ; this is handled differently in different stages: The concolic-execution driver precisely models richer SQL features (§4.4.4), but view generation and pruning must approximate complex queries using PSJ (§§ 4.5.3 and 4.8.3). We plan to extend our prototype to support more complex queries.

Policies Ote generates policies consisting of PSJ views, which are expressive enough for the applications we studied. A notable limitation is that PSJ views cannot generally express negations (e.g., "Q_1 is allowed only if Q_2 returns no rows"). Negations complicate approximating a query's information content [142, §2.2] and can slow down enforcement [159, §6], so we defer handling negations to future work.

Non-guarantees Ote *does not guarantee* completeness or tightness: It may generate a policy that disallows a query issued by the application, or one that can be tightened while allowing the same application queries. This is because:

- Concolic execution can miss execution paths as the input space explored is bounded.
- Uninstrumented operations in the *query-issuing core* (§4.4.1) can lead to incomplete path conditions (§4.6).
- Approximating complex queries using PSJ can make a policy more, or less, restrictive than ideal (§4.8.3).

In general, complete-and-tight policy extraction for Turing-complete code is impossible [118]. But in practice, Ote can produce policies more accurate than handwritten ones (§4.8.5).

Similarly, Ote *does not guarantee* that the policy has the most concise SQL representation, but its simplification and pruning steps (§4.5) make it feasible to inspect the policy (§4.8.5).

Application assumptions Ote uses a modified Ruby interpreter and Ruby on Rails framework for concolic execution (§4.4.5), and so it supports only applications written in Rails. However, our approach generalizes to other languages.

Ote's effectiveness depends on a "simple query-issuing core" assumption (§4.4.1). In short, we assume that the application's SQL query issuance depends only on *simple expressions*—ones that use only operations instrumented by Ote's Ruby interpreter. Most notably, Ote does not instrument string formatting (doing so would complicate SMT solving), and so it requires the application to issue SQL queries in parameterized form rather than splicing parameters into query strings within Ruby. Fortunately, idiomatic Rails code already issues parameterized queries by default, and the few places in our evaluation where this isn't the case were easy to fix (§4.8.1).

4.4 Exploring Executions

Ote begins by exploring paths through application code via concolic execution. This section explains how concolic execution works and why we chose it as the exploration strategy, and details our tailored implementation. But we begin with some empirically observed characteristics of typical web applications that motivated our choice of concolic execution.

4.4.1 Observation: Simple Query-issuing Cores

Given a web application's codebase, consider all program statements that issue SQL queries. Informally, imagine the backward program slice [147] from these statements. This slice, which we will call the *query-issuing core*, is what policy extraction is concerned with: It includes any program component on which a query-issuing statement has a control- or data-dependence, and omits the rest (e.g., HTML generation).

While the codebase as a whole can be complex, it has been observed [32, 128] that the query-issuing core of a typical web application is often *simple*. We confirm this observation—for the applications we studied, the core consists mostly of:

1. Conditionals that test if a query's result set is empty [128, § 2] or check basic conditions on primitive values (e.g., checking for equality or nullity) [129, § 6.2];

2. Loops over a query's result set [128, § 2] with no loop-carried dependencies [32, § 4]; and,

3. Trivial data-flow to query statements—e.g., passing a value returned by one query to another. (Given that the application issues parameterized SQL queries, no query-string formatting operation appears in the data flow.)

Our goal is then to analyze this simple query-issuing core while not wasting resources exploring the rest of the (more complex) codebase, with the additional challenge that the two parts of the code are not explicitly separated. We found concolic execution to be a technique that is up to this task. (In §4.6 we discuss what happens when the core is *not simple*.)

4.4.2 Concolic Execution: What and Why

In concolic execution [53, 127], a program is executed repeatedly using concrete inputs that have symbolic variables attached. As the program runs, its state is tracked both concretely and symbolically. When the program branches on a symbolic condition, the condition and its outcome are recorded. This produces a conjunction of constraints—the *path conditions*—that led execution down a path. The conjuncts are then negated using a solver to generate new inputs (up to a bound) that will steer execution down other paths.

We chose concolic execution because it offers a "pay-as-you-go" model for symbolic tracking: We selectively instrument the operations that might appear in the query-issuing core, and the rest will simply execute concretely by default. This strategy not only reduces our instrumentation effort, but also mitigates path explosion—a branch on an uninstrumented condition will not cause a new path to be explored.

Concolic execution has the additional benefit (like symbolic execution) of having "no false positives": Each path explored comes with a concrete input, which the user can use to recreate the run that led to a query being issued (§4.7.3).

4.4.3 System Architecture

Ote has a *driver* that generates inputs and concurrent *executors* that run application code on each input. The driver tracks explored paths in a prefix tree; for every prefix, it negates the last condition and invokes an SMT solver to generate a new input. (The driver keeps the prefix tree in memory and generates inputs sequentially, but both can be relaxed to improve performance.) It then sends the input to an executor, which runs a handler using an instrumented Ruby interpreter and Ruby on Rails framework (§4.4.5) and sends back a *transcript* capturing the path conditions and queries issued (§4.4.4). Exploration terminates when all prefixes have been visited.

4.4.4 Symbolic Modeling and Input Generation

Concolic execution requires symbolically modeling the handler's inputs, consisting of the database and session/request parameters. To ensure termination, the input space must be bounded. Following prior work [30], we model the database as tables containing a bounded number of symbolic rows (our prototype uses a bound of 2). Inspired by UrFlow's loop analysis [32, § 4.2], we also restrict the input space so that each query returns at most one row. For simplicity, we will describe our algorithms under this assumption, even though they can be extended to handle queries returning multiple rows.

Listing 4.3: A transcript from a run of the handler in Listing 4.1, when the user is an instructor for the course.

1. $\textsc{Query}_1$(`SELECT * FROM roles WHERE user_id = ? AND course_id = ?`, $\langle$`MyUserId`, `CourseId`$\rangle$, $\mathit{isEmpty} = \mathit{false}$)
2. $\textsc{Branch}(r_1.\mathit{is\_instructor}, \mathit{outcome} = \mathit{true})$
3. $\textsc{Query}_2$(`SELECT * FROM grades WHERE course_id = ?`, $\langle r_1.\mathit{course\_id}\rangle$, $\mathit{isEmpty} = \mathit{false}$)

Under this modeling, a transcript is a sequence of operations performed by the application with two types of records:

1. $\textsc{Query}_i(\mathit{sql}, \mathit{params}, \mathit{isEmpty})$, meaning the i^{th} query issued was the parameterized query *sql* with parameters *params*, and the result set was empty if *isEmpty* is true.[4] If not empty, a symbol r_i is introduced to later records representing the result row.

2. $\textsc{Branch}(\mathit{cond}, \mathit{outcome})$, meaning the condition *cond* was branched on, and the *outcome* (either true or false) branch was taken. The condition can reference session and request parameters as well as columns returned by previous queries (e.g., "$r_5.\mathit{author\_id} = \mathit{MyUserId}$").

Listing 4.3 shows an example transcript from a run of the handler in Listing 4.1, when the user is an instructor.

Our SMT encoding represents bounded database tables using conditional tables [65] and uses the theory of integers to model all database values [61], including timestamps and strings. Each nullable value is accompanied by a boolean indicating if it is `NULL`. This simple encoding naturally supports equality and arithmetic operations but not string operations, a limitation that can be lifted using a string solver [21, 84].

Similar to prior work [61], we encode a subset of SQL into SMT on bounded symbolic tables. Our encoding supports left- and inner-joins and count- and sum-aggregations. One notable unsupported feature is ordering, which we have not needed assuming that each query returns at most one row.[5]

The driver implements several optimizations for input generation; the most impactful ones were reusing Z3 AST objects, calling the solver incrementally, and caching conflicts (which result in infeasible paths) using unsat cores [125].

[4]Ote assumes that the application always inspects the result set of a query, branching on its emptiness, and does not track it as a separate $\textsc{Branch}$.

[5]Ordering cannot be ignored when a `LIMIT` clause can truncate a result set with multiple distinct rows; we encountered no such queries in our evaluation.

4.4.5 Instrumentation and Tracking

To maintain symbolic state, we modified the JRuby interpreter [68] to add an optional "symbolic expression" field to each Ruby object. For each class that we want represented symbolically, we implement a `with_sym` method that returns a clone with a symbolic expression attached, and amend methods that we want instrumented to attach symbolic expressions to their results. Unmodified methods simply return an object with no expression attached, concretizing the result as desired.

We implemented symbolic representations of nine built-in classes (including `String`, `Fixnum`, Booleans, and `NilClass`) and one Rails class (`ActiveSupport::TimeWithZone`), covering all symbolic inputs in our evaluation. For these classes we instrumented a few simple operations—equality, null-check, cloning, and logical negation (`!`)—that appear in the query-issuing core. Additional handling was needed for:

Certain primitive classes We ensure that instances of `true`, `false`, and `nil` with different symbolic expressions are treated as equal and have identical hash code.

Mutable classes Their mutating methods must by default clear the symbolic expression, lest it be out of sync with the new concrete state. For `String`, we implemented the clearing within JRuby's `RubyString::frozenCheck` method, which is called by all mutating methods.

At run time, Ote's library attaches symbolic expressions to request and session parameters, and a modified Rails database layer attaches symbolic expressions to query results (§4.7).

To track queries and branches, we added a library for maintaining transcript records, exposing methods `record_query` and `record_branch` (§4.4.4). We modify the Rails database layer to call `record_query` after every query, and modify JRuby's `isTrue` and `isFalse` methods (which evaluate an object's truthiness and lack thereof) to call `record_branch` with the object's symbolic expression if it has one.

Finally, we also implemented an optimization: In our setup, a few conversion methods in the database layer have unnecessary branching (e.g., `if x.nil? then nil else x end`); we manually model these using simple function summaries [52].

4.5 Generating a Policy

After exploration, Ote merges and simplifies the transcripts and creates a preliminary set of views for each handler. It then gathers the views for all handlers explored and removes redundancy by leveraging an existing enforcement tool, Blockaid [159]. We now delve into this policy-generation process.

4.5.1 Preprocessing Into Conditioned Queries

As a first step, Ote processes the transcripts for each handler into a set of *conditioned queries*. A conditioned query is a tuple $\langle sql, params, conditions \rangle$, where *conditions* is a list of prior QUERY and Branch records.[6] It associates each query with the conditions under which it is issued; one conditioned query is generated for each query issued in each transcript. As Ote does not currently support negations in policies (§4.3.2), it drops from *conditions* any Query record with $\textit{isEmpty} = \textit{true}$ (i.e., a condition that a prior query returns empty).

4.5.2 Simplifying Conditioned Queries

For each handler, Ote simplifies the set of conditioned queries by removing redundancy. As outlined in Algorithm 4.1, Ote:

- Removes Branches that *must* be taken due to an input constraint or a prior condition (Line 2), as determined by the SMT solver during exploration;
- Unifies variables that are constrained to be equal by query filters (Line 3);
- Removes identical Query records from each conditioned query's *conditions* (Line 4);[7]
- Drops *vacuous* Query records—for queries that *must* return a row due to, e.g., a foreign-key dependency—whose result is not subsequently referenced (Line 8);
- Merges pairs of conditioned queries that differ only in the outcome of a single Branch record (Line 10)—the query is issued no matter which way the branch goes;
- Removes a conditioned query if another exists with the same *sql* and *params* but only a subset of the *conditions* (Line 12)—the latter subsumes the former.

Each step is parallelized across cores. Due to the large number of conditioned queries, we designed these steps to favor efficiency over optimality. For example, in "remove subsumed" we refrain from full-blown implication checking and instead use a simpler procedure that attempts to map records from one set of conditions to another—any missed opportunities for simplification will be caught by the final pruning step (§4.5.4).

4.5.3 Generating SQL View Definitions

Ote now generates one SQL view per conditioned query. The view should reveal the same information as the conditioned query's path if its conditions are met, and no information otherwise. For this, Ote uses an iterative algorithm that "conjoins" each condition record onto an accumulated query definition A. It maintains the invariant that query A:

- Returns empty if any previous condition is violated;

[6] A prior Query represents the *condition* that a query Q returned empty or not; the *action* of Q's issuance is captured by Q's own conditioned queries.

[7] Identical queries arise because we disable Rails's query cache (§4.7), but often they can be identified only after unifying equal variables.

Algorithm 4.1 Simplifying a set of conditioned queries (§4.5.2).

```
 1: for all conditioned query do
 2:     remove vacuous branches
 3:     propagate equalities
 4:     remove duplicate queries
 5: end for
 6: repeat
 7:     for all conditioned query do
 8:         remove vacuous-and-unused query records
 9:     end for
10:     repeat merge branches until convergence
11: until convergence
12: remove subsumed
```

- Returns the Cartesian product of all prior queries' results (i.e., concatenations of one-row-per-query) otherwise.

Query A serves two purposes: It captures the branching conditions, and it exposes query results to be referenced by later records. Lastly, the algorithm conjoins the final query onto A.

Before fully specifying the view-generation algorithm, let us walk through an example.

Example 4.2. Consider Listing 4.3. We shall generate the view for the conditioned query associated with Query$_2$.

The algorithm starts with the query A that returns the empty tuple. After the first record (Query$_1$), A is updated to A_1:

```
SELECT * FROM roles
WHERE user_id = MyUserId
  AND course_id = CourseId
```

(We use CamelCase for request and session parameters.)

After the second record (the Branch), A is updated to A_2:

```
SELECT * FROM roles
WHERE user_id = MyUserId
  AND course_id = CourseId
  AND is_instructor
```

Observe that A indeed returns the same rows as Query$_1$ if the branch condition holds, and an empty result otherwise.

Then, Query$_2$ is conjoined onto A, resulting in view V:

```
SELECT roles.*, grades.* FROM roles, grades
WHERE roles.user_id = MyUserId
  AND roles.course_id = CourseId
  AND roles.is_instructor
  AND grades.course_id = roles.course_id
```

Algorithm 4.2 View generation from conditioned query (§4.5.3).

```
1: procedure GenerateSqlView(cq)
2:     $A \leftarrow \{\langle\rangle\}$                          ▷ Constant query returning empty tuple
3:     $\mathcal{M} \leftarrow \{\}$                          ▷ Maps query result column to $A$'s column
4:     for all $cond \in cq.conds$ do
5:         if $cond$ is Branch($\theta$, $outcome = true$) then
6:             $A \leftarrow \sigma_{\theta[\mathcal{M}]}(A)$
7:         else if $cond$ is Branch($\theta$, $outcome = false$) then
8:             $A \leftarrow \sigma_{\neg\theta[\mathcal{M}]}(A)$
9:         else if $cond$ is Query$_\ell$($sql$, $params$) then
10:            $Q_\ell \leftarrow$ SqlToRa($sql$, $params$)
11:                                                       ▷ Converts SQL query to relational algebra
12:            $(A, \mathcal{M}) \leftarrow$ ConjoinQuery($Q_\ell, A, \mathcal{M}$)
13:        end if
14:    end for
15:    $Q_{\ell+1} \leftarrow$ SqlToRa($cq.sql$, $cq.params$)
16:    $(A, \mathcal{M}) \leftarrow$ ConjoinQuery($Q_{\ell+1}, A, \mathcal{M}$)
17:    return RaToSql($A$)
18: end procedure
19: procedure ConjoinQuery($Q_\ell, A, \mathcal{M}$)
20:    let $Q_\ell = \pi_{j_1,\ldots,j_m}\sigma_\theta(S_1 \times S_2 \times \cdots)$          ▷ Normal form
21:    $n \leftarrow \mathrm{arity}(A)$                                 ▷ Number of columns in $A$
22:    $\theta' \leftarrow \theta[k \mapsto k+n][\mathcal{M}]$                        ▷ $\forall$ column index $k$
23:    $A \leftarrow \pi_{1,\ldots,n,n+j_1,\ldots,n+j_m}\sigma_{\theta'}(A \times S_1 \times S_2 \times \cdots)$
24:    $\mathcal{M} \leftarrow \mathcal{M} \cup \{r_\ell.i \mapsto i+n \mid 1 \le i \le m\}$
25:    return $(A, \mathcal{M})$
26: end procedure
```

Note that Query$_2$'s sole parameter, r_1.*course_id*, has been replaced by the `course_id` column exposed by A_2. A final step remains to remove the `CourseId` parameter, which we will discuss at the end of this subsection. ◀

A generated view can be thought of as a natural generalization of the conditioned query to cases where a SQL query can return multiple rows: The view allows a query to be issued for every combination of rows returned by the prior queries, as long as the branching conditions are met. This would allow the program to have loops over result sets in the form we assume in the simple query-issuing core (§4.4.1).

We specify the view-generation procedure in Algorithm 4.2, which uses relational algebra notations (under the unnamed perspective) [1, § 3.2] for conciseness. The algorithm works for PSJ queries in normal form [1, Prop. 4.4.2]. It does not currently handle general joins or aggregations; when these arise, we approximate them using supported constructs (§4.8.3).

Algorithm 4.2 essentially follows the steps illustrated by Example 4.2. It maintains a mapping $\mathcal{M}$ from result columns of prior queries to columns in A (e.g., mapping r_1.*course_id* to the third column of A). It then uses this mapping to resolve references to results from prior queries.

Removing request parameters Recall from Example 4.1 that request parameters like `CourseId` must not appear in view definitions. So strictly speaking, Example 4.2 has produced not one view, but a set of views:[8]

$$\{V[\texttt{CourseId} \mapsto x] : x \in \mathrm{dom}(\texttt{roles.course\_id})\}$$

To collapse this set into one view, we use the following fact.

Fact 4.3 (Informal)**.** Let $V[\texttt{X}]$ be a view definition of the form:

```
SELECT col₁, col₂, ... FROM tbl₁, tbl₂, ...
WHERE colⱼ = X AND f
```

where $\texttt{col}_j$ is a non-nullable column, `X` is a request parameter, and f does not refer to `X`. Then the set of views $\{V[\texttt{X} \mapsto x] : x \in \mathrm{dom}(\texttt{col}_j)\}$ reveals the same information as the view:

```
SELECT colⱼ, col₁, col₂, ...
FROM tbl₁, tbl₂, ... WHERE f
```

While this fact applies only to queries of a specific form, it already covers all cases encountered in our evaluation. We leave a theoretical study of the general case to future work.

Example 4.4 (continues=example:view-generation)**.** Starting from view V, Ote removes the condition `roles.course_id = CourseId`, but refrains from adding `roles.course_id` to the `SELECT` statement as it is redundant with the already-present `grades.course_id`. This brings us to the final view $V^\star$:

```
SELECT * FROM roles, grades
WHERE roles.user_id = MyUserId
  AND roles.is_instructor
  AND grades.course_id = roles.course_id
```

Compared to the handwritten view V_2 from Listing 4.2, the view $V^\star$ is identical except that it retains more columns. However, *conditioned on* the existence of V_1 (which Ote would have generated from another conditioned view), $V^\star$ reveals the same information as V_2 and so is equally tight. For simplicity, Ote outputs $V^\star$ without removing extraneous columns, since we find $V^\star$ just as concise and readable as V_2. ◀

Outputting SQL Ote outputs SQL views in the form:

```
SELECT ... FROM tbl1, tbl2, ... WHERE ...
```

[8]We let $V[\texttt{X} \mapsto x]$ denote the view definition V with each occurrence of `X` replaced by x, and $\mathrm{dom}(\texttt{col})$ denote the set of valid values for column `col`.

similar to the representation of $V^\star$ above. These view definitions are not always as concise as they can be—e.g., they may contain joins that are redundant due to unique- or foreign-key dependencies. Ote reduces clutter in the generated SQL by:

- Removing joins of two copies of a table on a unique key;
- Reducing parenthesizing in `WHERE` clauses by transforming a SQL `AND` tree into a left-deep tree before unparsing;
- Coalescing column names into `table.*` when possible.

We plan to extend Ote to remove other redundancies using standard techniques for query optimization [1, § 6].

4.5.4 Pruning Views via Enforcement

Lastly, Ote minimizes the set of views for each handler—producing a subset that reveals the same information—and then takes their union and minimizes it again. To minimize a set $\mathbf{V}$ of views, Ote goes through each view $V \in \mathbf{V}$ and checks whether the information revealed by V is already contained in that revealed by $\mathbf{V} \setminus \{V\}$; if so, it removes V. Heuristically, Ote sorts the views by the number of joins in decreasing order, so that longer views have a chance of being removed first.

It remains to check information containment. This is the same problem as policy enforcement: checking whether issuing V *as a query* is allowed under the policy $\mathbf{V} \setminus \{V\}$. To tackle this, we repurpose an existing enforcement tool, Blockaid [159]. Specifically, we extended Blockaid with a command-line interface, which Ote invokes as described earlier. Finally, Ote outputs the minimized set of views for human review.

4.6 Discussion

The simple query-issuing core assumption (§4.4.1) is not true for all practical applications. It can be violated in two ways.

Complex control-flow conditions This is when a query's issuance depends on a complex expression. For example, diaspora* crashes if a photo's `url` is not a valid URL, thus preventing later queries; the URL-validity condition relies on regex-matching operations, which Ote does not instrument.

Suppose a query Q is issued conditioned on an expression C with some uninstrumented operations. This may cause Ote to generate a policy that is either broader or tighter than ideal:

1. If Ote happens to generate an input that satisfies C (this is not guaranteed because the driver is unaware of C), then Q will appear as a conditioned query *without* C in its condition, leading to a policy broader than ideal.

2. If Ote never generates an input that satisfies C, then Q may not appear in any transcript, leading to a policy that may not allow Q—which is tighter than ideal.

In practice, Case 1 has not been a problem for us because (1) the complex condition C typically reflects business logic and has no privacy implications (§4.8.6), and (2) query Q is likely also issued elsewhere under a simpler condition. Case 2 is problematic for validations (e.g., URL validity); we get around this by manually setting the input using database constraints (e.g., setting URLs to always be `http://foo.com`).

If an uninstrumented operation turns out important for an application domain, we would update Ote's Ruby instrumentation, SMT encoding, and SQL generation to support it.

Complex data flow This is when the transcript contains an expression derived from a symbolic value through uninstrumented operations—a problematic scenario because the symbolic value is concretized. For example, the conditional `if uid < uid * uid` may insert a BRANCH record for $\mathit{uid} < 16$ into the transcript, as multiplication is not instrumented. In such cases, the driver might fail to terminate because a "new path condition" emerges for every value of `uid`. Even if we cut off the exploration, the generated policy would be overly strict and verbose, littered with concrete filters like $\mathit{uid} < 16$.

To detect such harmful concretizations, the driver reports each new constant and new query it encounters (there should be a bounded number of these). This mechanism alerted us to a few non-parameterized SQL queries (§4.8.1), where symbolic values were formatted into the query string. We encountered no concretization issues in our final evaluation, and we plan to implement more precise detection of such "partially concrete" conditions via heavier-weight data-flow tracking.

4.7 Implementational and Practical Aspects

4.7.1 Driver and Policy Generator

Ote's concolic-execution driver (§4.4) and policy generator (§4.5) are implemented in Scala 3. The driver uses Apache Calcite [17] to parse SQL queries and convert them into relational algebra, and invokes Z3 using its Java binding [43]. It communicates with executors using Protobuf messages, and saves program inputs and transcripts to compressed JSON files. The policy generator uses Calcite's query analyses and optimizations for conditioned-query simplification (§4.5.2), and uses Scala's parallel-collections library [111] for parallelization.

4.7.2 Executors

Ote's library uses RSpec [121] to invoke handlers (Rails "controller actions") with inputs from the driver. At startup, the executor clears the database. For each input, it begins a transaction, populates the database, installs symbolic request parameters by patching the `params` hash, and sets two symbolic session parameters: the user ID (by logging in using a symbolic integer) and the current time (by patching `Time.now`). It then invokes the handler and rolls back the transaction.

We disable Rails's fragment and low-level caching to expose queries issued only on cache misses. As an optimization, we also disable Rails's query cache, which introduces branches that do not affect what data is fetched. We execute handlers on MySQL backed by an in-memory `tmpfs`, and configure string columns to use a case-sensitive collation [161] as our SMT encoding (§4.4.4) does not support case insensitivity.

4.7.3 Tooling

Generating database constraints To help the user write down an application's database constraints (§4.3.1), Ote provides a tool that generates common simple types of constraints for a Rails application, including uniqueness and foreign keys. It does so by inspecting the database schema's SQL constraints and the Active Record models' validators, associations, and inheritance hierarchy. The user may then supplement the generated list with more advanced constraints (§4.8.1).

In general, generating constraints is a tricky task because (1) constraints from the object-relational mapping (ORM) are commonly absent from the database [153, § 4.2], and (2) some constraints are *not even declared in the ORM* [64]. We plan to explore more advanced techniques for inferring constraints from either code [64, 88] or data [16, 37, 63, 154].

Tracing a view back to the application When reviewing an extracted view, the user may wonder: What code in the application was responsible for this view's generation? To help trace a view back to its source, Ote outputs with each view the ID of an execution from which the view is derived. The user can then recover the concrete input for that execution and re-run it. (Ote supports launching an executor using an ad hoc input.) To gain visibility into the run, the user may:

- Launch an executor in "verbose mode", which captures a stack trace for each transcript record; or,
- Launch an executor under a Ruby debugger, allowing them to view the program's state when a query is issued.

We found both methods helpful for understanding a view's origin. In the future, we plan to explore better workflows and user interfaces to enhance the "policy debugging" experience.

4.8 Evaluation

We applied Ote to three existing Ruby-on-Rails applications:

1. diaspora* [46], a social network with over 850 k users;
2. Autolab [29], a platform for managing course assignments used at over 20 schools; and,

3. The Odin Project (Odin) [137], a site where over a million users take web-dev classes and share their work.

We chose diaspora* and Autolab because we had previously written policies for them by hand; we will be comparing these handwritten policies to the extracted ones. In contrast, we had never worked with Odin before, and so our experience applying Ote to Odin will be unbiased by prior knowledge.

The key takeaways from this evaluation are:

- Using Ote does not require significant human effort.
- Ote can extract a policy within hours.
- The extracted policies avoid, and alerted us to, several errors and omissions in the handwritten policies.
- Reviewing the extracted policies revealed an access-check bug that we had inadvertently introduced.

4.8.1 Setting Up Applications for Ote

Code changes We used versions of diaspora* (v0.7.14) and Autolab (v2.7.0) that had previously been modified to work with Blockaid. To summarize the most relevant modifications:

- We had modified a few code locations to issue parameterized queries. (Most queries were already parameterized, but some needed a rewrite—see below for an example.)
- We had modified the applications to fetch sensitive data only if it affects the output. (This allows for finer-grained policies, but is *not necessary for use with Ote*. We kept these modifications to enable an apples-to-apples comparison between extracted and handwritten policies.)
- We had rewritten one query to an equivalent form supported by Ote. (This modification is *not fundamentally required*—it could have been implemented in Ote itself.)

For Odin, we made changes to three code locations in its backend source code (commit `f6762f0`):

- We changed two code locations so that they issue parameterized SQL queries—e.g., in one location we changed:
  ```
  where('expires >= ?', Time.now)
  ```
  to a form that uses a Ruby endless range [107]:
  ```
  where(expires: Time.now..)
  ```
  because the former inserts `Time.now` into the query string within Ruby, resulting in non-parameterized SQL.

Table 4.1: **Number of database constraints.** "Application Logic" = constraints from application logic; "pruning" = constraints we added to prune the input space. "Auto" = auto-generated constraints; "manual" = manually written ones.

	Application Logic		**Pruning**	
	Auto	Manual	Manual	**Total**
diaspora*	347	63	11	421
Autolab	185	42	3	230
Odin	116	9	1	126

- We deleted one line of code to prevent an aggregation from appearing in the *conditions* portion of conditioned queries (Ote does not currently handle this). This change preserves application behavior.

We also omitted JRuby-incompatible gems used for development and testing, modified the configuration to use MySQL, and adjusted some HTML and JavaScript so that pages render correctly. These changes are to accommodate our prototype and environment and are not fundamental to policy extraction.

Database constraints Table 4.1 breaks down the database constraints supplied to Ote. The constraint generator (§4.7.3) produced over $80\,\%$ of the constraints that were used. The rest, which we manually wrote, were either inferred from application logic or added for input-space pruning (see below). The number of manual constraints ranged from 10 to 74.

A few manual constraints were unobvious—e.g., that two diaspora* posts cannot be reshares of each other. When we missed these, the application would crash or hang during path exploration, which would then alert us to the missing constraints. For fields with constraints that are either unsupported by our SMT encoding (e.g., URL validity) or related to the external environment (e.g., an Autolab course name must have a file directory), we manually constrained them to a fixed set of valid values (after creating any environmental state).

Input-space pruning To reduce the input space, we impose a few more constraints based on our application knowledge:

- We restrict nullable string fields to be non-empty. (Null and empty strings are typically treated identically.)
- We restrict a diaspora* user to be non-admin and non-moderator, and an Autolab user to non-admin.[9]

[9]There is less of a need to extract policies for these cases—the additional privileges for these roles are easy to write down.

- We assume there are no Autolab scheduler actions, which are background tasks not triggered by user requests.
- We assume there are no diaspora* "services," which are concerned with external calls and not database access.
- We constrain a diaspora* user to have non-null names and photos, and to have English as their language.

We applied no pruning to Odin other than the first restriction.

To develop these constraints, we ran path exploration for a bounded time, inspected the longest transcripts to identify branches that could be removed without substantially degrading the quality of the extracted policy, and imposed constraints to remove these branches by rendering them vacuous.

For Autolab, we aimed to execute the six handlers for which we had previously handwritten policies, but ended up excluding one of them due to its large number of paths (Remark 4.5).

4.8.2 Experiment Setup

We ran the experiments on Google Compute Engine using a `t2a-standard-48` instance (48 Ampere Altra Arm cores). We used 24 parallel executors, which saturated input generation. Executors run a modified version of JRuby v9.3.13.0 atop OpenJDK 21. The driver uses Z3 v4.11.2. For view pruning, we configured Blockaid with a timeout of $5\,\mathrm{s}$.

4.8.3 Paths, Conditioned Queries, and Views

Table 4.2 shows that while the number of explored paths can be large, Ote is able to reduce the number of views to between 24 and 140—a manageable number for human review.

For view generation, we had to approximate some SQL features that either Ote or Blockaid does not support. To do so, we wrote scripts to transform conditioned queries:

- We rewrite a `LEFT JOIN` into an equivalent `INNER JOIN` if possible. Otherwise, we split it into an `INNER JOIN` (for rows that match) and a `SELECT` (for rows that do not); this will split one conditioned query into two.
- We rewrite a `SELECT COUNT(*) FROM tbl` in the query portion into `SELECT id FROM tbl`.
- We omitted date-timestamp comparisons and replaced "`Now + 1 second`" with "`Now`".

These approximations are "lossy": They can broaden or tighten the policy and must be applied with human discretion.

Table 4.2: **Statistics and performance for path exploration and policy generation.** Under Statistics, "#Cond. Queries" shows the number of conditioned queries before and after simplification (§4.5.2); "#SQL Views" shows the number of views after per- and cross-handler pruning (§4.5.4). Under Running Time, "Simplify" stands for conditioned-query simplification and view generation (§§ 4.5.2 and 4.5.3), "Prune" for per-handler view-pruning, and "Final Prune" for cross-handler view-pruning (§4.5.4).

	Statistics			**Running Time**			
Handler	#Paths	#Cond. Queries	#SQL Views	Explore	Simplify	Prune	Final Prune
diaspora*			→ 140				50 min
`People#stream`	1 184 510	3 260 165 → 201	148	8.1 h	6 min	49 min	
`Posts#show`	256 124	1 668 722 → 209	87	2.0 h	2 min	28 min	
`People#show`	16 652	51 128 → 38	32	8 min	33 s	6 min	
`Notifications#index`	5306	24 756 → 76	54	4 min	32 s	12 min	
`Conversations#index`	75	511 → 41	16	2 min	28 s	5 min	
`Comments#index`	19	42 → 24	17	2 min	27 s	4 min	
Autolab			→ 51				17 min
`Assessments#gradesheet`	18 972	10 000 → 103	72	7 min	33 s	17 min	
`Assessments#index`	9047	10 218 → 97	27	5 min	32 s	11 min	
`Submissions#download`	99	59 → 25	18	1 min	25 s	4 min	
`Courses#index`	36	22 → 10	5	2 min	24 s	1 min	
`Metrics#getNumPending`	11	20 → 12	7	1 min	24 s	2 min	
The Odin Project			→ 24				11 min
`Lessons#show`	75 466	73 812 → 153	46	20 min	39 s	19 min	
`Courses#show`	6192	4488 → 38	25	4 min	26 s	5 min	
`ProjectSubmissions#index`	3052	2616 → 40	17	3 min	25 s	5 min	
`Users#show`	2803	2924 → 15	9	3 min	24 s	2 min	
`Paths#index`	29	27 → 7	4	2 min	22 s	55 s	
`Sitemap#index`	28	14 → 5	3	1 min	22 s	40 s	

4.8.4 Performance

Table 4.2 (right) shows the running times for each step of policy extraction. End to end, the most time-consuming handler (`People#stream`) completes in around ten hours. Aside from path exploration, the most expensive step is view pruning (§4.5.4). This is because (1) Ote relaunches Blockaid for every view, incurring start-up cost; (2) Blockaid is optimized for checking *compliant* queries against a *fixed* policy, while Ote uses it to check often non-compliant (i.e., non-redundant) queries against a policy being iteratively pruned; and (3) view pruning is sequential. We plan to improve Blockaid for our use case and devising parallel pruning algorithms.

Remark 4.5. We attempted path exploration on Autolab's `Assessments#show` handler, but it did not finish after many hours. This handler displays the details of an assessment, branching on the fields of the assessment and its submissions, scores, etc., but many of these branches have no influence on data access. One potential solution is to apply selective constraint generation [146, § 2.5], which could help prune irrelevant branches by approximating backward slices.

4.8.5 Findings From the Extracted Policies

We now compare the extracted policies with the handwritten ones for diaspora* and Autolab.

Expectations We generally expect an extracted policy to be tighter (more restrictive) than a handwritten one: Ote aims to produce the tightest possible policy, while humans often relax non-privacy-critical conditions and allow broader accesses than the application requires (§4.8.6). Thus, the extracted policy would be longer as it encodes conditions not imposed by the handwritten one. Indeed, Table 4.3 shows that the extracted policies have roughly twice as many views as their handwritten counterparts—but they remain manageable for human review.

Where handwritten policies reveal too much But not all relaxations by the human policy-writer are benign. When we compared the extracted Autolab policy against the handwritten one, we found that the latter granted course assistants access to five types of records in a "disabled" course, when the application's logic states that only *instructors* should have access. Such an erroneous policy, if enforced, would allow a future code change to leak sensitive data to course assistants.

Where handwritten policies reveal too little Unexpectedly, there is also information revealed by the extracted policy but not by the handwritten one. To investigate, we used Blockaid to check whether each extracted view is allowed as a query under the handwritten policy and found that:

- The handwritten diaspora* policy failed to reveal the pod of a "remote" person and data for `MentionedInPost` and `MentionedInComment` notifications.

Table 4.3: **View count in extracted vs handwritten policies.** The extracted policies have more views because in many places, they are more fine-grained and restrictive (§4.8.5).

	diaspora*	Autolab	Odin
Extracted policy	140	51	24
Handwritten policy	66	28	–

- The handwritten Autolab policy overlooked granting instructors access to all attachments in their courses.

Enforcing a policy with oversights like these would disrupt normal application function by denying legitimate data accesses from the application.

A defective access check While reviewing the extracted Autolab policy, we noticed that the `assessments.exam` column was never checked in the submissions-related views. This was suspicious, as we knew that Autolab prohibited students from downloading prior exam submissions.

It turned out that we had introduced a bug years earlier when adapting Autolab for access control. To manage the sensitive columns in the `assessments` table, we used the `lazy_column` gem [89] to defer fetching these columns until needed. But when specifying columns to lazy-load, we mistakenly used the column's query method name `exam?` instead of its actual name `exam`; this altered the `exam?` method to always return `nil` (interpreted as false). So a correct-looking access check in the controller code was rendered a no-op due to a misuse of external library elsewhere in the codebase—a subtle bug that we discovered only from the extracted policy.

4.8.6 Broadening the Extracted Policy

Sometimes, an extracted policy includes many combinations of conditions under which some data can be accessed, typically reflecting business logic rather than privacy concerns. For example, the extracted diaspora* policy has 44 views (out of 140) that reveal profile data under various conditions—if the profile belongs to the current user, or to the author of a public post, or to the author of a comment on a public post, etc. But by our understanding, a diaspora* user's profile is generally public, except for a few columns guarded by the `public_details` flag. If this interpretation holds, we can simplify the policy by *broadening* it.

We can broaden the policy with the help of Ote's view pruning (§4.5.4). First, we write down four simpler, broader views to capture the relaxed conditions for accessing profiles:

1.
   ```
   SELECT id, first_name, ... FROM profiles
   ```
   (For all profiles, some columns are always visible…)

2.
   ```
   SELECT * FROM tags, taggings
   WHERE tags.id = taggings.tag_id
     AND taggings.taggable_type = 'Profile';
   ```

(. . . and so are the profile taggings.)

3.
```
SELECT * FROM profiles
WHERE public_detail = TRUE
```
(All columns are visible if profile has "public details". . .)

4.
```
SELECT * FROM profiles, people
WHERE profiles.person_id = people.id
  AND people.owner_id = MyUserId
```
(. . . or if the profile belongs to the current user)

We add these to the policy and re-run view-pruning, which removes 41 of the 44 profile-related views as redundant. This process spares the human from having to reason about view subsumption, which can be tedious and tricky. (For example, the remaining three views, which pertain to the current user's contacts, are *not* subsumed by the ones that we added!)

4.9 Related Work

Symbolic execution Symbolic execution [70] is a classic path-exploration technique that maintains symbolic program state. We decided not to use it because implementing it for an interpreted language is challenging (more so than for lower-level languages [28, 96, 131]) due to the dynamic features and functionality implemented outside the language [26, § 2.2].

Bug finding Many tools use symbolic or concolic execution to look for bugs in web applications [9, 30, 146], typically aiming to trigger certain statements or program states. Policy extraction requires not only reaching the query-issuing statements, but also gathering the conditions under which these statements are reached. Derailer [100] and Space [101] check for security bugs in web applications by validating data exposures against human input or a catalog. We could use similar techniques to (semi-)automatically analyze extracted policies.

Instrumentation strategies Some systems track symbolic values in dynamic languages by performing instrumentation *within* the target language. They represent symbolic objects through proxying [25,143] or inheritance [12], and track path conditions using Boolean-conversion hooks [12, 25] or debug tracing [143, § 4.1.2]. These approaches avoid the need for a custom interpreter and support standard environments [8]. In contrast, Ote performs *offline* analysis, allowing us to modify the interpreter for better transparency [41] and performance.

Learning models of database applications Konure infers models of database-backed applications [128, 129] by generating targeted inputs to probe the application as a black-box. We took inspiration from Konure when stating the simple query-issuing core assumption (§4.4.1). However, a black-box approach cannot effectively extract conditionals from application code [129, § 6.2]; Ote is able to do so via instrumentation.

Policy mining Policy-mining systems share our high-level goal of generating access-control rules that capture existing practice. They take existing access-control lists [27,148,149], operation logs [50,67,95], or human interactions [66], and produce role- [50,95,148], attribute- [67,149,150], or relationship-based [27, 66] rules, often via (statistical) learning techniques. In contrast, Ote does not require live data and instead analyzes the application, which also provides the visibility needed to produce fine-grained view-based policies.

4.10 Conclusion and Future Work

Ote is an initial step towards better access control for existing web applications. Our results show that Ote can extract policies from practical web applications and avoid errors found in handwritten policies. We have also identified areas for improvement, such as parallelizing input generation (§4.4.3) and view pruning (§4.8.4), supporting richer SQL features (§4.4.4), and improving scalability with advanced instrumentation (Remark 4.5).

We end by pointing out a risk in policy extraction: for humans to rubber-stamp an extracted policy that appears reasonable. Enforcing such a policy would create a false sense of security and fail to catch errors. This is an example of *automation bias and complacency*, where humans overly rely on computer output and abdicate their decision-making responsibility [109]. We will study the extent of these issues and mitigate them using established measures [51].

Chapter 5

Future Directions

Let us now recap what we have achieved with Blockaid and Ote:

During development We can take an existing web application and use Ote to extract a draft policy for it. This draft policy may not be perfect—in fact, it *will be buggy* if the existing application has access-check bugs—but a human will review the policy and fix or adjust it as needed to produce the final policy.

At deployment We can install Blockaid to enforce this final policy, so that if the application has a bug—either right now or in the future—Blockaid will prevent it from revealing any information it shouldn't.

These two systems form a holistic solution for protecting sensitive data in web applications today.

But our work is not done. In the remainder of this chapter, we will discuss some challenges we have not yet addressed, and point out some directions for future work.

5.1 Policy Testing

5.1.1 Challenge: Evaluating a Policy for Sensitive-data Disclosure

A policy, be it hand-written or extracted, should be sanity-checked before being put into production. Policies have two potentially conflicting imperatives. On the one hand, they must allow queries required for the application's operation. On the other, they must prevent users from learning something about sensitive information (i.e., data that should remain hidden). In many cases, no policy satisfies both imperatives. So how can an administrator evaluate how much sensitive information is disclosed, so that she can determine whether the policy (and the application's functionality) must be modified to limit such disclosure?

To be more precise, suppose S is a query whose answer should be hidden (we will refer to such queries as *sensitive queries*). The administrator first checks whether query S is blocked by the policy. But she must go further: even if S is blocked, substantial information could be disclosed on the answer to S from answers to other queries *allowed* by the policy [94].

Example 5.1 ([45, §2])**.** Consider a hospital-management system whose policy allows staff to view 1. the doctor assigned to each patient, and 2. the diseases treated by each doctor; but the disease each patient is treated for is deemed sensitive information. Suppose patient John is treated by a doctor who only treats two diseases. The policy would block a direct query for John's disease, but discloses enough information to narrow the answer down to two possibilities. ◀

Challenge. Design an evaluation tool that detects potential sensitive-data disclosure by a given data-access policy.

The first problem we face when tackling this challenge is to define what we mean by "disclosure". Despite much prior work, identifying a practical notion of disclosure useful to operators turns out to be nontrivial.

5.1.2 Existing Work: Bayesian Privacy

One of the most well-studied notions of disclosure in the database literature is that of Bayesian privacy [45, 94], where disclosure is modeled as the *shift in an adversary's belief* for the answer to a sensitive query S after observing the views. The bigger the shift, the more the knowledge gained by the adversary, and the more extensive the disclosure.

To complicate matters, such shift depends not only on the policy and sensitive query, but also on the adversary's *prior belief*. For example, a neighbor who has seen John coughing might change his belief only slightly when he learns that John is treated by a doctor who treats only pneumonia and tuberculosis (Example 5.1); someone without prior knowledge of John's cough might undergo a bigger shift.

As a result, Bayesian privacy criteria are typically parameterized by the *class of prior beliefs* considered. But we are now caught between a rock and a hard place: we can assume either (1) a general class of priors (e.g., all tuple-independent distributions [94]), yielding a criterion that applies to diverse adversaries but imposes impractically strict restrictions on what a policy can reveal; or (2) a specific family of priors (as in Dalvi et al. [42]), yielding a criterion that is more permissive but applies only to a specific class of adversaries—one that might not match the adversaries that arise in reality.

5.1.3 Proposal: Prior-agnostic Privacy

At the root of this dilemma is Bayesian privacy's reliance on modeling the adversary's prior belief. In contrast to the kinds of distributions routinely modeled in systems work—like for traffic arrivals [106], which can be readily measured and validated—distributions on people's prior beliefs are much harder to model realistically and validate empirically. And if we can't validate a prior, we can't precisely interpret a Bayesian guarantee based on that prior.

For this reason, we think it is time to turn to *prior-agnostic* privacy criteria—ones that do not require modeling priors. Many such criteria can be defined, and no one criterion fits all. We highlight two examples from computational logic: *positive query implication (PQI)* and *negative query implication (NQI)* [20, Def. 3.5], adapted to view-based access control.

Fix a set $\mathbf{V}$ of policy views and a sensitive query S. We call a row t a *possible answer* to S if it is returned by S on *some* database, a *certain answer* if on all databases, and an *impossible answer* if on no database. Then, we say:

- $\mathrm{PQI}_S(\mathbf{V})$ holds if revealing the contents of $\mathbf{V}$ could render a possible answer to S certain.
- $\mathrm{NQI}_S(\mathbf{V})$ holds if revealing the contents of $\mathbf{V}$ could render a possible answer to S impossible.

PQI and NQI signal disclosure—i.e., the contents of $\mathbf{V}$ enabling certain inferences about the answer to S. We illustrate these concepts with a toy example.

Example 5.2. Define two queries on an employee database:

(Q_1) `SELECT name FROM Employees WHERE age >= 60`

(Q_2) `SELECT name FROM Employees WHERE age >= 18`

Take $\mathbf{V} = \{Q_1\}$ and $S = Q_2$. Revealing Q_1's answer allows positive inference on Q_2's answer: if Q_1 returns `"Alex"`, then so must Q_2. Thus, $\mathrm{PQI}_{Q_2}(\{Q_1\})$ holds.

Conversely, take $\mathbf{V} = \{Q_2\}$ and $S = Q_1$. Revealing Q_2's answer allows negative inference on Q_1's: if Q_2 *doesn't* return `"Alex"`, then nor can Q_1. So we have $\mathrm{NQI}_{Q_1}(\{Q_2\})$.

PQI and NQI are prior-agnostic: nowhere in our reasoning did we appeal to assumptions on the adversary's belief. ◀

Remark 5.3. In all fairness, *if* it were possible to accurately model belief as a probability distribution, then Bayesian privacy would be a valuable metric as it provides the probability of someone holding that belief correctly guessing a sensitive value—exactly the event we wish to avoid. Our proposal is motivated only by the inherent difficulty of modeling belief.

As far as we know, algorithms for checking PQI and NQI have been studied only in theoretical contexts for simple, conjunctive queries [20]. Practical algorithms exist for checking k-anonymity [123, 136] (another prior-agnostic criterion), but they typically assume single-table schemas. It is a promising direction to explore how to extend these algorithms to complex schemas and queries found in practice.

5.2 Violation Diagnosis

5.2.1 Challenge: Troubleshooting Violations

Having produced a policy she's happy with, the administrator enables policy enforcement on her application. One day, the application (possibly after a code update) issues a query that gets blocked due to policy violation. What has gone wrong?

Answering this question can be difficult. Because we use allow-list policies (i.e., views that a user is *allowed* to access),[1] no item or subset of items in the policy can be singled out for causing the violation. Then what form of feedback should be provided to help the administrator diagnose the problem?

While providing feedback is straightforward for simpler policy specifications (like row- or column-level policies), the solution is less obvious for the more expressive view-based policies. A natural proposal is to display a *counterexample*—in Blockaid's case, a pair of databases on which every view produces the same answer, but the blocked query produces different answers.[2] However, while a counterexample is a proof-of-violation, it is not easily interpreted by the administrator—what is she to do with two databases shown side by side?

Challenge. Assist human in diagnosing policy violations.

Streamlining diagnosis is crucial to keeping an access-control deployment manageable. The more effort needed to resolve violations, the more likely is the administrator to forgo access control out of frustration or, worse, to silence violations by setting overly permissive policies, leaving data unprotected.

5.2.2 Proposal: Patch Generation

A policy violation is caused by either the policy being stricter than intended, or the application accessing more data than intended. A tool cannot easily distinguish between the two cases, but it can *suggest patches* to both the policy and the application such that, once any patch is applied, the offending query would be allowed. Even patches that do not get applied can help. For example, if all policy patches look unreasonable (e.g., they allow every user access to all calendar events), then the application—not the policy—is the likely culprit.

Patching the policy

Policy patches, consisting of modified/added view definitions, can be generated via policy extraction (Chapter 4): run the extraction algorithm either on the up-to-date source code, or on a test suite augmented with the offending query, and then compare the extracted policy with the current one. The extraction algorithm could also be augmented to produce deltas over an existing policy.

Patching the application

A typical application patch would take one of two forms:

[1]Allow-lists can naturally implement least privilege: simply write the policy to allow the minimum necessary information. Block-lists, where the extent of allowed access is implicit, risk granting more privilege than necessary.

[2]Intuitively, for a query to be allowed, its answer must be uniquely determined by the answers to the views; a counterexample refutes this property. See prior work for a more formal discussion [159, §4.2].

1. Narrowing down the offending query (e.g., by adding a conjunct to its SQL `WHERE` clause), or

2. Wrapping the offending query in an additional access check (along the lines of the `if` statements in Listing 4.1).

We envision both forms of patching will work at the query level, and can be applied to applications written in any language. In particular, an access-check patch will consist of a condition on database content (e.g., the existence of a particular row), which can be checked in any application language.[3]

The two patch forms might require different techniques to generate. Conceptually, the task of **narrowing down a blocked query** Q reduces to the database-theoretic problem of finding a *contained rewriting* Q' of Q using the policy views [83]—i.e., Q' may refer only to view names (and not base tables), and its answer must be a subset of Q's on all databases.[4] There has also been theoretical work on finding *maximally* contained rewritings—ensuring Q' returns as much data as possible without violating the policy—for restricted query languages like conjunctive queries (CQs) [82, 112] and CQs with arithmetic comparisons [4]. The practical systems problems, then, are (1) to extend these algorithms to more expressive query languages found in practice, and to implement them efficiently; and (2) to empirically evaluate the extent to which the rewriting found helps a developer.

Generating an access check requires finding a statement about database content such that (1) once known, this statement (with the existing trace) makes the blocked query compliant; and (2) the statement is consistent with the existing trace. In Example 2.2, if Query 2 were issued alone (it would be blocked), one such statement would be "the `Attendance` table contains row `(UId=2, EId=5)`", which the developer can check for in her code before issuing the query.

The search for such a statement falls under *abductive inference*: finding an "explanatory hypothesis for a desired outcome" [47], with the desired outcome being policy compliance for the blocked query. As such, a promising approach is to leverage program synthesis techniques for abduction [116].

5.3 Policy Comprehension

While (a subset of) SQL as a policy language is precise and familiar, it can be verbose for complex policies. A promising direction is to design a more concise policy domain-specific language (DSL) that desugars into SQL. To ensure that the DSL is comprehensible and easy to learn, we can borrow ideas from user studies on query languages [7, 114] and from the design of object-relational mappings [138]. Once a DSL is in place, a policy-extraction tool must convert extracted SQL

[3] This is in contrast to leak repair for liquid information flow control [108, §5], which statically analyzes source code written in a special type system.

[4] More precisely, we only need the latter condition to hold on all databases consistent with the trace prior to Q's issuance.

views into this DSL; we may be able to implement this conversion using techniques like verified lifting [69, 76].

5.4 Decidable Compliance Checking

SMT solvers have proven effective for checking query compliance in real-world scenarios, but it remains theoretically unclear if compliance checking is decidable in these cases. Even for single-query checking, where Blockaid's criterion degenerates into query determinacy, our understanding still remains "at the extremes". On one side, we proved that query determinacy is decidable for project-select views and a project-select-join query with no self joins, provided the selection formulas are reasonable (Chapter 3), but this result is too restricted for practical use. On the other side, query determinacy is undecidable for conjunctive queries [54, 55], yet practical views and queries look far simpler than those constructed in the undecidability proofs. So a natural question would be: Is there a natural class of views and queries—reflecting those in practice—for which compliance checking is decidable? An affirmative answer could lead to explicit compliance-checking algorithms with more stable performance than SMT solving [104].

Bibliography

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases.* Addison-Wesley, 1995.

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

[3] Foto N. Afrati and Rada Chirkova. *Answering Queries Using Views, Second Edition.* Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2019.

[4] Foto N. Afrati, Chen Li, and Prasenjit Mitra. Rewriting queries using views in the presence of arithmetic comparisons. *Theor. Comput. Sci.*, 368(1-2):88–123, 2006.

[5] Rakesh Agrawal, Paul Bird, Tyrone Grandison, Jerry Kiernan, Scott Logan, and Walid Rjaibi. Extending relational database systems to automatically enforce privacy policies. In Karl Aberer, Michael J. Franklin, and Shojiro Nishio, editors, *Proceedings of the 21st International Conference on Data Engineering, ICDE 2005, 5-8 April 2005, Tokyo, Japan*, pages 1013–1022. IEEE Computer Society, 2005.

[6] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. Hippocratic databases. In *Proceedings of 28th International Conference on Very Large Data Bases, VLDB 2002, Hong Kong, August 20-23, 2002*, pages 143–154. Morgan Kaufmann, 2002.

[7] Alireza Ahadi, Julia Coleman Prior, Vahid Behbood, and Raymond Lister. A quantitative study of the relative difficulty for novices of writing seven different types of SQL queries. In *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education, ITiCS 2015.* ACM, 2015.

[8] Kalev Alpernas, Aurojit Panda, Leonid Ryzhyk, and Mooly Sagiv. Cloud-scale runtime verification of serverless applications. In Carlo Curino, Georgia Koutrika, and Ravi Netravali, editors, *SoCC '21: ACM Symposium on Cloud Computing, Seattle, WA, USA, November 1 - 4, 2021*, pages 92–107. ACM, 2021.

[9] Shay Artzi, Adam Kiezun, Julian Dolby, Frank Tip, Danny Dig, Amit M. Paradkar, and Michael D. Ernst. Finding bugs in dynamic web applications. In Barbara G. Ryder and Andreas Zeller, editors, *Proceedings of the ACM/SIGSOFT International Symposium on Software*

Testing and Analysis, ISSTA 2008, Seattle, WA, USA, July 20-24, 2008, pages 261–272. ACM, 2008.

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

[11] Thomas H. Austin, Jean Yang, Cormac Flanagan, and Armando Solar-Lezama. Faceted execution of policy-agnostic programs. In Prasad Naldurg and Nikhil Swamy, editors, *Proceedings of the 2013 ACM SIGPLAN Workshop on Programming Languages and Analysis for Security, PLAS 2013, Seattle, WA, USA, June 20, 2013*, pages 15–26. ACM, 2013.

[12] Thomas Ball and Jakub Daniel. Deconstructing dynamic symbolic execution. In Maximilian Irlbeck, Doron A. Peled, and Alexander Pretschner, editors, *Dependable Software Systems Engineering*, volume 40 of *NATO Science for Peace and Security Series, D: Information and Communication Security*, pages 26–41. IOS Press, 2015.

[13] 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 Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*, volume 13243 of *Lecture Notes in Computer Science*, pages 415–442. Springer, 2022.

[14] 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.

[15] 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.

[16] Jana Bauckmann, Ziawasch Abedjan, Ulf Leser, Heiko Müller, and Felix Naumann. Discovering conditional inclusion dependencies. In Xue-wen Chen, Guy Lebanon, Haixun Wang, and Mohammed J. Zaki, editors, *21st ACM International Conference on Information and Knowledge Management, CIKM'12, Maui, HI, USA, October 29 - November 02, 2012*, pages 2094–2098. ACM, 2012.

[17] 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 Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 221–230. ACM, 2018.

[18] Gabriel Bender, Lucja Kot, and Johannes Gehrke. Explainable security for relational databases. In Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu, editors, *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 1411–1422. ACM, 2014.

[19] Gabriel Bender, Lucja Kot, Johannes Gehrke, and Christoph Koch. Fine-grained disclosure control for app ecosystems. In Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 869–880. ACM, 2013.

[20] Michael Benedikt, Pierre Bourhis, Balder ten Cate, Gabriele Puppis, and Michael Vanden Boom. Inference from visible information and background knowledge. *ACM Trans. Comput. Log.*, 22(2):13:1–13:69, 2021.

[21] Murphy Berzish, Vijay Ganesh, and Yunhui Zheng. Z3str3: A string solver with theory-aware heuristics. In Daryl Stewart and Georg Weissenbacher, editors, *2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017*, pages 55–59. IEEE, 2017.

[22] Aaron R. Bradley and Zohar Manna. *The calculus of computation - decision procedures with applications to verification.* Springer, 2007.

[23] 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.

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

[25] Alessandro Bruni, Tim Disney, and Cormac Flanagan. A peer architecture for lightweight symbolic execution, February 2011. Retrieved April 4, 2024 from `https://hoheinzollern.wordpress.com/wp-content/uploads/2008/04/seer1.pdf`.

[26] Stefan Bucur, Johannes Kinder, and George Candea. Prototyping symbolic execution engines for interpreted languages. In Rajeev Balasubramonian, Al Davis, and Sarita V. Adve, editors, *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2014, Salt Lake City, UT, USA, March 1-5, 2014*, pages 239–254. ACM, 2014.

[27] Thang Bui and Scott D. Stoller. A decision tree learning approach for mining relationship-based access control policies. In Jorge Lobo, Scott D. Stoller, and Peng Liu, editors, *Proceedings of the 25th ACM Symposium on Access Control Models and Technologies, SACMAT 2020, Barcelona, Spain, June 10-12, 2020*, pages 167–178. ACM, 2020.

[28] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In Richard Draves and

Robbert van Renesse, editors, *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 209–224. USENIX Association, 2008.

[29] Autolab project. https://autolabproject.com/.

[30] Avik Chaudhuri and Jeffrey S. Foster. Symbolic security analysis of Ruby-on-Rails web applications. In Ehab Al-Shaer, Angelos D. Keromytis, and Vitaly Shmatikov, editors, *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, October 4-8, 2010*, pages 585–594. ACM, 2010.

[31] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. Optimizing database-backed applications with query synthesis. In Hans-Juergen Boehm and Cormac Flanagan, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 3–14. ACM, 2013.

[32] Adam Chlipala. Static checking of dynamically-varying security policies in database-backed applications. In Remzi H. Arpaci-Dusseau and Brad Chen, editors, *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings*, pages 105–118. USENIX Association, 2010.

[33] Adam Chlipala. Ur: statically-typed metaprogramming with type-level record computation. In Benjamin G. Zorn and Alex Aiken, editors, *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, pages 122–133. ACM, 2010.

[34] Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. Secure web application via automatic partitioning. In Thomas C. Bressoud and M. Frans Kaashoek, editors, *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*, pages 31–44. ACM, 2007.

[35] 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.

[36] Shumo Chu, Konstantin Weitz, Alvin Cheung, and Dan Suciu. Hottsql: proving query rewrites with univalent SQL semantics. In Albert Cohen and Martin T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 510–524. ACM, 2017.

[37] Xu Chu, Ihab F. Ilyas, and Paolo Papotti. Discovering denial constraints. *Proc. VLDB Endow.*, 6(13):1498–1509, 2013.

[38] E. F. Codd. Relational completeness of data base sublanguages. *Research Report / RJ / IBM / San Jose, California*, RJ987, 1972.

[39] Ellis S. Cohen. Information transmission in computational systems. In Saul Rosen and Peter J. Denning, editors, *Proceedings of the Sixth Symposium on Operating System Principles, SOSP 1977, Purdue University, West Lafayette, Indiana, USA, November 16-18, 1977*, pages 133–139. ACM, 1977.

[40] Brian J. Corcoran, Nikhil Swamy, and Michael W. Hicks. Cross-tier, label-based security enforcement for web applications. In Ugur Çetintemel, Stanley B. Zdonik, Donald Kossmann, and Nesime Tatbul, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, pages 269–282. ACM, 2009.

[41] Limitations – crosshair 0.0.54 documentation. Retrieved April 16, 2024 from https://crosshair.readthedocs.io/en/latest/limitations.html.

[42] Nilesh N. Dalvi, Gerome Miklau, and Dan Suciu. Asymptotic conditional probabilities for conjunctive queries. In Thomas Eiter and Leonid Libkin, editors, *Database Theory - ICDT 2005, 10th International Conference, Edinburgh, UK, January 5-7, 2005, Proceedings*, volume 3363 of *Lecture Notes in Computer Science*, pages 289–305. Springer, 2005.

[43] Leonardo de Moura. Z3 for Java, 2012. Retrieved October 6, 2024 from https://leodemoura.github.io/blog/2012/12/10/z3-for-java.html.

[44] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[45] Alin Deutsch. Privacy in database publishing: A bayesian perspective. In Michael Gertz and Sushil Jajodia, editors, *Handbook of Database Security - Applications and Trends*, pages 461–487. Springer, 2008.

[46] The diaspora* project. https://diasporafoundation.org/.

[47] Isil Dillig, Thomas Dillig, and Alex Aiken. Automated error diagnosis using abductive inference. In Jan Vitek, Haibo Lin, and Frank Tip, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 181–192. ACM, 2012.

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

[49] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam D. Smith. Calibrating noise to sensitivity in private data analysis. In Shai Halevi and Tal Rabin, editors, *Theory of*

Cryptography, Third Theory of Cryptography Conference, TCC 2006, New York, NY, USA, March 4-7, 2006, Proceedings, volume 3876 of *Lecture Notes in Computer Science*, pages 265–284. Springer, 2006.

[50] Nurit Gal-Oz, Yaron Gonen, Ran Yahalom, Ehud Gudes, Boris Rozenberg, and Erez Shmueli. Mining roles from web application usage patterns. In Steven Furnell, Costas Lambrinoudakis, and Günther Pernul, editors, *Trust, Privacy and Security in Digital Business - 8th International Conference, TrustBus 2011, Toulouse, France, August 29 - September 2, 2011. Proceedings*, volume 6863 of *Lecture Notes in Computer Science*, pages 125–137. Springer, 2011.

[51] Kate Goddard, Abdul V. Roudsari, and Jeremy C. Wyatt. Automation bias: a systematic review of frequency, effect mediators, and mitigators. *J. Am. Medical Informatics Assoc.*, 19(1):121–127, 2012.

[52] Patrice Godefroid. Compositional dynamic test generation. In Martin Hofmann and Matthias Felleisen, editors, *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, pages 47–54. ACM, 2007.

[53] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In Vivek Sarkar and Mary W. Hall, editors, *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, pages 213–223. ACM, 2005.

[54] 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.

[55] Tomasz Gogacz and Jerzy Marcinkowski. Red spider meets a rainworm: Conjunctive query finite determinacy is undecidable. In Tova Milo and Wang-Chiew Tan, editors, *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 121–134. ACM, 2016.

[56] 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.

[57] 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.

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

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

[60] Raju Halder and Agostino Cortesi. Fine grained access control for relational databases by abstract interpretation. In José Cordeiro, Maria Virvou, and Boris Shishkov, editors, *Software and Data Technologies - 5th International Conference, ICSOFT 2010, Athens, Greece, July 22-24, 2010. Revised Selected Papers*, volume 170 of *Communications in Computer and Information Science*, pages 235–249. Springer, 2010.

[61] Yang He, Pinhan Zhao, Xinyu Wang, and Yuepeng Wang. Verieql: Bounded equivalence verification for complex SQL queries with integrity constraints. *Proc. ACM Program. Lang.*, 8(OOPSLA1):1071–1099, 2024.

[62] 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.

[63] Arvid Heise, Jorge-Arnulfo Quiané-Ruiz, Ziawasch Abedjan, Anja Jentzsch, and Felix Naumann. Scalable discovery of unique column combinations. *Proc. VLDB Endow.*, 7(4):301–312, 2013.

[64] Haochen Huang, Bingyu Shen, Li Zhong, and Yuanyuan Zhou. Protecting data integrity of web applications with database constraints inferred from application code. In Tor M. Aamodt, Natalie D. Enright Jerger, and Michael M. Swift, editors, *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, pages 632–645. ACM, 2023.

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

[66] Padmavathi Iyer and Amir Masoumzadeh. Towards automated learning of access control policies enforced by web applications. In Silvio Ranise, Roberto Carbone, and Daniel Takabi, editors, *Proceedings of the 28th ACM Symposium on Access Control Models and Technologies, SACMAT 2023, Trento, Italy, June 7-9, 2023*, pages 163–168. ACM, 2023.

[67] Padmavathi Iyer and Amirreza Masoumzadeh. Mining positive and negative attribute-based access control policy rules. In Elisa Bertino, Dan Lin, and Jorge Lobo, editors, *Proceedings of the 23nd ACM on Symposium on Access Control Models and Technologies, SACMAT 2018, Indianapolis, IN, USA, June 13-15, 2018*, pages 161–172. ACM, 2018.

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

[69] Shoaib Kamil, Alvin Cheung, Shachar Itzhaky, and Armando Solar-Lezama. Verified lifting of stencil computations. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016*. ACM, 2016.

[70] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

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

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

[73] Paraschos Koutris, Prasang Upadhyaya, Magdalena Balazinska, Bill Howe, and Dan Suciu. Query-based data pricing. In Michael Benedikt, Markus Krötzsch, and Maurizio Lenzerini, editors, *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 167–178. ACM, 2012.

[74] Laura Kovács and Andrei Voronkov. First-order theorem proving and vampire. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 1–35. Springer, 2013.

[75] Brian Krebs. USPS site exposed data on 60 million users, 2018. https://krebsonsecurity.com/2018/11/usps-site-exposed-data-on-60-million-users/.

[76] Shadaj Laddad, Conor Power, Mae Milano, Alvin Cheung, and Joseph M. Hellerstein. Katara: synthesizing crdts with verified lifting. *Proc. ACM Program. Lang.*, 6(OOPSLA2), 2022.

[77] Leslie Lamport. How to write a long formula (short communication). *Formal Aspects Comput.*, 6(5):580–584, 1994.

[78] Leslie Lamport. Lower bounds for asynchronous consensus. *Distributed Comput.*, 19(2):104–125, 2006.

[79] Leslie Lamport. How to write a 21st century proof. *Journal of Fixed Point Theory and Applications*, 11(1):43–63, March 2012.

[80] Kristen LeFevre, Rakesh Agrawal, Vuk Ercegovac, Raghu Ramakrishnan, Yirong Xu, and David J. DeWitt. Limiting disclosure in hippocratic databases. In Mario A. Nascimento, M. Tamer Özsu, Donald Kossmann, Renée J. Miller, José A. Blakeley, and K. Bernhard Schiefer, editors, *(e)Proceedings of the Thirtieth International Conference on Very Large Data*

Bases, VLDB 2004, Toronto, Canada, August 31 - September 3 2004, pages 108–119. Morgan Kaufmann, 2004.

[81] 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 Angela Demke Brown and Jay R. Lorch, editors, *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*, pages 441–459. USENIX Association, 2021.

[82] Alon Y. Levy, Alberto O. Mendelzon, Yehoshua Sagiv, and Divesh Srivastava. Answering queries using views. In Mihalis Yannakakis and Serge Abiteboul, editors, *Proceedings of the Fourteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 22-25, 1995, San Jose, California, USA*, pages 95–104. ACM Press, 1995.

[83] Alon Y. Levy, Anand Rajaraman, and Joann J. Ordille. Querying heterogeneous information sources using source descriptions. In T. M. Vijayaraman, Alejandro P. Buchmann, C. Mohan, and Nandlal L. Sarda, editors, *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*, pages 251–262. Morgan Kaufmann, 1996.

[84] Tianyi Liang, Andrew Reynolds, Cesare Tinelli, Clark W. Barrett, and Morgan Deters. A DPLL(T) theory solver for a theory of strings and regular expressions. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 646–662. Springer, 2014.

[85] Mark H. Liffiton and Ammar Malik. Enumerating infeasibility: Finding multiple muses quickly. In Carla P. Gomes and Meinolf Sellmann, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 10th International Conference, CPAIOR 2013, Yorktown Heights, NY, USA, May 18-22, 2013. Proceedings*, volume 7874 of *Lecture Notes in Computer Science*, pages 160–175. Springer, 2013.

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

[87] Lightbend. HOCON (human-optimized config object notation), 2019. Retrieved April 3, 2024 from `https://github.com/lightbend/config/blob/main/HOCON.md`.

[88] Xiaoxuan Liu, Shuxian Wang, Mengzhu Sun, Sicheng Pan, Ge Li, Siddharth Jha, Cong Yan, Junwen Yang, Shan Lu, and Alvin Cheung. Leveraging application data constraints to optimize database-backed web applications. *Proc. VLDB Endow.*, 16(6):1208–1221, 2023.

[89] 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`.

[90] Alana Marzoev, Lara Timbó Araújo, Malte Schwarzkopf, Samyukta Yagati, Eddie Kohler, Robert Tappan Morris, M. Frans Kaashoek, and Sam Madden. Towards multiverse databases. In *Proceedings of the Workshop on Hot Topics in Operating Systems, HotOS 2019, Bertinoro, Italy, May 13-15, 2019*, pages 88–95. ACM, 2019.

[91] 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/.

[92] Aastha Mehta, Eslam Elnikety, Katura Harvey, Deepak Garg, and Peter Druschel. Qapla: Policy compliance for database-backed systems. In Engin Kirda and Thomas Ristenpart, editors, *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, pages 1463–1479. USENIX Association, 2017.

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

[94] Gerome Miklau and Dan Suciu. A formal analysis of information disclosure in data exchange. In Gerhard Weikum, Arnd Christian König, and Stefan Deßloch, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*, pages 575–586. ACM, 2004.

[95] Ian M. Molloy, Youngja Park, and Suresh Chari. Generative models for access control policies: applications to role mining over logs with attribution. In Vijay Atluri, Jaideep Vaidya, Axel Kern, and Murat Kantarcioglu, editors, *17th ACM Symposium on Access Control Models and Technologies, SACMAT '12, Newark, NJ, USA - June 20 - 22, 2012*, pages 45–56. ACM, 2012.

[96] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*, pages 1186–1189. IEEE, 2019.

[97] Amihai Motro. An access authorization model for relational databases based on algebraic manipulation of view definitions. In *Proceedings of the Fifth International Conference on Data Engineering, February 6-10, 1989, Los Angeles, California, USA*, pages 339–347. IEEE Computer Society, 1989.

[98] Andrew C. Myers. Jflow: Practical mostly-static information flow control. In Andrew W. Appel and Alex Aiken, editors, *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*, pages 228–241. ACM, 1999.

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

[100] Joseph P. Near and Daniel Jackson. Derailer: interactive security analysis for web applications. In Ivica Crnkovic, Marsha Chechik, and Paul Grünbacher, editors, *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, pages 587–598. ACM, 2014.

[101] Joseph P. Near and Daniel Jackson. Finding security bugs in web applications using a catalog of access control patterns. In Laura K. Dillon, Willem Visser, and Laurie A. Williams, editors, *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 947–958. ACM, 2016.

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

[103] OWASP Foundation. OWASP Top 10:2021, 2021. https://owasp.org/Top10/.

[104] Oded Padon. *Deductive Verification of Distributed Protocols in First-Order Logic.* PhD thesis, Tel Aviv University, Israel, 2018.

[105] Daniel Pasaila. Conjunctive queries determinacy and rewriting. In Tova Milo, editor, *Database Theory - ICDT 2011, 14th International Conference, Uppsala, Sweden, March 21-24, 2011, Proceedings*, pages 220–231. ACM, 2011.

[106] Vern Paxson and Sally Floyd. Wide-area traffic: The failure of poisson modeling. In Jon Crowcroft, editor, *Proceedings of the ACM SIGCOMM 1994 Conference on Communications Architectures, Protocols and Applications, London, UK, August 31 - September 2, 1994*, pages 257–268. ACM, 1994.

[107] Alex Piechowski. Rails: How to use greater than/less than in Active Record where statements, 2019. Retrieved April 17, 2024 from https://piechowski.io/post/how-to-use-greater-than-less-than-active-record/.

[108] Nadia Polikarpova, Deian Stefan, Jean Yang, Shachar Itzhaky, Travis Hance, and Armando Solar-Lezama. Liquid information flow control. *Proc. ACM Program. Lang.*, 4(ICFP):105:1–105:30, 2020.

[109] Amanda Potasznik. ABCs: Differentiating algorithmic bias, automation bias, and automation complacency. In *2023 IEEE International Symposium on Ethics in Engineering, Science, and Technology (ETHICS)*, pages 1–5, 2023.

[110] Alessandro Previti and João Marques-Silva. Partial MUS enumeration. In Marie desJardins and Michael L. Littman, editors, *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence, July 14-18, 2013, Bellevue, Washington, USA*, pages 818–825. AAAI Press, 2013.

[111] Aleksandar Prokopec and Heather Miller. Overview | Parallel Collections | Scala Documentation. Retrieved April 16, 2024 from https://docs.scala-lang.org/overviews/parallel-collections/overview.html.

[112] Anand Rajaraman, Yehoshua Sagiv, and Jeffrey D. Ullman. Answering queries using templates with binding patterns. In Mihalis Yannakakis and Serge Abiteboul, editors, *Proceedings of the Fourteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 22-25, 1995, San Jose, California, USA*, pages 105–112. ACM Press, 1995.

[113] Giles Reger, Martin Suda, and Andrei Voronkov. Finding finite models in multi-sorted first-order logic. In Nadia Creignou and Daniel Le Berre, editors, *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, volume 9710 of *Lecture Notes in Computer Science*, pages 323–341. Springer, 2016.

[114] Phyllis Reisner. Human factors studies of database query languages: A survey and assessment. *ACM Comput. Surv.*, 13(1), 1981.

[115] Raymond Reiter. On closed world data bases. In Hervé Gallaire and Jack Minker, editors, *Logic and Data Bases, Symposium on Logic and Data Bases, Centre d'études et de recherches de Toulouse, France, 1977*, Advances in Data Base Theory, pages 55–76, New York, 1977. Plemum Press.

[116] Andrew Reynolds, Haniel Barbosa, Daniel Larraz, and Cesare Tinelli. Scalable algorithms for abduction via enumerative syntax-guided synthesis. In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part I*, volume 12166 of *Lecture Notes in Computer Science*, pages 141–160. Springer, 2020.

[117] Andrew Reynolds, Cesare Tinelli, Amit Goel, Sava Krstic, Morgan Deters, and Clark W. Barrett. Quantifier instantiation techniques for finite model finding in SMT. In Maria Paola Bonacina, editor, *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*, volume 7898 of *Lecture Notes in Computer Science*, pages 377–391. Springer, 2013.

[118] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[119] Shariq Rizvi, Alberto O. Mendelzon, S. Sudarshan, and Prasan Roy. Extending query rewriting techniques for fine-grained access control. In Gerhard Weikum, Arnd Christian König, and Stefan Deßloch, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*, pages 551–562. ACM, 2004.

[120] Arnon Rosenthal and Edward Sciore. View security as the basis for data warehouse security. In Manfred A. Jeusfeld, Hua Shu, Martin Staudt, and Gottfried Vossen, editors,

Proceedings of the Second Intl. Workshop on Design and Management of Data Warehouses, DMDW 2000, Stockholm, Sweden, June 5-6, 2000, volume 28 of *CEUR Workshop Proceedings*, page 8. CEUR-WS.org, 2000.

[121] RSpec: Behaviour driven development for Ruby. Retrieved April 16, 2024 from `https://rspec.info/`.

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

[123] Pierangela Samarati. Protecting respondents' identities in microdata release. *IEEE Trans. Knowl. Data Eng.*, 13(6):1010–1027, 2001.

[124] Schneider Downs. Security notice: Major online banking platform vulnerability—Fiserv, 2018. `https://schneiderdowns.com/our-thoughts-on/online-banking-platform-vulnerability-fiserv/`.

[125] Daniel Schwartz-Narbonne, Martin Schäf, Dejan Jovanovic, Philipp Rümmer, and Thomas Wies. Conflict-directed graph coverage. In Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods - 7th International Symposium, NFM 2015, Pasadena, CA, USA, April 27-29, 2015, Proceedings*, volume 9058 of *Lecture Notes in Computer Science*, pages 327–342. Springer, 2015.

[126] Luc Segoufin and Victor Vianu. Views and queries: determinacy and rewriting. In Chen Li, editor, *Proceedings of the Twenty-fourth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 13-15, 2005, Baltimore, Maryland, USA*, pages 49–60. ACM, 2005.

[127] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In Michel Wermelinger and Harald C. Gall, editors, *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*, pages 263–272. ACM, 2005.

[128] Jiasi Shen and Martin C. Rinard. Using active learning to synthesize models of applications that access databases. In Kathryn S. McKinley and Kathleen Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 269–285. ACM, 2019.

[129] Jiasi Shen and Martin C. Rinard. Active learning for inference and regeneration of applications that access databases. *ACM Trans. Program. Lang. Syst.*, 42(4):18:1–18:119, 2021.

[130] Jie Shi, Hong Zhu, Ge Fu, and Tao Jiang. On the soundness property for SQL queries of fine-grained access control in dbmss. In Huaikou Miao and Gongzhu Hu, editors, *8th IEEE/ACIS International Conference on Computer and Information Science, IEEE/ACIS ICIS 2009, June 1-3, 2009, Shanghai, China*, pages 469–474. IEEE Computer Society, 2009.

[131] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Krügel, and Giovanni Vigna. SOK: (state of) the art of war: Offensive techniques in binary analysis. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pages 138–157. IEEE Computer Society, 2016.

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

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

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

[135] Michael Stonebraker and Eugene Wong. Access control in a relational data base management system by query modification. In Roger C. Brown and Donald E. Glaze, editors, *Proceedings of the 1974 ACM Annual Conference, San Diego, California, USA, November 1974, Volume 1*, pages 180–186. ACM, 1974.

[136] Latanya Sweeney. k-anonymity: A model for protecting privacy. *Int. J. Uncertain. Fuzziness Knowl. Based Syst.*, 10(5):557–570, 2002.

[137] The Odin Project. https://www.theodinproject.com/.

[138] Alexandre Torres, Renata Galante, Marcelo Soares Pimenta, and Alexandre Jonatan B. Martins. Twenty years of object-relational mapping: A survey on patterns, solutions, and their implications on application design. *Inf. Softw. Technol.*, 82, 2017.

[139] Margus Veanes, Pavel Grigorenko, Peli de Halleux, and Nikolai Tillmann. Symbolic query exploration. In Karin K. Breitman and Ana Cavalcanti, editors, *Formal Methods and Software Engineering, 11th International Conference on Formal Engineering Methods, ICFEM 2009, Rio de Janeiro, Brazil, December 9-12, 2009. Proceedings*, volume 5885 of *Lecture Notes in Computer Science*, pages 49–68. Springer, 2009.

[140] Margus Veanes, Nikolai Tillmann, and Jonathan de Halleux. Qex: Symbolic SQL query explorer. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*, volume 6355 of *Lecture Notes in Computer Science*, pages 425–446. Springer, 2010.

[141] Chenglong Wang, Alvin Cheung, and Rastislav Bodík. Synthesizing highly expressive SQL queries from input-output examples. In Albert Cohen and Martin T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 452–466. ACM, 2017.

[142] 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 Christoph Koch, Johannes Gehrke, Minos N. Garofalakis, Divesh Srivastava, Karl Aberer, Anand Deshpande, Daniela Florescu, Chee Yong Chan, Venkatesh Ganti, Carl-Christian Kanne, Wolfgang Klas, and Erich J. Neuhold, editors, *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, pages 555–566. ACM, 2007.

[143] Ruowen Wang, Peng Ning, Tao Xie, and Quan Chen. MetaSymploit: Day-one defense against script-based attacks with security-enhanced symbolic analysis. In Samuel T. King, editor, *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, pages 65–80. USENIX Association, 2013.

[144] 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.

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

[146] Gary Wassermann, Dachuan Yu, Ajay Chander, Dinakar Dhurjati, Hiroshi Inamura, and Zhendong Su. Dynamic test input generation for web applications. In Barbara G. Ryder and Andreas Zeller, editors, *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2008, Seattle, WA, USA, July 20-24, 2008*, pages 249–260. ACM, 2008.

[147] Mark D. Weiser. Program slicing. *IEEE Trans. Software Eng.*, 10(4):352–357, 1984.

[148] Zhongyuan Xu and Scott D. Stoller. Algorithms for mining meaningful roles. In Vijay Atluri, Jaideep Vaidya, Axel Kern, and Murat Kantarcioglu, editors, *17th ACM Symposium on Access Control Models and Technologies, SACMAT '12, Newark, NJ, USA - June 20 - 22, 2012*, pages 57–66. ACM, 2012.

[149] Zhongyuan Xu and Scott D. Stoller. Mining attribute-based access control policies. *CoRR*, abs/1306.2401, 2013. http://arxiv.org/abs/1306.2401.

[150] Zhongyuan Xu and Scott D. Stoller. Mining attribute-based access control policies from logs. In Vijay Atluri and Günther Pernul, editors, *Data and Applications Security and Privacy XXVIII - 28th Annual IFIP WG 11.3 Working Conference, DBSec 2014, Vienna, Austria, July 14-16, 2014. Proceedings*, volume 8566 of *Lecture Notes in Computer Science*, pages 276–291. Springer, 2014.

[151] Jean Yang, Travis Hance, Thomas H. Austin, Armando Solar-Lezama, Cormac Flanagan, and Stephen Chong. Precise, dynamic information flow for database-backed applications. In Chandra Krintz and Emery D. Berger, editors, *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 631–647. ACM, 2016.

[152] Jean Yang, Kuat Yessenov, and Armando Solar-Lezama. A language for automatically enforcing privacy policies. In John Field and Michael Hicks, editors, *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 85–96. ACM, 2012.

[153] Junwen Yang, Utsav Sethi, Cong Yan, Alvin Cheung, and Shan Lu. Managing data constraints in database-backed web applications. In Gregg Rothermel and Doo-Hwan Bae, editors, *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, pages 1098–1109. ACM, 2020.

[154] Meihui Zhang, Marios Hadjieleftheriou, Beng Chin Ooi, Cecilia M. Procopiuc, and Divesh Srivastava. On multi-column foreign key discovery. *Proc. VLDB Endow.*, 3(1):805–814, 2010.

[155] Wen Zhang, Dev Bali, Jamison Kerney, Aurojit Panda, and Scott Shenker. Extracting database access-control policies from web applications. *CoRR*, abs/2411.11380, 2024.

[156] Wen Zhang, Vivian Fang, Aurojit Panda, and Scott Shenker. Kappa: a programming framework for serverless computing. In Rodrigo Fonseca, Christina Delimitrou, and Beng Chin Ooi, editors, *SoCC '20: ACM Symposium on Cloud Computing, Virtual Event, USA, October 19-21, 2020*, pages 328–343. ACM, 2020.

[157] Wen Zhang, Aurojit Panda, Mooly Sagiv, and Scott Shenker. A decidable case of query determinacy: Project-select views. *CoRR*, abs/2411.08874, 2024.

[158] Wen Zhang, Aurojit Panda, and Scott Shenker. Access control for database applications: Beyond policy enforcement. In Malte Schwarzkopf, Andrew Baumann, and Natacha Crooks, editors, *Proceedings of the 19th Workshop on Hot Topics in Operating Systems, HOTOS 2023, Providence, RI, USA, June 22-24, 2023*, pages 223–230. ACM, 2023.

[159] Wen Zhang, Eric Sheng, Michael Alan Chang, Aurojit Panda, Mooly Sagiv, and Scott Shenker. Blockaid: Data access policy enforcement for web applications. In Marcos K. Aguilera and Hakim Weatherspoon, editors, *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, pages 701–718. USENIX Association, 2022.

[160] Wen Zhang, Scott Shenker, and Irene Zhang. Persistent state machines for recoverable in-memory storage systems with nvram. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, pages 1029–1046. USENIX Association, 2020.

[161] Xing Zhang. MySQL 8.0.1: Accent and case sensitive collations for utf8mb4, 2017. Retrieved Nov 7, 2024 from `https://dev.mysql.com/blog-archive/mysql-8-0-1-accent-and-case-sensitive-collations-for-utf8mb4/`.

[162] Zheng Zhang and Alberto O. Mendelzon. Authorization views and conditional query containment. In Thomas Eiter and Leonid Libkin, editors, *Database Theory - ICDT 2005, 10th International Conference, Edinburgh, UK, January 5-7, 2005, Proceedings*, volume 3363 of *Lecture Notes in Computer Science*, pages 259–273. Springer, 2005.

[163] Zeljka Zorz. OpenEMR vulnerabilities put patients' info, medical records at risk, 2018. https://www.helpnetsecurity.com/2018/08/08/openemr-vulnerabilities/.