Access Control for Database Applications: Beyond Policy Enforcement

Wen Zhang
UC Berkeley
Berkeley, CA, USA
zhangwen@cs.berkeley.edu

Aurojit Panda
NYU
New York, NY, USA
apanda@cs.nyu.edu

Scott Shenker
UC Berkeley & ICSI
Berkeley, CA, USA
shenker@icsi.berkeley.edu

ABSTRACT

There have been many recent advances in enforcing fine-grained access control for database-backed applications. However, operators face significant challenges both before and after an enforcement mechanism has been deployed. We identify three such challenges *beyond enforcement* and discuss possible solutions.

CCS CONCEPTS

• **Security and privacy** → **Information accountability and usage control**; **Access control**.

KEYWORDS

Access control, database-backed applications

ACM Reference Format:
Wen Zhang, Aurojit Panda, and Scott Shenker. 2023. Access Control for Database Applications: Beyond Policy Enforcement. In *Workshop on Hot Topics in Operating Systems (HotOS '23), June 22–24, 2023, Providence, RI, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3593856.3595905

1 INTRODUCTION

Many modern applications store sensitive data in databases. Today, it is often the application's responsibility to ensure that such sensitive data is shown only to authorized users, but application bugs can lead to sensitive data being improperly disclosed [4, 16, 17, 26, 44]. This issue can be mitigated by applying the principle of least privilege,[1] which in this context entails giving an application access only to the database content that the user being served is allowed to see.

[1]This principle states that: "every privileged user of the system should operate using the least amount of privilege necessary" [40].

Applying the principle of least privilege requires enforcing per-user access control outside the application. Access control for databases is a well-trodden research area, and there have been many proposals for secure and efficient access-control *enforcement* [9, 18, 19, 25, 27, 45, 49, 50]—i.e., ensuring that the database reveals only data allowed by a *data-access policy*. Once deployed, these mechanisms are indeed effective in preventing unauthorized data access.

And yet, if we consider what it takes to get these enforcement mechanisms deployed in practice, we realize that enforcement is not the end of the access-control story—it is not even the beginning of it. In fact, challenges arise *before* enforcement can be deployed, and remain *after* enforcement is in place. For example, consider the following questions:

(1) Given an existing application, how can we infer what policy it was designed to enforce?
(2) Given a policy, how can we validate that it sufficiently protects sensitive data?
(3) When a program submits inappropriate queries, how can we help the developer find and resolve the issue?

Until we have answered such questions, we have not addressed the full life-cycle of controlling access to sensitive data. It is important for access-control research to *look beyond enforcement design* and explore the challenges that arise as enforcement mechanisms become used in practice.

2 BACKGROUND AND MOTIVATION

2.1 A Primer on Policy Enforcement

Before discussing issues beyond policy enforcement, we first provide some brief background on enforcement itself.

The goal of access-control enforcement is to restrict an application's data accesses to the data allowed by a given *policy*. An enforcement design must answer two questions:

(1) At what granularity is a policy specified?
(2) How is a policy enforced?

And the many enforcement designs in literature are largely distinguished by their answers to these questions—e.g.:

(1) Policies (i.e., what data can be accessed) can be specified at the granularity of table columns [42], rows [25, 32], cells [18, 19, 25, 32, 49], views [5, 29, 39, 50], etc.

(2) Policy enforcement can be static [9, 19, 42] vs dynamic [18, 25, 32, 49, 50], query-modifying [18, 25, 27, 49] vs semantic-preserving [9, 19, 50], etc.

These designs offers different trade-offs among policy expressiveness, ease of use, and performance.

2.2 A Concrete Setting

All the challenges we identify in this paper apply to a range of enforcement scenarios. But for concreteness, we frame our discussion in the setting of Blockaid [50], a recent enforcement solution proposed in the systems community.

Blockaid enforces access control for applications that store sensitive data in a relational database. A policy is specified using SQL views (parameterized by user ID), and is enforced using a SQL proxy that intercepts each application-issued query, allowing it if its answer is guaranteed to reveal no more information than the views do, and blocking it otherwise. When vetting a query, the proxy considers the history of prior queries and their results; this allows it to safely allow certain queries that would be blocked in isolation.

Example 2.1 (Calendar application [51]). Consider a policy for a calendar application consisting of two views:

(V_1) `SELECT EId FROM Attendance WHERE UId = ?MyUId`
(Each user can see the IDs of events they attend.)

(V_2) `SELECT * FROM Events e JOIN Attendance a ON e.EId = a.EId WHERE a.UId = ?MyUId`
(Each user can see the details of events they attend.)

Suppose the application, when serving a user with `UId` = 1, issues the following sequence of queries:

(Q_1) `SELECT 1 FROM Attendance WHERE UId=1 AND EId=2`
(Does User #1—the current user—attend Event #2?)
This query is allowed under the policy because the information it reveals is covered by View V_1.
Now, suppose the database returns one row, indicating the current user does attend the event.

(Q_2) `SELECT * FROM Events WHERE EId=2`
(Fetches details about Event #2.)
This query is allowed *given that Q_1 has returned one row*, an assumption that guarantees the information revealed by this query is covered by View V_2.

If not for the history of Q_1 returning non-empty, Q_2 would have been blocked: by itself, Q_2 is not *guaranteed* to reveal only information covered by V_2. ◀

We stress two traits of Blockaid's design. First, a query is either executed as-is or blocked outright; it is never executed with modification. Second, with Blockaid the application must still contain access checks; Blockaid merely ensures that the checks are sufficient to confine data revelation to what the policy allows. The specifics of the research directions laid out in this paper will sometimes rely on these two traits, but every challenge we identify has equivalents in other settings.

2.3 Challenges Beyond Enforcement

Enforcement provides the security guarantees that rule out unauthorized data access, and so it is deservedly the central piece of the access-control puzzle. But it is not the only piece that matters. As access control is used in practice, challenges can arise both *before* and *after* enforcement. In what follows, we identify three such challenges beyond enforcement that are crucial to the life-cycle of data access control and, in each case, propose potential solutions.

3 POLICY CREATION

3.1 Challenge: Creating a Policy for an Existing Application

Ideally, a widely-applicable enforcement mechanism should apply to a range of *currently deployed* applications. However, before such an enforcement mechanism can be installed, an administrator—we'll call her Dora—must come up with a policy that grants the application access to the minimal amount of information it needs. While existing applications employ data-access checks internally (e.g., in the form of if-statements), they typically do not come with a data-access policy specified for external access control. And so Dora faces her first obstacle: how to come up with a policy that captures the data accesses required by an existing application?

The most immediate way is for Dora to sit down and study the application's intended data accesses, and then write down the most restrictive set of views that allows them. Indeed, this is what we did for Blockaid's evaluation [50, §8.1].

But manual policy creation is tedious: Dora, who may not be application's developer, must painstakingly map out possible application behaviors, including edge cases. What is worse, tedium leads to errors: a human who has to wade through thousands of lines of application code is prone to typos and omissions, which can lead to the policy being overly restrictive or overly permissive. (We made at least one such mistake, which remained undetected for a while, when we wrote policies manually in the past [50, §8.7].)

Challenge 1. Systematically determine the policy embodied in an existing application.

3.2 Proposal: Policy Extraction

Part of the struggle is that Dora must write a policy from scratch. What she had a tool that could generate a *draft policy* from the application, to be then refined into its final form?

Such a tool is plausible because legacy applications already contain enough information to produce a draft policy. Often, developers use access checks and filters to query only the

```
def show_event(db, params, session):
  if db.sql("SELECT 1 FROM Attendance "
            "WHERE UId = ? AND EId = ?",
        session["user_id"], params["event_id"]
      ).is_empty():
    raise Http404("event not found")
  return format_event(db.sql(
    "SELECT * FROM Events WHERE EId = ?"
    params["event_id"]))
```

Listing 1: A handler that displays a calendar event.

data required for a functionality (e.g., rendering a calendar event). The tool need simply extract this existing behavior.

Example 3.1. Continuing from Example 2.1, Listing 1 shows a code snippet for rendering a calendar event. (This could be the code that issued Q_1–Q_2.) Dora, by studying the code or observing queries issued by runs of this code, can infer the policy V_1–V_2, which is in fact a *maximally restrictive* policy that allows this functionality, as prescribed by the principle of least privilege. We wish to automate this process. ◀

We propose to aid in policy creation for legacy applications via *policy extraction*: automatically generating a maximally restrictive policy that allows the application's current behavior. A human can then examine the extracted policy to see if any view is too permissive; if so, the program behavior that led to the view's production is likely a bug.

There are many ways to implement such an extraction tool; here we outline two ways forward.

3.2.1 Language-based extraction: Symbolic execution. Assuming the application's source code is available, we can symbolically execute it to find (1) what SQL queries can be issued, and (2) under what conditions each SQL query is issued (i.e., a path condition for each query-issuing statement). We can then generate a policy to allow only these queries, and only under their respective conditions.

At first glance, it is unclear if this approach would work: symbolic execution can be impractically slow due to path explosion [8], a challenge exacerbated by web applications' usage of dynamic languages (like Ruby). But we have hope:

- Path explosion is often caused by loops [31]; but as prior work has observed, web applications typically have simple loop structure [9, 43].
- Prior work was able to symbolically execute code in a dynamic language without re-implementing language features, by co-opting the language's standard interpreter [30]; we could leverage similar techniques here.

An extraction tool based on symbolic execution would have to be language-specific. Fortunately, server-side web programming is dominated by just a few popular languages and technologies like PHP, .NET, and Ruby [47], and so even a tool specific to one of these languages can have wide impact.

There has been much prior work using code analysis to explore an application's data-access behavior, two example being privilege inference via taint tracking [7] and vulnerability detection via dynamic invariant generation [13]. Unlike these systems, policy extraction based on symbolic execution would be able to systematically explore program paths without relying on Dora to craft comprehensive test cases.

3.2.2 Language-agnostic extraction: Specification mining. If application source code is unavailable, or if we insist on language-agnostic extraction, then we must resort to black-box techniques. A black-box extraction tool would execute the application on inputs of its choice and observe the queries issued and answers returned, before outputting a policy.

If we regard the policy as a "specification" of the application's data accesses, then our extraction task becomes a task of dynamic *specification mining* [2]: inferring likely specifications by analyzing an application's behaviors [23]. But unlike prior efforts that extract FSM [2, 48] or LTL [20] specifications, or those that infer specifications for social-network updates [24] or syscall sandboxing [33], we seek to extract SQL views that summarize the application's data queries.

A black-box extraction tool must carry out two steps. First, it must run the application and collect query traces. Here, it is crucial to achieve good coverage. If the application has a comprehensive test suite, it may suffice to just run the suite. Otherwise, we could leverage test generation [3], guided fuzzing [15], or active learning [43] to achieve good coverage.

Second, the tool must "learn" a policy from the traces in a *generalizing* way—i.e., produce a policy that allows not only the traces observed so far, but also any other trace generated by the same application logic. For Example 2.1, a generalizing learner would extrapolate the trace Q_1–Q_2, which is specific to User #1 and Event #2, into views V_1–V_2, which are generic over users and events. But it must not over-generalize by, say, allowing any user to view any event regardless of attendance.

So how can we design an extraction algorithm that generalizes, but not too much? While in theory this problem might be unsolvable without the ground truth for application behavior, a few practical approaches look promising:

- Limit the size of the generated policy. A policy that relies on non-generalizing views must, by nature, contain a lot of them (e.g., one for each user in the database). This can be avoided by using a large collection of traces but insisting that the generated policy be small.
- Solicit hints on what can appear in a policy—e.g., require that a *concrete* event ID (like `EId=2`) never appear in a policy, since event IDs are opaque identifiers. With this rule, traces will be generalized across all events.

- Actively discover which constraints in a trace to keep. For example, unsure if an `Attendance` row's `notes` value matters to data-access checking, the algorithm can re-run the application with the `notes` cell mutated to a random string. If the subsequent trace is unaffected, it can conclude that `notes` does not affect if an event can be accessed, and omit it from the policy.

3.3 Applicability and Usefulness

We end this section by discussing when policy extraction applies, and why we expect it to be useful to Dora.

Legacy vs new applications. Policy extraction simplifies policy creation for *legacy applications*, ones developed without externally-enforced access control in mind. For a new application whose policy is produced alongside its business logic—perhaps using a policy-integrated language like Hails [14] or Scooter [37]—we imagine policy creation would be less burdensome, and policy extraction likely unnecessary.

Applicability under query modification. Policy extraction does not apply to programs written to work with query-modifying access control [18, 25, 27, 49]. These programs issue broad queries for any data they could possibly use, and rely on an external entity to rid answers of disallowed data. Without using access checks or query filters, these programs contain no information from which to extract a policy. But the developer is already required to produce a policy as the program is written, so no extraction is necessary.

Vetting an extracted policy. A policy extraction tool cannot tell whether an extracted policy reflects human intentions; this question must be answered by Dora. But why would Dora have an easier time vetting an extracted policy than she would writing one from scratch?

From our experience writing view-based policies for web applications, the meaning of each view has often been easy to interpret, and the rationale behind it easy to ascertain—*but only in retrospect*, after the view definition has been written down. Coming up with the view in the first place can be hard, especially if the view is needed only for an edge case, and if the policy writer is not intimately familiar with the application. Policy extraction helps by laying out the views required by the nooks and crannies of the application's logic.

Finally, once we extract a policy, we could evaluate it for sensitive-data disclosure, a direction we explore next.

4 POLICY EVALUATION

4.1 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 our operator, Dora, wants hidden). In many cases, no policy satisfies both imperatives. So how can Dora 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 Dora wants hidden (we will refer to such queries as *sensitive queries*). Dora 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 [28].

Example 4.1 ([11, §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 2. 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.

4.2 Existing Work: Bayesian Privacy

One of the most well-studied notions of disclosure in the database literature is that of Bayesian privacy [11, 28], 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 4.1); 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 [28]), 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. [10]), 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.

4.3 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 [34], 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)* [6, 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:

- $\text{PQI}_S(\mathbf{V})$ holds if revealing the contents of $\mathbf{V}$ could render a possible answer to S certain.
- $\text{NQI}_S(\mathbf{V})$ 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 4.2. 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, $\text{PQI}_{Q_2}(\{Q_1\})$ 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 $\text{NQI}_{Q_1}(\{Q_2\})$.

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

Remark 4.3. 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.

Once Dora finds PQI or NQI, or some other condition, to be a useful criterion for her application, she can write down her sensitive queries and ensure that a policy is disclosure-free by invoking a checking algorithm for the criterion.

It remains to develop checking algorithms prior-agnostic privacy. As far as we know, algorithms for checking PQI and NQI have been studied only in theoretical contexts for simple, conjunctive queries [6]. Practical algorithms exist for checking k-anonymity [41, 46] (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 VIOLATION DIAGNOSIS

5.1 Challenge: Troubleshooting Violations

Having produced a policy she's happy with, Dora 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),[2] 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 Dora 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.[3] However, while a counterexample is a proof-of-violation, it is not easily interpreted by Dora—what is she to do with two databases shown side by side?

Challenge 3. 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 Dora to forgo access control out of frustration or, worse, to silence violations by setting overly permissive policies, leaving data unprotected.

5.2 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

[2] 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.
[3] 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 [50, §4.2].

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.

5.2.1 Patching the policy. Policy patches, consisting of modified/added view definitions, can be generated via policy extraction (Section 3.2): 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.

5.2.2 Patching the application. A typical application patch would take one of two forms:

(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` statement in Listing 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.[4]

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 [22]—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.[5] 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) [21, 36] and CQs with arithmetic comparisons [1]. 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 2.1, if Q_2 were issued alone (it would be blocked), one such statement would be "the `Attendance` table contains row `(UId=1, EId=2)`", which the developer can check for in her code before issuing the query.

[4]This is in contrast to leak repair for liquid information flow control [35, §5], which statically analyzes source code written in a special type system.
[5]More precisely, we only need the latter condition to hold on all databases consistent with the trace prior to Q's issuance.

The search for such a statement falls under *abductive inference*: finding an "explanatory hypothesis for a desired outcome" [12], with the desired outcome being policy compliance for the blocked query. As such, a promising approach is to leverage program synthesis techniques for abduction [38].

6 CONCLUSION

Data-access policy enforcement has received much attention and seen great advances in recent years, but less attention has been paid to issues that arise before and after enforcement is deployed. We hope this paper will spur further research on *challenges beyond enforcement*—like policy creation, policy evaluation, and violation diagnosis—whose resolution will be crucial to addressing the full life-cycle of access control.

ACKNOWLEDGMENTS

We thank the reviewers for their comments, Victor Vianu for discussions on database privacy, and Lucas Lovric and Phillip Lovric for discussions on policy extraction. This work was supported by NSF grants 1704941 and 2145471, and by funding from Intel, VMware, Ericsson, Google, and IBM.

REFERENCES

[1] Foto N. Afrati, Chen Li, and Prasenjit Mitra. 2006. Rewriting queries using views in the presence of arithmetic comparisons. *Theor. Comput. Sci.* 368, 1-2 (2006), 88–123. https://doi.org/10.1016/j.tcs.2006.08.020
[2] Glenn Ammons, Rastislav Bodík, and James R. Larus. 2002. Mining specifications. In *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002*, John Launchbury and John C. Mitchell (Eds.). ACM, 4–16. https://doi.org/10.1145/503272.503275
[3] Shay Artzi, Adam Kiezun, Julian Dolby, Frank Tip, Danny Dig, Amit M. Paradkar, and Michael D. Ernst. 2008. Finding bugs in dynamic web applications. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2008, Seattle, WA, USA, July 20-24, 2008*, Barbara G. Ryder and Andreas Zeller (Eds.). ACM, 261–272. https://doi.org/10.1145/1390630.1390662
[4] Warwick Ashford. 2015. *Facebook photo leak flaw raises security concerns.* https://www.computerweekly.com/news/2240242708/Facebook-photo-leak-flaw-raises-security-concerns
[5] Gabriel Bender, Lucja Kot, Johannes Gehrke, and Christoph Koch. 2013. Fine-grained disclosure control for app ecosystems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias (Eds.). ACM, 869–880. https://doi.org/10.1145/2463676.2467798
[6] Michael Benedikt, Pierre Bourhis, Balder ten Cate, Gabriele Puppis, and Michael Vanden Boom. 2021. Inference from Visible Information and Background Knowledge. *ACM Trans. Comput. Log.* 22, 2 (2021), 13:1–13:69. https://doi.org/10.1145/3452919
[7] Aaron Blankstein and Michael J. Freedman. 2014. Automating Isolation and Least Privilege in Web Services. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014.* IEEE Computer Society, 133–148. https://doi.org/10.1109/SP.2014.16
[8] James Bornholt and Emina Torlak. 2018. Finding code that explodes under symbolic evaluation. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 149:1–149:26. https://doi.org/10.1145/3276519

[9] Adam Chlipala. 2010. Static Checking of Dynamically-Varying Security Policies in Database-Backed Applications. In *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings*, Remzi H. Arpaci-Dusseau and Brad Chen (Eds.). USENIX Association, 105–118. http://www.usenix.org/events/osdi10/tech/full_papers/Chlipala.pdf

[10] Nilesh N. Dalvi, Gerome Miklau, and Dan Suciu. 2005. Asymptotic Conditional Probabilities for Conjunctive Queries. In *Database Theory - ICDT 2005, 10th International Conference, Edinburgh, UK, January 5-7, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3363)*, Thomas Eiter and Leonid Libkin (Eds.). Springer, 289–305. https://doi.org/10.1007/978-3-540-30570-5_20

[11] Alin Deutsch. 2008. Privacy in Database Publishing: A Bayesian Perspective. In *Handbook of Database Security - Applications and Trends*, Michael Gertz and Sushil Jajodia (Eds.). Springer, 461–487. https://doi.org/10.1007/978-0-387-48533-1_19

[12] Isil Dillig, Thomas Dillig, and Alex Aiken. 2012. Automated error diagnosis using abductive inference. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, Jan Vitek, Haibo Lin, and Frank Tip (Eds.). ACM, 181–192. https://doi.org/10.1145/2254064.2254087

[13] Viktoria Felmetsger, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. 2010. Toward Automated Detection of Logic Vulnerabilities in Web Applications. In *19th USENIX Security Symposium, Washington, DC, USA, August 11-13, 2010, Proceedings*. USENIX Association, 143–160. http://www.usenix.org/events/sec10/tech/full_papers/Felmetsger.pdf

[14] Daniel B. Giffin, Amit Levy, Deian Stefan, David Terei, David Mazières, John C. Mitchell, and Alejandro Russo. 2012. Hails: Protecting Data Privacy in Untrusted Web Applications. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, Chandu Thekkath and Amin Vahdat (Eds.). USENIX Association, 47–60. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/giffin

[15] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. 2008. Automated Whitebox Fuzz Testing. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2008, San Diego, California, USA, 10th February - 13th February 2008*. The Internet Society. https://www.ndss-symposium.org/ndss2008/automated-whitebox-fuzz-testing/

[16] Eddie Kohler. 2013. *Hide review rounds from paper authors*. https://github.com/kohler/hotcrp/commit/5d53ab

[17] Eddie Kohler. 2015. *Download PC review assignments obeys paper administrators*. https://github.com/kohler/hotcrp/commit/80ff96

[18] Kristen LeFevre, Rakesh Agrawal, Vuk Ercegovac, Raghu Ramakrishnan, Yirong Xu, and David J. DeWitt. 2004. Limiting Disclosure in Hippocratic Databases. In *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB 2004, Toronto, Canada, August 31 - September 3 2004*, Mario A. Nascimento, M. Tamer Özsu, Donald Kossmann, Renée J. Miller, José A. Blakeley, and K. Bernhard Schiefer (Eds.). Morgan Kaufmann, 108–119. https://doi.org/10.1016/B978-012088469-8.50013-9

[19] Nico Lehmann, Rose Kunkel, Jordan Brown, Jean Yang, Niki Vazou, Nadia Polikarpova, Deian Stefan, and Ranjit Jhala. 2021. STORM: Refinement Types for Secure Web Applications. In *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*, Angela Demke Brown and Jay R. Lorch (Eds.). USENIX Association, 441–459. https://www.usenix.org/conference/osdi21/presentation/lehmann

[20] Caroline Lemieux, Dennis Park, and Ivan Beschastnikh. 2015. General LTL Specification Mining (T). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, Myra B. Cohen, Lars Grunske, and Michael Whalen (Eds.). IEEE Computer Society, 81–92. https://doi.org/10.1109/ASE.2015.71

[21] Alon Y. Levy, Alberto O. Mendelzon, Yehoshua Sagiv, and Divesh Srivastava. 1995. Answering Queries Using Views. In *Proceedings of the Fourteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 22-25, 1995, San Jose, California, USA*, Mihalis Yannakakis and Serge Abiteboul (Eds.). ACM Press, 95–104. https://doi.org/10.1145/212433.220198

[22] Alon Y. Levy, Anand Rajaraman, and Joann J. Ordille. 1996. Querying Heterogeneous Information Sources Using Source Descriptions. In *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*, T. M. Vijayaraman, Alejandro P. Buchmann, C. Mohan, and Nandlal L. Sarda (Eds.). Morgan Kaufmann, 251–262. http://www.vldb.org/conf/1996/P251.PDF

[23] Wenchao Li. 2014. *Specification Mining: New Formalisms, Algorithms and Applications*. Ph. D. Dissertation. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-20.html

[24] Paul Marinescu, Chad Parry, Marjori Pomarole, Yuan Tian, Patrick Tague, and Ioannis Papagiannis. 2017. IVD: Automatic Learning and Enforcement of Authorization Rules in Online Social Networks. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*. IEEE Computer Society, 1094–1109. https://doi.org/10.1109/SP.2017.33

[25] Alana Marzoev, Lara Timbó Araújo, Malte Schwarzkopf, Samyukta Yagati, Eddie Kohler, Robert Tappan Morris, M. Frans Kaashoek, and Sam Madden. 2019. Towards Multiverse Databases. In *Proceedings of the Workshop on Hot Topics in Operating Systems, HotOS 2019, Bertinoro, Italy, May 13-15, 2019*. ACM, 88–95. https://doi.org/10.1145/3317550.3321425

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

[27] Aastha Mehta, Eslam Elnikety, Katura Harvey, Deepak Garg, and Peter Druschel. 2017. Qapla: Policy compliance for database-backed systems. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, Engin Kirda and Thomas Ristenpart (Eds.). USENIX Association, 1463–1479. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/mehta

[28] Gerome Miklau and Dan Suciu. 2004. A Formal Analysis of Information Disclosure in Data Exchange. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*, Gerhard Weikum, Arnd Christian König, and Stefan Deßloch (Eds.). ACM, 575–586. https://doi.org/10.1145/1007568.1007633

[29] Amihai Motro. 1989. 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*. IEEE Computer Society, 339–347. https://doi.org/10.1109/ICDE.1989.47234

[30] Joseph P. Near and Daniel Jackson. 2014. *Symbolic Execution for (Almost) Free: Hijacking an Existing Implementation to Perform Symbolic Execution*. Technical Report MIT-CSAIL-TR-2014-007. CSAIL, Massachusetts Institute of Technology, Cambridge, MA. http://hdl.handle.net/1721.1/86235

[31] Jan Obdrzálek and Marek Trtík. 2011. Efficient Loop Navigation for Symbolic Execution. In *Automated Technology for Verification and Analysis, 9th International Symposium, ATVA 2011, Taipei, Taiwan, October 11-14, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6996)*, Tevfik Bultan and Pao-Ann Hsiung (Eds.). Springer, 453–462.

https://doi.org/10.1007/978-3-642-24372-1_34

[32] Oracle. 2017. *Using Oracle Virtual Private Database to Control Data Access.* https://docs.oracle.com/database/121/DBSEG/vpd.htm

[33] Shankara Pailoor, Xinyu Wang, Hovav Shacham, and Isil Dillig. 2020. Automated policy synthesis for system call sandboxing. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 135:1–135:26. https://doi.org/10.1145/3428203

[34] Vern Paxson and Sally Floyd. 1994. Wide-Area Traffic: The Failure of Poisson Modeling. In *Proceedings of the ACM SIGCOMM 1994 Conference on Communications Architectures, Protocols and Applications, London, UK, August 31 - September 2, 1994*, Jon Crowcroft (Ed.). ACM, 257–268. https://doi.org/10.1145/190314.190338

[35] Nadia Polikarpova, Deian Stefan, Jean Yang, Shachar Itzhaky, Travis Hance, and Armando Solar-Lezama. 2020. Liquid information flow control. *Proc. ACM Program. Lang.* 4, ICFP (2020), 105:1–105:30. https://doi.org/10.1145/3408987

[36] Anand Rajaraman, Yehoshua Sagiv, and Jeffrey D. Ullman. 1995. Answering Queries Using Templates with Binding Patterns. In *Proceedings of the Fourteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 22-25, 1995, San Jose, California, USA*, Mihalis Yannakakis and Serge Abiteboul (Eds.). ACM Press, 105–112. https://doi.org/10.1145/212433.220199

[37] John Renner, Alex Sanchez-Stern, Fraser Brown, Sorin Lerner, and Deian Stefan. 2021. Scooter & Sidecar: a domain-specific approach to writing secure database migrations. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 710–724. https://doi.org/10.1145/3453483.3454072

[38] Andrew Reynolds, Haniel Barbosa, Daniel Larraz, and Cesare Tinelli. 2020. Scalable Algorithms for Abduction via Enumerative Syntax-Guided Synthesis. In *Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 12166)*, Nicolas Peltier and Viorica Sofronie-Stokkermans (Eds.). Springer, 141–160. https://doi.org/10.1007/978-3-030-51074-9_9

[39] Shariq Rizvi, Alberto O. Mendelzon, S. Sudarshan, and Prasan Roy. 2004. Extending Query Rewriting Techniques for Fine-Grained Access Control. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*, Gerhard Weikum, Arnd Christian König, and Stefan Deßloch (Eds.). ACM, 551–562. https://doi.org/10.1145/1007568.1007631

[40] Jerome H. Saltzer. 1974. Protection and the Control of Information Sharing in Multics. *Commun. ACM* 17, 7 (1974), 388–402. https://doi.org/10.1145/361011.361067

[41] Pierangela Samarati. 2001. Protecting Respondents' Identities in Microdata Release. *IEEE Trans. Knowl. Data Eng.* 13, 6 (2001), 1010–1027. https://doi.org/10.1109/69.971193

[42] Daniel Schoepe, Daniel Hedin, and Andrei Sabelfeld. 2014. SeLINQ: tracking information across application-database boundaries. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, Johan Jeuring and Manuel M. T. Chakravarty (Eds.). ACM, 25–38. https://doi.org/10.1145/2628136.2628151

[43] Jiasi Shen and Martin C. Rinard. 2019. Using active learning to synthesize models of applications that access databases. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 269–285. https://doi.org/10.1145/3314221.3314591

[44] Ben Stock. 2018. *Search leaks hidden tags.* https://github.com/kohler/hotcrp/issues/135

[45] Michael Stonebraker and Eugene Wong. 1974. Access control in a relational data base management system by query modification. In *Proceedings of the 1974 ACM Annual Conference, San Diego, California, USA, November 1974, Volume 1*, Roger C. Brown and Donald E. Glaze (Eds.). ACM, 180–186. https://doi.org/10.1145/800182.810400

[46] Latanya Sweeney. 2002. K-Anonymity: A Model for Protecting Privacy. *Int. J. Uncertain. Fuzziness Knowl.-Based Syst.* 10, 5 (2002).

[47] W3Techs. 2023. *Usage statistics of server-side programming languages for websites.* https://w3techs.com/technologies/overview/programming_language

[48] Westley Weimer and George C. Necula. 2005. Mining Temporal Specifications for Error Detection. In *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3440)*, Nicolas Halbwachs and Lenore D. Zuck (Eds.). Springer, 461–476. https://doi.org/10.1007/978-3-540-31980-1_30

[49] Jean Yang, Travis Hance, Thomas H. Austin, Armando Solar-Lezama, Cormac Flanagan, and Stephen Chong. 2016. Precise, dynamic information flow for database-backed applications. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, Chandra Krintz and Emery D. Berger (Eds.). ACM, 631–647. https://doi.org/10.1145/2908080.2908098

[50] Wen Zhang, Eric Sheng, Michael Alan Chang, Aurojit Panda, Mooly Sagiv, and Scott Shenker. 2022. Blockaid: Data Access Policy Enforcement for Web Applications. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, Marcos K. Aguilera and Hakim Weatherspoon (Eds.). USENIX Association, 701–718. https://www.usenix.org/conference/osdi22/presentation/zhang

[51] Wen Zhang, Eric Sheng, Michael Alan Chang, Aurojit Panda, Mooly Sagiv, and Scott Shenker. 2022. *Blockaid: Data Access Policy Enforcement for Web Applications (slides).* https://www.usenix.org/sites/default/files/conference/protected-files/osdi22_slides_zhang-wen.pdf