Pointer Analysis for Database-Backed Applications

YUFEI LIANG, TENG ZHANG, GANLIN LI, and TIAN TAN*, Nanjing University, China
CHANG XU, CHUN CAO, XIAOXING MA, and YUE LI*, Nanjing University, China

Database-backed applications form the backbone of modern software, yet their complexity poses significant challenges for static analysis. These applications involve intricate interactions among application code, diverse database frameworks such as JDBC, Hibernate, and Spring Data JPA, and languages like Java and SQL. In this paper, we introduce DBridge, the first pointer analysis specifically designed for Java database-backed applications, capable of statically constructing comprehensive Java-to-database value flows. DBridge unifies application code analysis, database access specification modeling, SQL analysis, and database abstraction within a single pointer analysis framework, capturing interactions across a wide range of database access APIs and frameworks. Additionally, we present DB-Micro, a new micro-benchmark suite with 824 test cases crafted to systematically evaluate static analysis for database-backed applications. Experiments on DB-Micro and large, complex, real-world applications demonstrate DBridge's effectiveness, achieving high recall and precision in building Java-to-database value flows efficiently and outperforming state-of-the-art tools in SQL statement identification. To further validate DBridge's utility, we develop three client analyses for security and program understanding. Evaluation on these real-world applications reveals 30 Stored XSS attack vulnerabilities and 3 horizontal broken access control vulnerabilities, *all previously undiscovered and real*, as well as a high detection rate in impact analysis for schema changes. By open-sourcing DBridge (14K LoC) and DB-Micro (22K LoC), we seek to help advance static analysis for modern database-backed applications in the future.

CCS Concepts: • **Theory of computation → Program analysis**.

Additional Key Words and Phrases: Static Analysis, Pointer Analysis, Database Framework, Java

ACM Reference Format:
Yufei Liang, Teng Zhang, Ganlin Li, Tian Tan, Chang Xu, Chun Cao, Xiaoxing Ma, and Yue Li. 2025. Pointer Analysis for Database-Backed Applications. *Proc. ACM Program. Lang.* 9, PLDI, Article 204 (June 2025), 25 pages. https://doi.org/10.1145/3729307

1 Introduction

Database-backed applications (referred to as apps) are central to modern software, offering structured and efficient mechanisms for data management. In the Java ecosystem, these apps rely on database access APIs, such as Java Database Connectivity (JDBC) [2], to bridge the app layer with underlying databases. However, growing enterprise demands and the adoption of frameworks like Hibernate [1] and Spring Data JPA [5] have significantly complicated Java-to-database interactions, involving complex specifications and diverse implementations [14]. This complexity poses challenges for developers in understanding and maintaining the intricate value flows between Java apps and databases, especially as codebases grow in scale and interdependency. These challenges

*Corresponding authors.

Authors' Contact Information: Yufei Liang, 602024330013@smail.nju.edu.cn; Teng Zhang, dz21330044@smail.nju.edu.cn; Ganlin Li, 502022320006@smail.nju.edu.cn; Tian Tan, tiantan@nju.edu.cn, State Key Laboratory for Novel Software Technology, Nanjing University, China; Chang Xu, changxu@nju.edu.cn; Chun Cao, caochun@nju.edu.cn; Xiaoxing Ma, xxm@nju.edu.cn; Yue Li, yueli@nju.edu.cn, State Key Laboratory for Novel Software Technology, Nanjing University, China.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2025 Copyright held by the owner/author(s).
ACM 2475-1421/2025/6-ART204
https://doi.org/10.1145/3729307

lead to higher maintenance costs and security risks, such as broken access controls and injection vulnerabilities, which are among the top risks in the OWASP Web Application Security list [4].

Despite advances in static analysis for security and program understanding [9, 11, 24, 35, 38], current approaches face substantial limitations when applied to the full spectrum of Java-to-database interactions. Fig. 1 shows common scenarios in which a database-backed app interacts with a database. Typically, such apps rely on frameworks like Spring [6], using features like Inversion of Control (IoC) for logical task management, and communicate with databases via JDBC, which uses SQL for actual data operations. To ease development, programmers often use advanced database access specifications like JPA or Spring Data JPA, with the latter acting as a sophisticated wrapper for the former. Prominent implementations of these specifications include frameworks like Hibernate and Spring Data JPA. Other specifications, like MyBatis, are simpler and can be addressed similarly, so they are omitted in this work.

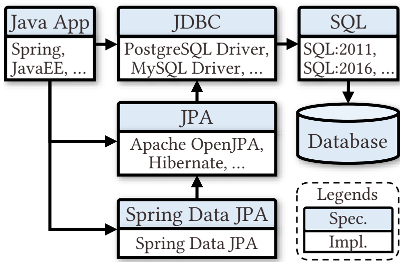


Fig. 1. Java-to-database interactions.

Statically capturing *Java-to-database* value flows requires tracing the flow of data *from app code, through various database access APIs in frameworks, via SQL manipulations, to the database, and then back to the app code.* This full-spectrum value flow analysis is essential for tools that support tasks like security analysis and program understanding in database-backed apps. Yet, no existing database-related static analysis approach [9, 15, 21, 27, 28, 31, 32, 45] can capture these fundamental value flows. The challenge arises mainly from the complexity of database frameworks, which involve intricate API behaviors shaped by factors like entity relationships, object mappings, and persistence states. Analyzing these flows requires approximating not only database accesses but also app behavior, SQL semantics, and database structure—complicating the balance between analysis soundness and precision (where better soundness means capturing more program behaviors).

To construct Java-to-database value flows, we rely on pointer analysis, a fundamental static analysis upon which most client analyses are based [33, 37]. Accordingly, we introduce DBridge, the first pointer analysis specifically crafted to comprehensively analyze Java-to-database interactions. DBridge comprises three main components: (1) the *application analyzer*, which handles Java app code, including enterprise features and its interactions with database access APIs; (2) the *database framework analyzer*, which models complex processes within database frameworks, transforming database access API calls into a set of framework primitives to uniformly express API semantics across frameworks like JDBC, Hibernate, and Spring Data JPA; (3) the *database analyzer*, which converts the output of database framework analyzer into SQL primitives, approximating SQL execution and statically tracking value flows in the database based on an abstracted database model. Each component of DBridge extends traditional pointer analysis by introducing rules tailored to the newly proposed framework and SQL primitives. This enables DBridge to address the diversity and complexity of constructing Java-to-database value flows, supporting client analyses like security analysis and program understanding, thereby enhancing security and reducing maintenance complexity in database-backed apps. In summary, this paper makes the following contributions.

- We present DBridge, the first pointer analysis approach designed for Java database-backed apps, capable of constructing comprehensive value flows across the full spectrum of Java-to-database interactions: this includes handling app code (and relevant enterprise framework features like Spring's IoC), complex database access APIs from various database frameworks, SQL syntax and semantics, and the abstraction of the underlying database.

- We introduce DB-Micro, the first micro-benchmark suite to systematically evaluate static analysis for database-backed apps. DB-Micro includes 824 carefully crafted test cases covering key SQL features and database access APIs from JDBC, JPA, and Spring Data JPA, essential for constructing value flows in static analysis. DBridge successfully passed 721 out of 824 test cases, demonstrating robustness in handling diverse language constructs and framework use cases.
- We further evaluate DBridge on ten popular real-world database-backed apps. DBridge achieves an average recall of 90% (high soundness) and precision of 75% in constructing Java-to-database value flows. Additionally, DBridge significantly outperforms SLocator [21], a state-of-the-art approach for SQL statement identification, a fundamental analysis task that is a primary focus in existing static analysis for database-backed apps.
- To validate DBridge's practical utility as a foundational tool, we develop three client analyses: two security tools—a Stored XSS attack analysis and a horizontal broken access control vulnerability analysis—and a program understanding tool, an impact analysis for database schema changes. Evaluation on popular apps reveals 30 Stored XSS attack vulnerabilities and 3 horizontal broken access control vulnerabilities, *all previously undiscovered and real*, as well as a high detection rate in identifying 186 out of 195 database access sites affected by schema changes.
- We built DBridge on Tai-e, a state-of-the-art static analysis framework for Java, and fully open-sourced DBridge (14,842 LoC) and DB-Micro (22,933 LoC) in a publicly accessible artifact (Section of Data Availability). DBridge will also be released and maintained on Tai-e. We expect that DBridge and its resources will help advance static analysis for modern database-backed apps.

2 Motivating Example

We start with an example to illustrate how a Java-to-database value flow is established in a database-backed app (Section 2.1). Next, we introduce the key challenges of static analysis in resolving these value flows (Section 2.2), and present the design insights behind DBridge (Section 2.3).

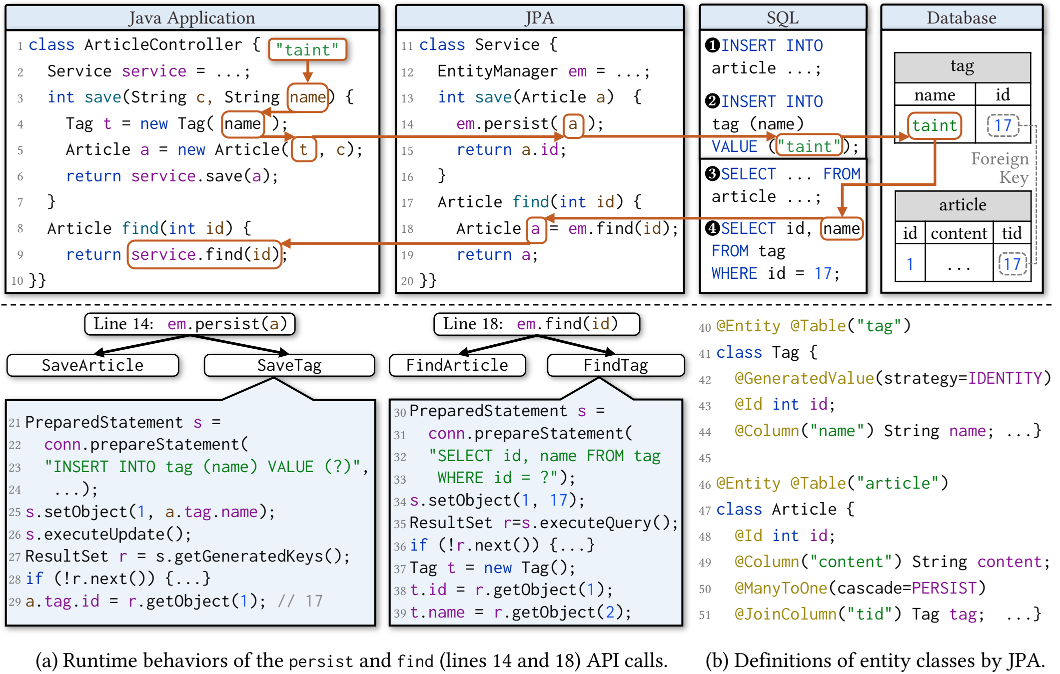
2.1 Java-to-Database Value Flows

2.1.1 Overview of Value Flows. Fig. 2 shows a simplified code example from a real-world BBS web app where it uses JPA (Java Persistence API) [3] to interact with databases, showing how a tainted value flow is established through Java-to-database interactions.

This app includes three components: the `ArticleController` class (line 1), which provides the `save` (line 3) and `find` (line 8) APIs for user access; the `Service` class (line 11), which manages database interactions via `JPA`; and the database itself, comprising two tables, `article` and `tag`. Each `tag` associates a category with an `article`, and the `tid` column in the `article` table maintains the `id` of the associated `tag`. When the `save` and `find` APIs are invoked in order, they enable the saving and retrieval of `Article` and the associated `Tag` objects. This process also triggers a value flow from the app to the database and back, as depicted by the arrows in the figure.

When a user calls the `save` API (line 3), the Spring framework extracts `name` and content `c` from the input. The app then creates a `Tag` and an `Article` object (lines 4-5), stores the input in these objects and associates these objects (i.e., `a.tag = t`, executed in line 5 but not shown). The app then calls the `save` method (line 6) in the `Service` class (line 13), which uses JPA `persist` to store the `Article` object in the database (line 14). This `persist` call also saves the associated `Tag` due to the annotation in line 50, a JPA feature that will be explained in Section 2.1.2. During this call, JPA generates two SQL `INSERT` statements to save these objects (❶ ❷ in Fig. 2). Specifically, statement ❷, `INSERT INTO tag (name) VALUE ("taint")`, saves the `name` field of the `Tag` object, which contains user-provided "taint" name, into the `name` column of a new row in the `tag` table.

When the user calls the `find` API (line 8) with the `id` of the `Article` object, the `id`—generated by the database and returned by the `save` API in line 6—is used to invoke the JPA `find` (line 18).



(a) Runtime behaviors of the `persist` and `find` (lines 14 and 18) API calls. (b) Definitions of entity classes by JPA.

Fig. 2. A simplified example of a real-world BBS app illustrating Java-to-database value flows.

This JPA `find` API call retrieves the `Article` object, along with the associated `Tag` object. During this call, JPA generates and executes two SQL `SELECT` statements (❸ ❹), and converts the query results into `Article` and `Tag` objects. Consequently, the "taint" in the `name` column of the first row of the `tag` table flows into the `name` field of the `Tag` object created by JPA. The tainted `Tag` object will be used alongside the `Article` object to render the webpage in the user's browser.

This example exposes a real Stored XSS vulnerability within the Java-to-database value flow: if an attacker injects a malicious script into the `name` parameter of the `save` API (line 3, replacing "taint" with a malicious script string), the script will be stored in the `tag` table. When an unsuspecting user later invokes the `find` API, the script may be retrieved and executed in the user's browser.

2.1.2 Understanding JPA Specification. To abstract and approximate Java-to-database value flows, it is essential to understand the semantics of high-level database access APIs provided by specifications like JPA and Spring Data JPA, which ultimately interact with the database through JDBC. Here, we use two key JPA APIs, `persist` (line 14) and `find` (line 18), to illustrate how these high-level APIs handle automatic conversion between entity objects (e.g., `Article` and its associated `Tag`) and corresponding database records (e.g., rows in the `article` and `tag` tables). Fig. 2(a) shows the semantics-equivalent JDBC statements of `persist` and `find` generated by the JPA framework based on the user-defined entity classes with JPA annotations (Fig. 2(b)). Because understanding (1) how these JDBC statements are generated and (2) their runtime behaviors is essential for static analysis to model these high-level APIs, we illustrate (1) and (2) below with examples.

(1) JDBC Generation. In line 14, `persist(a)` is meant to save the `Article` object (i.e., `SaveArticle` in Fig. 2(a), code not shown), but it also saves the `Tag` object (i.e., `SaveTag`). This is because, in the entity class `Article` (see Fig. 2(b)), the `tag` field has a JPA annotation `@ManyToOne(cascade=PERSIST)`, which specifies that multiple `Article` objects may share the same `Tag` and ensures that persisting

an `Article` also saves its related `Tag`. Similarly, the reason why the SQL-like JDBC `INSERT` statement in line 23 knows to insert into the `tag` table and target the `name` column is determined by annotations in the `Tag` entity class. Specifically, the `@Table("tag")` (line 40) maps the `Tag` entity to the `tag` table, and the `@Column` (line 44) maps the `name` field to the `name` column in this table.

(2) Runtime Behavior of JDBC. It is essential to capture the runtime behaviors of the four core steps in JDBC statement manipulation: *New Statement*, *Init Statement*, *Execute Statement*, and *Process Result*. We illustrate these steps using the JDBC code for saving `Tag` to the database (lines 21–29).

- *New Statement*: in lines 21–24, `PreparedStatement` is created to construct an SQL-like statement to be initialized and executed. Here, the `INSERT` statement (line 23) includes a parameter "?", allowing for substitution at runtime.
- *Init Statement*: in line 25, `setObject` initializes the parameter in the SQL-like statement. The first argument, 1, specifies that the first parameter of the `INSERT` statement is set to `a.tag.name`. In some JDBC statements, if the *New Statement* does not include an SQL-like statement, it is initialized separately by calling a similar method as in the *Init Statement*.
- *Execute Statement*: in line 26, `executeUpdate` sends the initialized `INSERT` (a real SQL statement) to the database to create a new row in the `Tag` table, insert the `name` field value into the `name` column, and generate an auto-incremented ID, say 17, as specified by the annotation in line 42.
- *Process Result*: JDBC uses `ResultSet` to store database results, which can be fetched in different ways, like using `getGeneratedKeys` in line 27 or querying directly with `executeQuery` in line 35. Lines 36–39 show how a database record is converted into an object (`ResultSet` iterates through each row, and `getObject` returns the value of a specified column in the row).

After reviewing the JDBC code for `persist`, understanding the `find` code (lines 30–39) is straightforward, so we will omit further explanation. However, one point needs clarification as it is also critical for static analysis when modeling these high-level APIs: how does the generated JDBC code identify the `Tag` object's ID as 17 (line 34)? This occurs because, to maintain the association between the `Article` and `Tag` objects, the `Tag`'s ID is stored as a foreign key in the `tid` column of the `article` table during the persistence of the `Article` object (indicated by `@JoinColumn("tid")` in line 51). Thus, when an `Article` is retrieved by its ID, say 1, by `em.find(1)`, JPA first looks up the `tid` value associated with the `Article` (17 in this case). This fetches the `Tag` object with ID 17 (line 38), along with its `name` ("`taint`") (line 39), finally resulting in a tainted value flow.

2.2 Challenges

We highlight two key challenges in pointer analysis for database-backed apps: the complexity and diversity of database access APIs, and the intricacies of cross-language analysis for Java and SQL.

Pointer Analysis for Complex and Diverse Database Access APIs. The complexity of analyzing Java database access APIs, such as JDBC, JPA, and Spring Data JPA, is immense. These specifications define hundreds of APIs, each exhibiting diverse behaviors influenced by factors like entity relationships, query types and persistence states. Unlike our simplified example with a many-to-one relationship (line 50), real-world scenarios involve a wide range of relationships—one-to-one, one-to-many, many-to-many, embedded, and inheritance—which can be unidirectional or bidirectional, with cascading or non-cascading behaviors. This variability significantly complicates the tracking of interactions between entities and their database representations. Moreover, while our example shows a simple entity-type query based on an entity's ID (line 18), real-world frameworks support various query types—such as JPQL, Native, Criteria, Named, Method-Name, and Method-Annotation queries—each requiring distinct handling for query translation and execution. The persistence state of entities (transient or persistent) further complicates database access, affecting how entities are saved, updated, or queried. Additionally, in JPA implementations like Hibernate, behaviors are not

only defined by annotations but also rely on external configuration files and metadata, dynamically applied through reflection. This further complicates the analysis, making it challenging to account for all the varying scenarios in a comprehensive manner.

Cross-language Pointer Analysis for Java and SQL. SQL is a language distinct from Java, with its own syntax and semantics. For constructing Java-to-database value flows, cross-language pointer analysis is essential. However, no existing pointer analysis or static analysis approach addresses this challenge. The difficulty largely stems from the complexity of SQL, which includes a broad array of operations that impact pointer analysis. These operations cover statements (e.g., INSERT, SELECT, UPDATE), clauses (e.g., WHERE, INNER/OUTER JOIN, FROM), predicates (e.g., IN, LIKE, NULL checks), expressions (e.g., logical/arithmetic/comparison/bit operations, column references, functions), and aliases (e.g., table and column aliases). Given their frequent use in real-world apps, a comprehensive analysis must model the effects of as many of these operations as possible. While our motivating example presents simple SQL statements, real-world scenarios often involve more complex operations. For instance, JPA can retrieve data from both the `article` and `tag` tables in a single query like: `SELECT a.id, a.tid, t.id FROM article AS a INNER JOIN tag AS t ON a.tid = t.id WHERE a.id = 1` (supplementary materials offer much more intricate cases).

Finally, it is worth mentioning that despite these two challenges, DBridge effectively handles the majority of the diverse and complex scenarios discussed above, as demonstrated in Section 6.1.

2.3 Design Insights of DBridge

The goal of DBridge is to design a unified pointer analysis that integrates the analysis of all components involved in Java-to-database interactions, including app code, database access APIs from various frameworks, SQL statements, and the underlying databases. Consolidating these analyses within a single pointer analysis offers two main advantages. First, the unified, on-the-fly pointer analysis can improve soundness by enhancing the resolution of one component using results from others. Second, it enables the direct application of selective context-sensitivity techniques [17, 34] inherent in pointer analysis to any component's code, without modifying the analysis rules. This allows us to effectively balance precision and efficiency. This is also the key reason why DBridge can easily achieve high efficiency in analyzing large, database-backed apps, as explained and demonstrated in Section 6.2. Given these benefits, for future work like handling other types of frameworks or hard-to-analyze programs, we recommend integrating the analysis of each component under a unified pointer analysis framework to achieve reinforced soundness and a tunable performance trade-off.

Analyzing database access APIs and SQL presents significant challenges due to their diversity and complexity, as discussed in Section 2.2. To address this, we propose to design pointer analysis rules based on two sets of primitives: framework primitives (F-primitives) and SQL primitives. This design, akin to a multiple intermediate representation (IR) in compilers, allows the analysis algorithms to focus on the core language representation while avoiding the diversity and intricacies of high-level abstractions. One might question why we use both F-primitives and SQL primitives, given that database access APIs ultimately translate to SQL for database interaction. Why not rely solely on SQL primitives? The answer lies in the fact that F-primitives capture more information than SQL primitives. For instance, F-primitives also represent the instructions for retrieving records from the database and mapping them to Java objects.

Drawing from extensive specifications of various database access frameworks and their real-world usage, we meticulously define four types of F-primitives, totaling 19 distinct F-primitives, and five types of SQL primitives, totaling 21 distinct SQL primitives (see Section 5 for details). To facilitate the analysis, we implement a framework statement transformer that converts JDBC, JPA,

and Spring Data JPA APIs into F-primitives (e.g., JDBC offers 50 APIs just for initializing JDBC statements, all of which are translated into a single F-primitive). A transformer then converts the F-primitives representing SQL execution into SQL primitives. We choose not to first translate JPA and Spring Data JPA to JDBC before converting to F-primitives, as such a static translation would be too complex and impractical.

Finally, we extend traditional Andersen-style pointer analysis for regular Java statements [33, 37] to incorporate new analysis rules for both F-primitives and SQL primitives. Since the entire analysis is built on the same theoretical framework, the effect of pointer analysis on an F-primitive can influence the analysis of regular Java statements or SQL primitives, and vice versa. This results in a cohesive, on-the-fly analysis.

3 DB-Micro

The interaction between Java apps and databases is inherently complex, involving diverse and intricate API calls from database frameworks, as well as SQL statements with complex constructs and semantics. To our knowledge, no benchmark systematically captures the key behaviors of these components, posing a challenge for effectively evaluating static analysis tools. To address this gap, we developed DB-Micro, a suite of micro-benchmarks designed to comprehensively evaluate the capability of static analysis tools in constructing Java-to-database value flows.

DB-Micro comprehensively encompasses the diverse behaviors involved in Java-to-database interactions, including all the interaction scenarios mentioned in the Challenges section (Section 2.2). This suite systematically incorporates key SQL operations, a broad range of query types, diverse entity relationships, and various entity persistence states, all of which are critical for constructing Java-to-database value flows in static analysis.

Specifically, DB-Micro consists of four benchmarks: one for SQL and three others tailored to JDBC, JPA, and Spring Data JPA (Fig. 5). The SQL benchmark covers a broad array of SQL operations relevant to pointer analysis. It includes key SQL statements (e.g., INSERT, SELECT, UPDATE), clauses (e.g., WHERE, INNER JOIN, OUTER JOIN, FROM), predicates (e.g., IN, LIKE, NULL checks), expressions (e.g., logical, arithmetic, comparison, and bitwise operations, column references, functions), and aliases (e.g., table and column aliases.) The JDBC, JPA, and Spring Data JPA benchmarks include eight query types, like Native Query, JPQL Query, Named Query, and Criteria Query, covering all query types except for StoredProcedure Query, which is less commonly used. Moreover, DB-Micro encompasses a comprehensive set of entity relationships, including one-to-one, one-to-many, many-to-one, and many-to-many relationships, as well as embedded objects and inheritance hierarchies. It also considers variations in these relationships, such as different associations (unidirectional vs. bidirectional), different fetch strategies (eager vs. lazy loading), and various cascade operations.

By integrating these multifaceted aspects, DB-Micro includes 824 carefully crafted test cases (10,172 lines of Java code), designed based on specifications, documentation, and inspections of high-quality real-world apps. Additionally, it provides corresponding ground truth data (12,761 lines of JSON code) for each test case, along with automated evaluation scripts. By converting the output of a static analysis tool into the required JSON format, these scripts can automatically generate an evaluation report. More details of DB-Micro are available in the supplementary material [20].

4 DBridge, Informally

In this section, we describe the working mechanism of DBridge, focusing on its analysis of Java-to-database interactions. Fig. 3 provides an overview of DBridge, which is composed of three main components: (1) an application analyzer that performs pointer analysis on the Java application to construct its value flows; (2) a database framework analyzer that handles database access APIs in

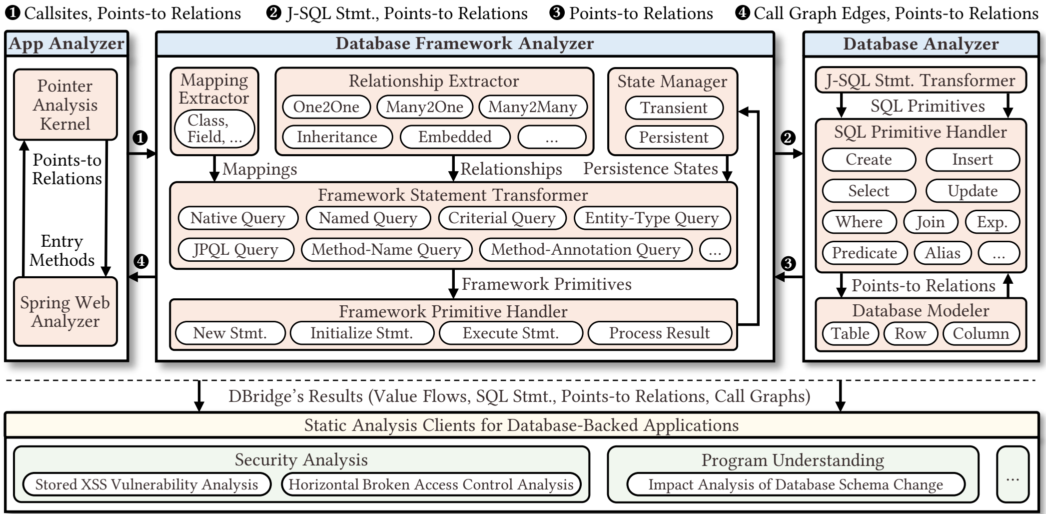


Fig. 3. The DBridge framework.

JDBC, JPA, and Spring Data JPA, bridging the value flows between the Java application and the database; and (3) a database analyzer that processes SQL statements to model the database and trace value flows within the database. Fig. 3 illustrates the interactions among these components, marked by ❶-❹ and discussed in detail later. The output of DBridge includes essential information such as Java-to-database value flows and SQL statements that might be executed at runtime, supporting various clients, including security analysis and program understanding.

Below, we introduce the three components of DBridge and explain their interactions.

Application (App) Analyzer. The app analyzer performs pointer analysis on the Java application to compute points-to relations and construct its call graph. During this process, when the app analyzer encounters a database access API callsite, it delegates the callsite, along with the points-to relations of the API call arguments, to the database framework analyzer to initiate its analysis (❶).

We observed that database-backed applications often use the Spring framework, which introduces complex mechanisms implemented with challenging Java features (e.g., reliance on annotations and native code) that pose significant challenges for pointer analysis. To enhance the effectiveness of DBridge in analyzing such applications, we incorporate a *Spring Web Analyzer* in the app analyzer. This component models the intricate mechanisms provided by the Spring framework, including customized entry points and dependency injection, following state-of-the-art approaches [7].

Database Framework Analyzer. We introduce a database framework analyzer to handle complex database access APIs in JDBC, JPA, and Spring Data JPA. To interact with a database, these frameworks generate and initialize JDBC SQL (J-SQL for short—an extended SQL containing parameters that reference Java values) statements, send them to the database, and process the result. The analyzer models these behaviors.

Triggered by API calls and their arguments from the app analyzer (❶), the database framework analyzer models the API semantics to generate corresponding J-SQL statements and forwards them, along with the points-to relations of the parameters in the J-SQL statements, to the database analyzer (❷). The database analyzer processes these J-SQL statements (discussed in detail later) and returns the results (❸). Based on these results and the API semantics, the database framework analyzer analyzes the side effects on pointer analysis and reports them to the app analyzer (❹).

To address the complexities of various database access APIs across different database frameworks, as discussed in Section 2.3, we designed 19 framework primitives that abstract the diverse behaviors involved in the four core steps of database access. Due to space constraints, we only detail the handling of key F-primitives in Section 5.2. The database framework analyzer operates based on F-primitives, transforming database access API calls into corresponding F-primitives and then analyzing them to model their behavior.

As shown in Fig. 3, the database framework analyzer employs the *mapping extractor*, *relationship extractor*, and *state manager* to collect database-related information needed for analyzing database access API calls, such as mappings between Java objects and database contents and relationships between entities. This information is primarily extracted from the application through APIs or annotations provided by database frameworks. The *framework statement transformer* uses this information to convert queries into F-primitives. Each query consists of a series of operations that complete a database access. For example, `SaveTag` and `FindTag` in Fig. 2(a) represent two distinct queries. The analyzer supports a wide range of queries across various frameworks. The *framework primitive handler* processes these F-primitives, generating J-SQL statements and passing them to the database analyzer with the points-to relations of the parameters. It also links value flows from the database to the Java application (e.g., transforming the query results from the database into entity objects), while updating the persistence states of entity objects in the *state manager*.

Database Analyzer. In the database analyzer, we design a database model to simulate the execution of J-SQL statements. The analyzer consists of three main components: a *J-SQL statement transformer*, an *SQL primitive handler*, and a *database modeler*. The *J-SQL statement transformer* converts each J-SQL statement from the database framework analyzer (❷) into a series of SQL primitives designed to address the complexities of various SQL operations, as discussed in Section 2.3. The *database modeler* statically represents database tables, rows, columns, and their contents as a collection of pointers and objects that interact with pointer analysis. The *SQL primitive handler* works with the database modeler to perform pointer analysis on these primitives, establishing points-to relations between SQL variables (e.g., variables referencing tables, rows, or columns) and database objects (e.g., table objects, row objects, or constants), and sends the results back to the database framework analyzer (❸). The database model and the handling of key SQL primitives are detailed in Sections 5.1 and 5.3, respectively.

5 DBridge, in Detail

In this section, we delve into the core working mechanism of DBridge, which, as described in Section 4, involves the app analyzer, database framework analyzer, and database analyzer. Because the app analyzer employs standard Java pointer analysis [29, 30, 33, 37] and integrates state-of-the-art techniques [7] for handling the Spring framework, we focus on the handling of core F-primitives and SQL-primitives for the database framework analyzer and database analyzer, respectively.

In Section 5.1, we start by offering an overview of the core primitives' working mechanisms. This includes an introduction to the database framework model and the database model, both of which are crucial for managing these primitives. With this foundation, we can more easily understand the handling of F-primitives and SQL primitives in Sections 5.2 and 5.3, respectively. Finally, in Section 5.4, we further understand how DBridge constructs specific Java-to-database value flows by working through a concrete example that illustrates the processing of relevant primitives.

Please note that we have also formalized all the core primitives and the pointer analysis rules for managing them in the supplemental material [20]. Additionally, we have released the full implementation of DBridge as open-source [19], ensuring that all its details are publicly accessible.

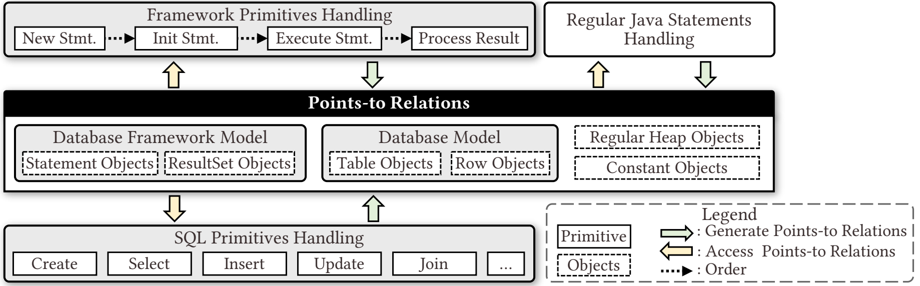


Fig. 4. Overview of pointer analysis for Framework primitives, SQL primitives, and regular Java statements.

5.1 Overview of Core Primitive Handling in DBridge

As explained in Section 2.3, DBridge builds upon traditional pointer analysis by incorporating new rules specifically designed to handle F-primitives and SQL primitives. As shown in Fig. 4, the essence of DBridge's pointer analysis lies in *generating* new points-to relations when processing various primitives (these relations include the relations between variables and objects, as well as those between objects, like $o_1.f$ pointing to o_2). This process typically involves *accessing* the points-to relations of pointers within primitives and generating points-to relations based on primitives' semantics. Once generated, these points-to relations are updated in the unified heap model.

Beyond the traditional heap model (comprising regular heap objects created at allocation sites and various constant objects), to construct Java-to-database value flows, we have developed a new database framework model and database model (see the middle part of Fig. 4). These models include essential abstracted objects pertinent to handling database frameworks (i.e., Statement objects and ResultSet objects) as well as objects that represent the database itself (i.e., Table objects and Row objects). We will introduce these two models in the subsequent part of this section and explain how DBridge handles F-primitives and SQL primitives in Sections 5.2 and 5.3, respectively.

Database Framework Model. We present a database framework model that enables us to simulate the behavior of database framework APIs. Most Java database frameworks are built upon JDBC, a Java interface that allows applications to execute SQL statements on a database. These frameworks offer APIs to manipulate JDBC statements, which is why the database framework analyzer is centered around JDBC. Hence, our database framework model is also designed around JDBC.

In JDBC, a JDBC statement encapsulates a J-SQL statement and a series of parameters for that statement. In our model, each JDBC statement is represented as a JDBC *statement object*. This object includes a field, *sql*, which points to the encapsulated J-SQL statement, as well as a series of fields that point to the resolved parameters for the statement. Additionally, the execution result of a JDBC statement is essentially a two-dimensional array, which we model as a *ResultSet object*.

Database Model. We present a database model designed to leverage pointer analysis for simulating database operations. In this model, each database table is represented by a *table object*, with the table name serving as its unique identifier. We associate this table object with a list of column names corresponding to the table's structure.

A database table can contain multiple rows, which we model as *row objects*. Each table object has a *rows* field that points to the row objects inserted into the table. Each row may contain several columns, each holding a specific value. In our model, these columns are represented as fields of the row objects, with the column names serving as the field names. The relationship between a column and its corresponding value is captured through points-to relations, where the field of a row points to the value it holds. Thus, row objects and their fields effectively model the structure of the table's rows and columns, while the values within the rows are represented as Java objects and constants.

5.2 Database Framework Analyzer

The database framework analyzer transforms each database access API call into a series of framework primitives (F-primitives) and analyzes them. Given the complexity and diversity of this transformation, involving multiple database frameworks and hundreds of APIs, we omit the full details here for brevity. Instead, we offer a brief introduction to our F-primitives and their notations, followed by an explanation of several representative primitives and how they are handled in detail.

F-Primitive and Notation. To model how frameworks manipulate JDBC statements, we define 19 framework primitives (F-primitives), which are categorized into four types: (1) *New Statement*, which creates a JDBC statement object and encapsulates a J-SQL statement within it; (2) *Initialize Statement*, which resolves the parameters for a JDBC statement object and stores them in its fields; (3) *Execute Statement*, which carries out the execution of an initialized JDBC statement; and (4) Process Result, which transforms the result of the Execute Statement (such as ResultSet objects) into Java elements, typically represented as a Java entity or an integer (e.g., the ID of a Java entity).

The notation for an F-primitive is $\text{PrimitiveName}\langle element_1^I, element_2^O, element_3^{IO}\rangle$, where each *element* signifies an input or output component of the primitive. These components can be pointers or string constants, either generated by DBridge when creating the primitive or originating from the Java application. Superscripts indicate *element*'s roles: I for input, O for output, and IO for both input and output. Input elements, whether the object pointed to by the pointers or string constants, are used in primitive handling, while output elements (always pointers) receive new points-to relations generated during primitive handling.

Next, we introduce one representative primitive for each type of F-primitive, detailing each in two parts: the description, which specifies its meaning along with its inputs and outputs, and the handling, which explains how DBridge processes it.

New Statement. Representative: $NewStmtByEntityType\langle entityType^I, sqlOp^I, stmt^O\rangle$.

- Description: This primitive describes the creation of a JDBC statement, which is subsequently used to query or save an entity object. The input $entityType^I$ is a string constant that represents the type of the manipulated entity object (e.g., `"Article.class"`), and the input $sqlOp^I$ is also a string constant, which represents the specified SQL operation (e.g., `"INSERT"`). The newly created JDBC statement will be stored in $stmt^O$ as output.
- Handling: To handle this primitive, DBridge starts by constructing a J-SQL statement, which is a string constant formulated based on the entity type $entityType^I$ and the SQL operation $sqlOp^I$. Next, it creates a JDBC statement object, setting its *sql* field to point to the constructed J-SQL statement. Finally, this object is added to the points-to set of the pointer $stmt^O$.

Notably, the created JDBC statement object will also be manipulated or utilized by other primitives: its parameter fields will be initialized through the handling of *Init Statement* primitives, and its execution will be modeled through the handling of the *Execute Statement* primitives.

Init Statement. Representative: $InitStmtByEntity\langle stmt^{IO}, entity^I\rangle$.

- This primitive describes the initialization of a JDBC statement, setting its parameters to the values from the fields of a specified entity. The input $stmt^{IO}$ is a pointer that points to the JDBC statement to be initialized, and the input $entity^I$ is a pointer to the entity that will be persisted in the database (e.g., an `Article` object). The values of the entity's fields will be assigned to the parameters of the JDBC statement as output.
- Handling: DBridge initializes a JDBC statement object pointed to by the pointer $stmt^{IO}$ using the fields derived from the entity object pointed to by $entity^I$. Specifically, the points-to set of the fields of the entity object is propagated to the corresponding parameter fields of the

JDBC statement object. By processing this type of primitives, DBridge constructs the value flows from Java app to database framework (represented by JDBC statement objects).

Execute Statement. Representative: $ExecuteStmt\langle stmt^I, resultSet^O\rangle$.

- Description: This primitive describes the execution of a JDBC statement. The input $stmt^I$ is a pointer that points to the JDBC statement to be executed. A newly created ResultSet, which represents the execution result, will be stored in $resultSet^O$ as output.
- Handling: For the JDBC statement object pointed to by $stmt^I$, DBridge transforms the contained J-SQL statement and its parameters into a series of SQL primitives and processes them (see Section 5.3 for details). Subsequently, DBridge converts the simulated execution result of the JDBC statement—such as data retrieved from the database—into a ResetSet object, which is then stored in the points-to set of pointer $resultSet^O$. Through the handling of such primitives, DBridge builds the value flow between database framework and database.

Process Result. Representative: $ResultToEntity\langle resultSet^I, entityType^I, entity^O\rangle$.

- Description: This primitive describes the conversion of a JDBC statement execution result into an entity object. The input $resultSet^I$ is a pointer that points to a ResultSet object, which represents the results of the statement execution. Additionally, the input $entityType^I$ is a string constant that specifies the type of the entity object into which the ResultSet object will be converted. The newly created entity will be stored in $entity^O$ as output.
- Handling: DBridge starts by creating an entity object of the specified type $entityType^I$. Then, it extracts the query results from the ResultSet object pointed to by $resultSet^I$ and stores them in the appropriate fields of the entity object. Finally, this entity object is added to the points-to set of the pointer $entity^O$. By handling this type of primitives, DBridge establishes the value flow from the database framework (represented by ResultSet objects) back to the Java application (represented by Java entity objects).

5.3 Database Analyzer

The database analyzer transforms each J-SQL statement (obtained from the database framework analyzer) into a series of SQL primitives and analyzes them. The transformation process involves parsing the J-SQL statement to generate its AST and performing semantic analysis to extract SQL primitives. Given the tedious and complex nature of this process, we will omit these intricate details and instead focus on the SQL primitives themselves and how they are handled.

SQL Primitive. In DBridge, we define 21 SQL primitives to represent various database operations initiated by SQL statements (such as create, select, etc.). The notation for SQL primitives mirrors that of F-primitives, as outlined in Section 5.2. Each primitive is similarly composed of two parts: the description and the handling. Below, we introduce the most essential and commonly used SQL primitives along with their handling. For clarity, we have simplified the structure of these primitives, with a more detailed version available in the supplemental material.

$create\langle table^O, tableName^I, columnName^I_1, \ldots, columnName^I_n\rangle$.

- Description: This primitive describes the creation of a database table. The input consists of a series of string constants, where $tableName^I$ represents the name of the table to be created, and $columnName^I_1, \ldots, columnName^I_n$ represents the column names of the table. A newly created table will be stored in $table^O$ as output.
- Handling: DBridge generates a table object, identified by $tableName^I$, and adds it to the points-to set of $table^O$. Additionally, this generated table object is associated with the column names $columnName^I_1, \ldots, columnName^I_n$.

$select\langle table^I_{source}, columnName^I_1, \ldots, columnName^I_n, table^O_{target}\rangle$.

- Description: This primitive describes a select operation that extracts specific columns from a source table to construct a new table. The input $table^I_{\text{source}}$ is the source table, and the inputs $columnName^I_1, \ldots, columnName^I_n$ are string constants representing the names of the columns to be selected. The selected columns are used to create a new table, which is stored in $table^O_{\text{target}}$ as output.
- Handling: DBridge creates a new table object and adds it in the points-to set of $table^O_{target}$. To initialize this table object, DBridge creates a new row object for each row object found in the table object pointed to by $table^I_{source}$. Besides, the points-to sets of the selected columns $columnName^I_1, \ldots, columnName^I_n$ (modeled as fields, as described in Section 5.1) from the source row objects are propagated to the corresponding columns in the new row objects.

$insert\langle table^{IO}, columnName^I_1, \ldots, columnName^I_n, exp^I_1, \ldots, exp^I_n, param^I_1, \ldots, param^I_m\rangle$.

- Description: This primitive describes the insertion of a new row into a database table. The input of this primitive consists of four parts: (1) $table^{IO}$ is the table to be inserted into; (2) the string constants $columnName^I_1, \ldots, columnName^I_n$, which represent the columns of the table to be inserted into; (3) the string constants $exp^I_1, \ldots, exp^I_n$, which represent SQL expressions used to compute the values to be inserted into the table; and (4) $param^I_1, \ldots, param^I_m$ are J-SQL parameters, which are used in SQL expressions evaluation. The evaluation results of these SQL expressions are used to create a new row object, which will be stored in the table corresponding to $table^{IO}$ as output.
- Handling: DBridge starts by creating a new row object and stores it into the table object pointed to by $table^{IO}$. It then initializes the row object. Specifically, DBridge propagates the objects obtained from evaluating the expressions $exp^I_1, \ldots, exp^I_n$ into the columns named $columnName^I_1, \ldots, columnName^I_n$ of the new row object.

$join\langle table^I_{left}, table^I_{right}, table^O_{result}\rangle$.

- Description: This primitive describes a join operation that combines two input tables into a temporary result table. The inputs $table^I_{left}$ and $table^I_{right}$ represent the tables to be joined. The resulting table is then stored in $table^O_{result}$ as output.
- Handling: When processing this primitive, DBridge first creates a new table object for the pointer $table^O_{result}$. It then computes the Cartesian product of all row objects in $table^I_{left}$ and $table^I_{right}$. For each resulting pair from this product, DBridge constructs a new row object that combines all columns from the two row objects. Finally, this newly created row object is added to the points-to set of the table object's *rows* field.

Note that the analysis conducted by DBridge is sound with respect to the primitives it handles, as we ensure an over-approximation of their semantics. However, DBridge is not sound for the entire range of Java-to-database interactions due to the inherent complexity. Specifically, for SQL, it struggles to model the semantics of SQL built-in functions; for database frameworks, DBridge cannot fully handle Criteria Query due to the numerous API calls needed for even a single criterion; for Java app, DBridge does not account for SQL statements dynamically generated through string concatenation. These limitations are further explained and demonstrated in Sections 6.1 and 6.2.2.

5.4 Example

In this section, we further illustrate how DBridge constructs specific Java-to-database value flows by working through a concrete example. We begin by presenting the F-primitives and SQL primitives

that DBridge generates for a given database access API call. Then, we illustrate how DBridge processes these primitives to construct Java-to-database value flows.

We develop this section around the call `em.persist(a)` (line 14 in Fig. 2), which triggers the saving of an `Article` object along with its associated `Tag` objects. For clarity and brevity, we focus solely on the saving of the `Tag` object. Nonetheless, a thorough understanding of the previous sections is necessary to grasp the following material; thus, we invite readers to revisit them first.

Primitive Generation. To abstract the process of persisting a `Tag` object via a JDBC statement, DBridge generates the following four primitives.

(1) $NewStmtByEntityType\langle \texttt{"Tag.class"}^I, \texttt{"INSERT"}^I, stmt^O \rangle$.This F-primitive describes the creation of the JDBC statement, which wraps the `INSERT` statement for saving the `Tag` object.
(2) $InitStmtByEntity\langle stmt^{IO}, entity^I \rangle$. This F-primitive describes the process of converting the `Tag` object held by $entity^I$ into parameters of the JDBC statement.
(3) $ExecuteStmt\langle stmt^I, resultSet^O \rangle$. This F-primitive describes the execution of the JDBC statement, transforming the `INSERT` statement and parameters of the JDBC statement to the following $insert$ primitive. The execution result of the JDBC statement is then assigned to $resultSet^O$.
(4) $insert\langle table^{IO}, \texttt{"name"}^I, \texttt{"?1"}^I, param^I \rangle$. This SQL primitive describes the execution of the `INSERT` statement, which ultimately inserts the `Tag` object into the `tag` table held by $table^{IO}$.

These F-primitives are interconnected through a shared primitive variable $stmt$. Specifically, the variable $stmt$ corresponds to the JDBC statement responsible for saving the `Tag` object, coordinating the entire process. It is created in $NewStmtByEntityType$, its parameter fields are initialized in $InitStmtByEntity$, and its execution is modeled in $ExecuteStmt$ primitives.

Primitive Handling. DBridge processes these four primitives step by step to construct the Java-to-database value flow, in conjunction with their handling presented in Sections 5.2 and 5.3.

(1) Handling $NewStmtByEntityType\langle \texttt{"Tag.class"}^I, \texttt{"INSERT"}^I, stmt^O \rangle$: DBridge creates a JDBC statement object for the variable $stmt^O$, and assigns the J-SQL statement `INSERT INTO tag (name) VALUES (?)` to its sql field. This J-SQL is generated based on the mapping between the `Tag` class and the `tag` table.
(2) Handling $InitStmtByEntity\langle stmt^{IO}, entity^I \rangle$: DBridge extracts the `name` field of the `Tag` object pointed to by $entity^I$ and propagates its points-to set to the first parameter of the JDBC statement object pointed to by $stmt^{IO}$.
(3) Handling $ExecuteStmt\langle stmt^I, resultSet^O \rangle$: Based on the J-SQL encapsulated in the JDBC statement object pointed to by $stmt^I$, DBridge generates the following SQL primitive: $insert\langle table^{IO}, \texttt{"name"}^I, \texttt{"?1"}^I, param^I \rangle$. Here, the variable $table^{IO}$ points to the `tag` table object, while the string $\texttt{"name"}^I$ denotes the column to be inserted. The parameter expression $\texttt{"?1"}^I$ indicates that the $\texttt{"name"}^I$ column receives data from the first parameter, i.e., the variable $param^I$. It's important to note that when constructing this $insert$ primitive, DBridge propagates the points-to set of the first parameter carried by the JDBC statement object, which originates from the `name` field of the `Tag` object, to $param^I$. Additionally, the execution result of this JDBC statement is stored in $resultSet^O$ in the form of a ResultSet object.
(4) Handling $insert\langle table^{IO}, \texttt{"name"}^I, \texttt{"?1"}^I, param^I \rangle$: DBridge first creates a row object and inserts it into the `tag` table object pointed to by $table^{IO}$. Subsequently, it initializes the row object by propagating the points-to set of $param^I$—which originates from the `name` field of the `tag` object—to the row object's `name` column.

Through the handling of these four primitives, DBridge constructs a Java-to-database value flow, where the `name` from the `Tag` object flows through the first parameter of the JDBC statement, into the SQL statement, and ultimately into the `tag` table.

6 Evaluation

We evaluate DBridge's effectiveness by conducting the following research questions.

RQ1. How does DBridge perform on DB-Micro, the micro-benchmark suite for evaluating static analysis of Java-database interactions?

RQ2. How does DBridge perform on real-world database-backed apps in terms of soundness, precision, and efficiency?

RQ3. How does DBridge compare to the state-of-the-art in recognizing SQL statements from database access API calls, a fundamental task in static analysis for database-backed apps?

To further assess DBridge's practical value as a foundational analysis tool providing comprehensive value flow information for database-backed apps, we evaluate its support for client analyses:

RQ4. Can DBridge support its security clients—Stored XSS analysis and horizontal broken access control analysis—in detecting real vulnerabilities in real-world database-backed apps?

RQ5. Can DBridge assist its program understanding client, an impact analysis for database schema changes, in identifying affected program sites—a critical task in database management?

We implement DBridge on top of Tai-e [40], a state-of-the-art static analysis framework for Java that offers powerful and highly extensible pointer analysis, which significantly facilitate the development of our analysis. All experiments are conducted on a machine with an Intel i7 2.6GHz CPU and 16GB of memory. Note that, while constructing Java-to-database value flows (as defined in Section 1) is crucial, no prior work can achieve this. So there is no comparable work in RQ1 and RQ2 where such value flows are required.

6.1 RQ1. The Effectiveness of DBridge on DB-Micro

As mentioned in Section 3, we constructed DB-Micro, which comprises 824 test cases, to comprehensively evaluate the static analyzer's ability to handle the complexity and diversity of Java-database interactions. Each test case in DB-Micro is accompanied by a ground truth that represents its runtime Java-to-database value flows. We evaluate DBridge by comparing the Java-to-database value flows it captures against this ground truth, considering a test case successful only if the static analysis result precisely matches the runtime value flows.

SQL Benchmark – Select (case type and number)			
constant_value	4	subquery	1
column_value	3	group_having	4
function_value	2	alias	2
unary_value	2	where_logical	3
nested_value	1	where_in	1
math_value	4	where_like	2
compare_value	4	where_join	5
TOTAL		38	

Fig. 5. DB-Micro benchmark and DBridge's overall results.

DBridge passed 721 out of 824 test cases across the four micro-benchmarks, a very promising result given the complexity of SQL, database frameworks, and the strict evaluation criteria. However, DBridge does not fully support certain features due to their inherent complexity. For instance, some SQL cases combining SELECT, WHERE, and LIKE clauses involve complex regular expressions, and some UPDATE cases require evaluating intricate arithmetic functions. In JPA, Criteria Queries present challenges due to the numerous API calls needed for even a single criterion. DBridge can handle *all* types of framework statements except for Criteria Query in JPA and Spring Data JPA. Further details on the passed and failed test cases are available in the supplementary material. We plan to extend DBridge's capabilities to address these limitations in future work.

6.2 RQ2. Effectiveness of DBridge in Building Real-World Java-to-Database Value Flows

6.2.1 ***Experimental Setups.*** We selected 10 popular Java web apps from GitHub as real-world benchmarks. The selection criteria are based on popularity, indicated by high stars on GitHub or frequent citations in related research. For the former, the apps have around 1,000 stars or more such as eladmin (21.5k), favorites-web (4.8k), SpringBlog (1.6k), etc. For the latter, the apps are frequently referenced in related works such as [8–10, 21]. These codebases (excluding libraries) range from 2K to 129K lines of code and cover various business domains, such as blogs, e-commerce, and CMS. Their popularity and diversity make them exemplary for evaluating DBridge in the real world.

To evaluate DBridge's capability in constructing Java-to-database value flows through static analysis, we need to collect executed value flows covering a broad range of Java-to-database interactions as metrics for measuring soundness and precision. However, the original test cases provided by the app developers were limited, capturing only a fraction of interactions. To address this, we expanded the test cases to cover a wider range of Java-to-database interactions.

Collecting all dynamic value flows triggered by the test cases (starting and ending at all possible variable combinations) for comparison with static analysis is impractical. Instead, we focus on a set of meaningful dynamic flows: critical parameters of various web request APIs, denoted as *Req.*, which can inject commands or receive sensitive inputs, and response variables of various response APIs, denoted as *Resp.*, which can receive data from the database. The dynamic flows of interest start from these Req. and end at these Resp., traversing realistic app code, database framework API calls, and SQL executions. If a static analysis can detect these dynamic flows—i.e., determine that the values from a Req. reaches a Resp.—it demonstrates the ability of the static analysis to abstract and over-approximates these complex Java-to-database interactions.

6.2.2 ***Understanding the Results.*** Table 1 presents DBridge's results in constructing Java-to-database value flows, using the real-world apps and setups described earlier. In the table, #Req., #Resp., and #Dyn., represent the number of request parameters, response variables, and dynamic flows, respectively. #Static represents the number of flows identified by static analysis, while #Match shows the number of static flows that coincide with dynamic flows.

Recall experiments measure the soundness of static analysis [8, 18, 25], showing how many true dynamic flows are detected. A higher recall rate (#Match/#Dyn.) indicates better soundness. A web app often has multiple entry points that can be triggered by different test cases at runtime. In traditional recall experiments, static analysis analyzes an app from all identified entry points, independent of specific test cases. If a static analysis reports a value flow not found in the dynamic flows, it cannot be considered false, as it may still be valid under other test cases not yet provided.

To assess DBridge's precision, we conducted a precision experiment, shown on the right side of Table 1. In this approach, static analysis is performed for each test case individually, starting only from the entry points triggered by that test case. For instance, with 30 test cases, static analysis is applied 30 times. Each analysis is limited to the runtime behavior of the specific test case, allowing a value flow to be marked as false if it is not executed with the current input (test case). This actually provides a stricter measure of precision. Note that in the precision experiment, #Dyn. is higher than in the recall experiment because it aggregates counts across each test case run. Different test cases may trigger the same req.-resp. pair, which is counted multiple times in the precision experiment. Similarly, #Static and #Match in the precision experiment are computed for each static analysis run and then summed (as shown in the table). With this context, we now detail DBridge's soundness, precision, and efficiency.

Soundness (Recall). As discussed in Section 2.2, the complexity and interdependence of Java-to-database interactions make static analysis challenging. DBridge demonstrates strong soundness,

Table 1. DBridge's results in building Java-to-database value flows for real-world applications.

Application	Time (s)	#Req.	#Resp.	Recall				Precision			
				#Dyn.	#Static	#Match	Recall	#Dyn.	#Static	#Match	Prec.
eladmin	326	24	48	93	135	72	77%	175	254	120	47%
mblog	153	21	22	59	60	53	90%	93	147	87	59%
SpringBlog	183	12	19	105	123	105	100%	115	127	115	91%
JavaQuarkBBS	225	24	42	96	96	93	97%	142	202	139	69%
petclinic-JPA	72	15	17	44	60	44	100%	56	68	56	82%
petclinic	86	15	17	44	60	44	100%	56	68	56	82%
favorites-web	210	24	33	27	30	18	67%	27	30	21	70%
OnlineMall	230	24	41	107	123	105	98%	116	126	114	90%
wallride	273	21	47	95	111	95	100%	108	132	108	82%
bbs-pro	395	24	41	70	57	50	71%	81	84	61	73%
AVERAGE	215	20	33	74	86	68	90%	97	124	88	75%

with an average recall rate of 90%. This success is due to DBridge's comprehensive and robust analysis of Java-to-database interactions, including Spring IoC in app code, various complex database access APIs in JPA, Spring Data JPA, and JDBC, as well as SQL syntax and semantics incorporated with database abstraction. However, DBridge still struggles with certain interactions. For example, in the *bbs-pro* app, the recall rate drops due to the use of string concatenation to construct SQL queries—a practice that is relatively uncommon in modern apps and not well supported by DBridge. Similarly, in the *eladmin* and *favorites-web* apps, the recall rate is lower because some query conditions are derived from complex calculations, such as SQL functions and file parsing, which DBridge's current implementation cannot handle.

Precision. DBridge achieves good precision, with an average rate of 75%, which is relatively high for static analysis of complex apps. This success is mainly due to DBridge's careful analysis of each key phase in Java-to-database interactions. First, for app code, DBridge employs advanced pointer analysis with selective context-sensitivity [17, 34, 41], applying deep contexts to the app code and precisely modeling the Spring framework features. Second, for database frameworks, DBridge creates framework primitives by using pointer analysis to precisely capture the relationships between entity objects and their persistence states, while accurately resolving the relevant parameters for database access API calls. Third, for SQL and database, DBridge accurately models the database structure and uses this foundation to model SQL executions as precisely as possible. These strategies together contribute to DBridge's high precision. A flaw in any of these areas may significantly reduce precision. For example, treating database tables as collections of columns without distinguishing rows drops the average precision to 29%.

Despite achieving good average precision, some apps experience noticeable precision losses due to certain factors. For example, while selective context-sensitive pointer analysis is used for app code, certain JDK code that can affect Java-to-database value flows is still analyzed context-insensitively to balance efficiency. This can cause value flows in these library classes to merge. Additionally, the complexity of database frameworks presents challenges in achieving high precision. While we make efforts to model core database mechanisms accurately, some features remain difficult to handle precisely. For instance, Spring Data JPA's Criteria Query, used in apps like *eladmin* and *wallride*, dynamically constructs SQL statements. To maintain soundness, DBridge does not analyze the WHERE clause of the Criteria Query APIs. Attempting to do so would force DBridge to include all possible query conditions in the SELECT statement, leading to the retrieval of fewer rows than expected at runtime because some query conditions might be omitted due to their conditional logic. While this approach is more sound, it reduces precision due to over-approximation.

Efficiency. DBridge exhibits rapid analysis speeds, completing each app analysis in under 7 minutes, including the largest, *bbs-pro*, which contains 129k lines of app code and depends on 152 libraries. This efficiency is mainly due to DBridge's differentiated analysis strategy: For developer-written app code, critical to analysis precision, DBridge employs deep context-sensitive pointer analysis. This portion of the code represents a small fraction of the total codebase, and the number of call paths for each app method is limited. Therefore, performing deep context-sensitive pointer analysis on this code does not result in an excessively large call graph and pointer flow graph. For most library code, DBridge applies fast context-insensitive pointer analysis, except for precision-critical JDK methods, such as those related to collections [7]. Additionally, DBridge deeply models complex database framework API calls, including the further calls these APIs make to interact with databases, effectively bypassing large portions of database-related code. This comprehensive modeling significantly reduces the analysis workload and further accelerates the overall process.

6.3 RQ3. Effectiveness of DBridge in Identifying SQL Statements

Statically identifying the SQL statements executed at runtime for a specific call site of a database framework API (e.g., `find(id)` in line 18 of Fig. 2) is a fundamental and critical task for optimizing and maintaining database-backed apps [15, 16, 21, 28, 32, 45]. Most research in static analysis focuses on this challenge, but to our knowledge, no existing approach captures Java-to-database value flows like DBridge does. Current methods are limited in recognizing SQL statements due to the diversity and complexity of database frameworks, as well as overlooking Java-to-database value flows, limitations that DBridge overcomes. Below, we compare DBridge with SLocator [21], the state-of-the-art (most advanced) open-source approach for SQL identification.

Table 2. Evaluation results of DBridge's and SLocator's capabilities in identifying SQL statements.

Application	#Dyn. SQL	**DBridge**		**SLocator**	
		#identified (#matched)	Cov.	#identified (#matched)	Cov.
eladmin	50	83 (37)	74%	7 (1)	2%
mblog	36	60 (26)	72%	15 (6)	17%
SpringBlog	20	22 (19)	95%	1 (1)	5%
JavaQuarkBBS	34	42 (22)	65%	4 (1)	3%
petclinic-JPA	10	16 (10)	100%	17 (6)	60%
petclinic	10	14 (10)	100%	3 (2)	20%
favorites-web	29	50 (22)	76%	26 (8)	28%
OnlineMall	25	31 (25)	100%	2 (1)	4%
wallride	51	83 (21)	41%	23 (3)	6%
bbs-pro	68	402 (55)	81%	0 (0)	0%
AVERAGE	33	80 (25)	80%	10 (3)	15%

Table 2 shows the results, showing that DBridge significantly outperforms SLocator in identifying SQL statements, with an average coverage rate (denoted as Cov.) of 80% compared to SLocator's 15%. This performance gap is due to DBridge's comprehensive modeling of three key factors influencing SQL generation (see Section 4): API types, relationships between entity objects, and entity states. In contrast, SLocator struggles with these factors. First, modern database frameworks support a variety of query types. While SLocator handles more types than other approaches, it still lacks support for several commonly used ones. For example, it only supports annotation-based queries for Spring Data JPA, leaving other query types unaddressed. Second, DBridge models the relationships between entity objects, which are crucial for identifying cascade SQL statements. SLocator, however, provides limited support for how these relationships impact cascade operations. Finally, SLocator does not model entity states, which prevents it from recognizing `UPDATE` statements in JPA and Spring Data JPA APIs.

Regarding precision, we calculated it using the same approach as the second recall experiment in Table 1. DBridge's soundness, reflected in the coverage rate (recall), is much higher than SLocator's (80% vs. 15%), making direct precision comparisons less meaningful. For example, in *bbs-pro* (Table 2), there are 68 actual SQL statements, and SLocator identified 0 and 0 matched, but the precision is 100%. In this context, DBridge has an average precision of 54%, and SLocator has 85%.

While comparing precision between DBridge and SLocator is less meaningful, we can explain why DBridge achieves lower recall and precision for SQL identification compared to value flow construction in RQ2. This is mainly due to strict string-matching criteria. Semantically equivalent SQL statements often fail to match because of syntax differences. For example, a single SELECT query with joins (e.g., `select * from a inner join b on a.id = b.id where a.id = 1`) might be split into sub-queries (e.g., `select * from a where id = 1` and `select * from b where id = 1`). DBridge identifies the latter, while the actual query at runtime is the former. These syntax mismatches reduce recall and precision in SQL identification but do not affect value flow construction, as the semantics remain equivalent.

6.4 RQ4. Can DBridge Assist in Security Analysis for Database-backed Apps?

To examine the practical utility of DBridge as a foundational tool for constructing comprehensive value flows in database-backed apps, we develop two DBridge-based clients for database-backed security analysis: a *Stored XSS vulnerability analysis* and a *horizontal broken access control vulnerability analysis* (HBAC Vulnerability analysis). To our knowledge, while such second-order security analyses exist for PHP (Section 7), no current static analysis can detect these database-related vulnerabilities in modern Java apps due to the lack of resolved Java-to-database value flows.

6.4.1 ***Stored XSS Vulnerability Analysis.*** Stored Cross-Site Scripting (XSS) vulnerabilities are critical injection flaws where attackers embed malicious scripts in a website's database, executing when users visit the affected page, posing serious security risks. We evaluate Stored XSS analysis on our real-world apps, with results in Table 3. "N/A" indicates that the vulnerability is not applicable to the app due to its nature. Each identified vulnerability was thoroughly examined, and all test cases that demonstrate the attacks have been made available in the artifact [19].

Table 3. DBridge's results for security.

Application	Stored XSS Analysis	HBAC Analysis
	#identified (#verified)	#identified (#verified)
eladmin	N/A	N/A
mblog	9 (2)	4 (0)
SpringBlog	15 (6)	3 (2)
JavaQuarkBBS	19 (19)	0 (0)
petclinic-JPA	N/A	N/A
petclinic	N/A	N/A
favorites-web	62 (7)	0 (0)
OnlineMall	4 (0)	1 (1)
wallride	25 (0)	N/A
bbs-pro	2 (0)	0 (0)
TOTAL	136 (34)	8 (3)

Results. DBridge identified 34 real Stored XSS attack vulnerabilities across four database-backed apps, 30 of which were previously undiscovered. These vulnerabilities are dangerous as they allow attackers to execute arbitrary malicious code in users' browsers (e.g., cookie leakage, redirection to malicious websites). All of them were confirmed by generating test cases that successfully triggered the attacks in real-world settings. These vulnerabilities primarily stem from developers' unawareness that certain user inputs are stored in the database and later used for webpage rendering, leading to missed security checks. At the time of writing, seven of these vulnerabilities have been confirmed by the app developers as present and worth fixing. As shown in Table 3, DBridge also identified some false positives, mainly due to imprecise sink selection. In practice, only sinks related to rich text rendering are likely to cause an attack. However, due to DBridge's current lack of front-end page analysis capabilities, it cannot precisely identify these specific sinks.

A Case Study. We present a Stored XSS vulnerability identified by DBridge in *favorites-web* (4.8k stars, 1.7k forks on GitHub), an app for managing and sharing favorite websites and resources. The vulnerability is exploited as follows: A malicious user shares a resource on the website, and when a victim bookmarks it and leaves a comment, the attacker replies with a script embedded in their response. When the victim views the reply, the script is executed. Exploiting this vulnerability requires six Web API calls, including user creation, resource sharing, bookmarking, commenting,

replying, and viewing the reply. Parameters between these calls are tightly coupled; for example, the bookmarking API requires the resource ID to match the ID of the resource originally shared by the attacker. These interdependencies make the vulnerability difficult to detect through dynamic approaches like testing and fuzzing. Using DBridge's reported results, we successfully exploited this vulnerability—and five others—in the app by interacting with its public website.

6.4.2 ***HBAC Vulnerability Analysis.*** Horizontal Broken Access Control (HBAC) vulnerabilities allow malicious users to access or manipulate other users' private data. Similar to private information leak analysis, an HBAC vulnerability occurs when high-confidentiality (High) data flows to a low-confidentiality (Low) sink [4]. The key difference is that database interactions propagate the vulnerability. High-confidentiality data is stored in the database via operations like *save*, and attackers retrieve it using actions like *update* and *query*, exploiting publicly accessible data. This leads to High data being exposed at a Low site (e.g., through an HTTP response).

Results. We evaluated the HBAC analysis on our real-world apps, with results shown in the HBAC analysis columns of Table 3. "N/A" means no experiments were conducted on these apps due to the absence of sensitive data. Overall, DBridge identified 8 vulnerabilities across five apps, three of which were confirmed as true and previously undiscovered. These vulnerabilities can leak crucial user privacy information, such as detailed records of products purchased by users. The verification process was the same as for the Stored XSS analysis, and we generated test cases to trigger the three real HBAC vulnerabilities, which are included in the artifact. In the *mblog* app, all identified vulnerabilities were false positives, caused by the app's use of Hibernate's aspect-oriented programming to protect sensitive data, a feature that DBridge currently does not support.

A Case Study. We present an HBAC vulnerability detected in the *SpringBlog* app (1.6k stars on GitHub), a blogging platform that allows users to create public or private posts, with private posts containing sensitive information. This vulnerability allows attackers to exploit public data within the app to access private posts. Here's how the vulnerability works: the app provides a Web API, `showPost`, to retrieve posts via a permalink. When the permalink is used as a query condition, the app generates an SQL query that restricts the post's status to *public* (i.e., only *public* post can be retrieved). However, if no post is found, the app throws an exception. In the exception handling, the permalink is then used as the post's ID for a second query, but this query does not enforce the *public* status restriction. This allows attackers to enumerate post IDs and use them as permalinks to access private posts. In our analysis, DBridge identified that private data in the `content` field of a `post` object flows to the `content` column of the `post` table and is also returned by the `showPost` call, thereby detecting the vulnerability. We deployed *SpringBlog* and generated test cases to successfully exploit this vulnerability, gaining access to confidential information stored in the database.

6.5 RQ5. Can DBridge Assist in Program Understanding for Database-backed Apps?

Impact analysis of database schema changes is a crucial task for understanding and maintaining database-backed apps. When programmers modify the database schema (e.g., altering the data type of a table column) via database management utilities, impact analysis automates the identification of affected app code, particularly the call sites of relevant database access APIs (called database access sites). This enables focused retesting and debugging of the impacted code areas [13, 26, 27].

In impact analysis, when a table column's data type is modified, DBridge identifies the SQL statements that involve the modified column and marks the corresponding database access sites. It also analyzes the code that maps table data to objects to identify the class field related to the modified column. DBridge then tracks the value flow of this field to locate additional impacted database access sites, often involving complex Java-to-database value flows.

Existing static analysis approaches for schema change impact analysis [26, 27] have notable limitations. They are unable to construct value flows across the full spectrum of Java-to-database interactions, and can identify only a very limited subset of SQL statements due to a considerable lack of support for various query types and database framework features [21]. Unfortunately, a direct comparison with these tools is not feasible as they are not open-source.

Table 4. DBridge's results for program understanding.

Application	Impact Analysis	
	#real	#identified (#matched)
eladmin	22	23 (21)
mblog	13	11 (11)
SpringBlog	23	23 (23)
JavaQuarkBBS	10	10 (10)
petclinic-JPA	7	7 (7)
petclinic	7	7 (7)
favorites-web	29	29 (29)
OnlineMall	10	10 (9)
wallride	63	59 (59)
bbs-pro	11	11 (10)
TOTAL	195	190 (186)

We implement the impact analysis based on DBridge, and evaluated it on our real-world apps, by selecting the most complex database table and randomly choosing three columns as the targets for simulated changes. We kept it to three columns because manually verifying which database access sites are affected by the changes is highly time-consuming, particularly when checking if the parameters of a site are impacted by the changes. The results are presented in Table 4. The "#real" column represents the number of runtime-invoked database access sites impacted by the simulated changes. Additionally, we manually inspected all database access sites not covered by dynamic testing but potentially affected by the changes. The "#identified" column shows the number of affected sites identified by DBridge, while the "#matched" column (in parentheses) indicates how many of these identified sites were actually affected, demonstrating an average detection rate of 95%.

This high detection rate is largely due to DBridge's comprehensive building of Java-to-database value flows and its ability to model the semantics of various database framework APIs and SQL. As shown in the table, DBridge also exhibits good precision, as the client only needs column-level precision (as schema changes are typically column-based [26, 27]). Row-level precision—resolving the WHERE clause of SQL—is not necessary here. Additionally, DBridge only needs to identify the affected sites, without distinguishing the specific flows impacting those sites. These factors together contribute to DBridge's high precision in this client.

Lastly, we did not specifically discuss the efficiency of these three clients in RQ4 and RQ5, as DBridge performed similarly well in terms of speed, as observed in the fundamental analysis experiments of RQ2. The maximum analysis time across the three clients was seven minutes, and the primary reasons for DBridge's efficiency have already been explained in Section 6.2.

7 Related Work

Fundamental Analysis for Database-Backed Apps. The analysis of database-backed apps involves resolving both the internal behavior of the app and its interactions with the database. Regarding the app's internal implementation, modern Java apps often rely on frameworks such as Spring and JavaEE, particularly for web development. Several studies have explored different approaches to address the challenges posed by these frameworks [7, 8, 36, 42–44]. In DBridge, we apply an approach inspired by JackEE [7] to handle the complexities of the Spring framework. However, none of these works address interactions between Java apps and databases. These works are limited to analyzing behaviors within the Java app alone and lack the ability to analyze the database framework, SQL, or database itself, preventing the construction of any Java-to-database value flows.

Existing research on the foundational analysis of Java-to-database interactions primarily centers on statically identifying SQL statements to facilitate optimization and maintenance. SLocator [21] is the state-of-the-art approach in this area. In our RQ3 experiment, we compare DBridge with SLocator in terms of SQL identification, showing that DBridge significantly outperforms SLocator. Other SQL identification approaches [15, 16, 28, 32, 45] support noticeably fewer database access

APIs than SLocator, limiting their SQL identification capabilities. Notably, none of these approaches construct Java-to-database value flows like DBridge, restricting them to a narrow set of clients.

In other programming languages like C# and PHP, some studies have attempted to capture app-to-database value flows [11, 12, 39]. However, these studies feature significantly simpler app-to-database interactions compared to those we address, which necessitates fundamentally different approaches. Firstly, the related studies primarily focused on SQL at a syntactic level, only considering table and column names involved in a single SQL statement. In contrast, DBridge delves into the semantics of SQL, accounting for various SQL clauses, predicates, expressions, and aliases. This level of depth led us to propose a dedicated Database Model and introduce diverse SQL primitives. Secondly, the database frameworks we handle are far more complex than those considered in previous work. They only concentrate on simpler JDBC-like database frameworks (e.g., ADO.NET for C#), whereas we further address intricate ORM frameworks like Hibernate and Spring Data JPA. These frameworks offer a wide array of queries, entity relationships, mappings, and persistent states. To address the complexity, we introduced a set of F-primitives. Note that these framework and SQL primitives go beyond simple representations of framework APIs and SQL. For instance, they incorporate newly crafted parameters that are absent in existing framework APIs and SQL, to facilitate pointer analysis. Moreover, designing these primitives required a careful balance of simplicity, diversity, and suitability for pointer analysis. Considering all these factors, we have developed the first end-to-end pointer analysis for Java database-backed apps.

Static Analysis Clients for Database-Backed Apps. In security analysis, Dahse et al. [11] and Su et al. [39] identify second-order injection vulnerabilities, such as Stored XSS, in PHP programs. However, the effectiveness of these analyses is constrained by the limitations of their underlying foundational analyses, which have been detailed in the preceding paragraph (the third paragraph of the "Fundamental Analysis" section above). As shown in our RQ4 experiment, DBridge implements a Stored XSS vulnerability analysis based on its robust ability to construct Java-to-database value flows, enabling it to detect a set of real vulnerabilities that were previously undiscovered.

SQL injection vulnerability analysis is another subject of security research [22, 23, 35, 46]. We believe that DBridge's capabilities could further enhance the effectiveness of these analyses, with tool development on top of DBridge planned for future work.

In program understanding, several studies have focused on database-backed apps, including impact analysis [26, 27], code smells [10, 31], and anti-patterns analysis [9, 24]. Among these, DBridge implements and evaluates impact analysis as a representative client, achieving promising results (see the RQ5 experiment). Unfortunately, since these studies are not open-source, a direct comparison is not possible. However, we discuss their methodological differences in Section 6.5.

8 Conclusion

We introduce DBridge, the first pointer analysis specifically designed for Java database-backed apps, capable of statically constructing comprehensive Java-to-database value flows. By integrating Java app code analysis, database access specification modeling, SQL analysis, and database abstraction within a unified pointer analysis, DBridge addresses a critical gap in static analysis approaches that have struggled to capture the full complexity of Java-database interactions. Through extensive evaluation on the DB-Micro micro-benchmark suite and real-world apps, we demonstrate DBridge's robustness and effectiveness, achieving high recall and precision in value flow construction, while outperforming state-of-the-art tools in SQL statement detection. Moreover, we validate DBridge's practical utility by developing client analyses that uncover previously undetected real vulnerabilities and showcase its potential for facilitating program understanding tasks. The open-sourcing of DBridge and DB-Micro offers a foundation for further research and development. We believe our work will make a meaningful contribution to advancing static analysis for database-backed apps.

Acknowledgments

We would like to thank the anonymous reviewers for their helpful comments. This work is supported in part by National Key R&D Program of China under Grant No. 2023YFB4503804, National Natural Science Foundation of China under Grant Nos. 62402210, 62025202, the Frontier Technologies R&D Program of Jiangsu under Grant No. BF2024059, the Leading-edge Technology Program of Jiangsu Natural Science Foundation under Grant No. BK20202001, and the Collaborative Innovation Center of Novel Software Technology and Industrialization, Jiangsu, China. Tian Tan, the co-corresponding author, is also supported by Xiaomi Foundation.

Data-Availability Statement

We have provided an artifact [19] to reproduce all experimental results presented in Section 6. The artifact includes the source code of DBridge, the DB-Micro, the real-world apps used in our experiments, as well as detailed documentation, scripts, and demonstration videos for reproducing the vulnerabilities revealed by DBridge. The artifact is available at https://doi.org/10.5281/zenodo.15171408. To reproduce the results, please refer to the instructions provided in the accompanying `README.pdf` document within the artifact.

In addition, we have provided supplementary material [20] that formalizes all core primitives and the associated pointer analysis rules. This document also includes a more detailed description of DB-Micro. You can access this supplementary material at the following URL https://doi.org/10.5281/zenodo.15167168.

References

[1] 2024. Hibernate. Retrieved November 1, 2024 from https://github.com/hibernate/hibernate-orm

[2] 2024. Java JDBC API. Retrieved November 1, 2024 from https://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/

[3] 2024. Java Persistence API. Retrieved November 1, 2024 from https://www.oracle.com/java/technologies/persistence-jsp.html

[4] 2024. OWASP Top Ten. Retrieved November 1, 2024 from https://owasp.org/www-project-top-ten/

[5] 2024. Spring Data JPA. Retrieved November 1, 2024 from https://github.com/spring-projects/spring-data-jpa

[6] 2024. Spring Framework. Retrieved November 1, 2024 from https://github.com/spring-projects/spring-framework

[7] Anastasios Antoniadis, Nikos Filippakis, Paddy Krishnan, Raghavendra Ramesh, Nicholas Allen, and Yannis Smaragdakis. 2020. Static analysis of Java enterprise applications: frameworks and caches, the elephants in the room. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, Alastair F. Donaldson and Emina Torlak (Eds.). ACM, 794–807. doi:10.1145/3385412.3386026

[8] Miao Chen, Tengfei Tu, Hua Zhang, Qiaoyan Wen, and Weihang Wang. 2022. Jasmine: A Static Analysis Framework for Spring Core Technologies. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*. ACM, 60:1–60:13. doi:10.1145/3551349.3556910

[9] Tse-Hsun Chen, Weiyi Shang, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed N. Nasser, and Parminder Flora. 2014. Detecting performance anti-patterns for applications developed using object-relational mapping. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, Pankaj Jalote, Lionel C. Briand, and André van der Hoek (Eds.). ACM, 1001–1012. doi:10.1145/2568225.2568259

[10] Tse-Hsun Chen, Weiyi Shang, Zhen Ming Jiang, Ahmed E. Hassan, Mohamed N. Nasser, and Parminder Flora. 2016. Finding and Evaluating the Performance Impact of Redundant Data Access for Applications that are Developed Using Object-Relational Mapping Frameworks. *IEEE Trans. Software Eng.* 42, 12 (2016), 1148–1161. doi:10.1109/TSE.2016.2553039

[11] Johannes Dahse and Thorsten Holz. 2014. Static Detection of Second-Order Vulnerabilities in Web Applications. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*, Kevin Fu and Jaeyeon Jung (Eds.). USENIX Association, 989–1003. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/dahse

[12] Arjun Dasgupta, Vivek R. Narasayya, and Manoj Syamala. 2009. A Static Analysis Framework for Database Applications. In *Proceedings of the 25th International Conference on Data Engineering, ICDE 2009, March 29 2009 - April 2 2009, Shanghai, China*, Yannis E. Ioannidis, Dik Lun Lee, and Raymond T. Ng (Eds.). IEEE Computer Society, 1403–1414.

doi:10.1109/ICDE.2009.98

[13] Julien Delplanque, Anne Etien, Nicolas Anquetil, and Olivier Auverlot. 2018. Relational Database Schema Evolution: An Industrial Case Study. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 635–644. doi:10.1109/ICSME.2018.00073

[14] Mathieu Goeminne and Tom Mens. 2015. Towards a survival analysis of database framework usage in Java projects. In *2015 IEEE International Conference on Software Maintenance and Evolution, ICSME 2015, Bremen, Germany, September 29 - October 1, 2015*, Rainer Koschke, Jens Krinke, and Martin P. Robillard (Eds.). IEEE Computer Society, 551–555. doi:10.1109/ICSM.2015.7332512

[15] Carl Gould, Zhendong Su, and Premkumar T. Devanbu. 2004. Static Checking of Dynamically Generated Queries in Database Applications. In *26th International Conference on Software Engineering (ICSE 2004), 23-28 May 2004, Edinburgh, United Kingdom*, Anthony Finkelstein, Jacky Estublier, and David S. Rosenblum (Eds.). IEEE Computer Society, 645–654. doi:10.1109/ICSE.2004.1317486

[16] Ding Li, Yingjun Lyu, Mian Wan, and William G. J. Halfond. 2015. String analysis for Java and Android applications. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, Elisabetta Di Nitto, Mark Harman, and Patrick Heymans (Eds.). ACM, 661–672. doi:10.1145/2786805.2786879

[17] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2020. A Principled Approach to Selective Context Sensitivity for Pointer Analysis. *ACM Trans. Program. Lang. Syst.* 42, 2 (2020), 10:1–10:40. doi:10.1145/3381915

[18] Yue Li, Tian Tan, and Jingling Xue. 2019. Understanding and Analyzing Java Reflection. *ACM Trans. Softw. Eng. Methodol.* 28, 2 (2019), 7:1–7:50. doi:10.1145/3295739

[19] Yufei Liang, Teng Zhang, Ganlin Li, Tian Tan, Chang Xu, Chun Cao, Xiaoxing Ma, and Yue Li. 2025. *Pointer Analysis for Database-Backed Applications (Artifact)*. doi:10.5281/zenodo.15171408

[20] Yufei Liang, Teng Zhang, Ganlin Li, Tian Tan, Chang Xu, Chun Cao, Xiaoxing Ma, and Yue Li. 2025. Pointer Analysis for Database-Backed Applications (Supplementary Material). doi:10.5281/zenodo.15167168

[21] Wei Liu and Tse-Hsun Chen. 2023. SLocator: Localizing the Origin of SQL Queries in Database-Backed Web Applications. *IEEE Trans. Software Eng.* 49, 6 (2023), 3376–3390. doi:10.1109/TSE.2023.3253700

[22] V. Benjamin Livshits and Monica S. Lam. 2005. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *Proceedings of the 14th USENIX Security Symposium, Baltimore, MD, USA, July 31 - August 5, 2005*, Patrick D. McDaniel (Ed.). USENIX Association. https://www.usenix.org/conference/14th-usenix-security-symposium/finding-security-vulnerabilities-java-applications-static

[23] Changhua Luo, Penghui Li, and Wei Meng. 2022. TChecker: Precise Static Inter-Procedural Analysis for Detecting Taint-Style Vulnerabilities in PHP Applications. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi (Eds.). ACM, 2175–2188. doi:10.1145/3548606.3559391

[24] Yingjun Lyu, Sasha Volokh, William G. J. Halfond, and Omer Tripp. 2021. SAND: a static analysis approach for detecting SQL antipatterns. In *ISSTA '21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, Denmark, July 11-17, 2021*, Cristian Cadar and Xiangyu Zhang (Eds.). ACM, 270–282. doi:10.1145/3460319.3464818

[25] Wenjie Ma, Shengyuan Yang, Tian Tan, Xiaoxing Ma, Chang Xu, and Yue Li. 2023. Context Sensitivity without Contexts: A Cut-Shortcut Approach to Fast and Precise Pointer Analysis. *Proc. ACM Program. Lang.* 7, PLDI (2023), 539–564. doi:10.1145/3591242

[26] Andy Maule, Wolfgang Emmerich, and David S. Rosenblum. 2008. Impact analysis of database schema changes. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*, Wilhelm Schäfer, Matthew B. Dwyer, and Volker Gruhn (Eds.). ACM, 451–460. doi:10.1145/1368088.1368150

[27] Loup Meurice, Csaba Nagy, and Anthony Cleve. 2016. Detecting and Preventing Program Inconsistencies under Database Schema Evolution. In *2016 IEEE International Conference on Software Quality, Reliability and Security, QRS 2016, Vienna, Austria, August 1-3, 2016*. IEEE, 262–273. doi:10.1109/QRS.2016.38

[28] Loup Meurice, Csaba Nagy, and Anthony Cleve. 2016. Static Analysis of Dynamic Database Usage in Java Systems. In *Advanced Information Systems Engineering - 28th International Conference, CAiSE 2016, Ljubljana, Slovenia, June 13-17, 2016. Proceedings (Lecture Notes in Computer Science, Vol. 9694)*, Selmin Nurcan, Pnina Soffer, Marko Bajec, and Johann Eder (Eds.). Springer, 491–506. doi:10.1007/978-3-319-39696-5_30

[29] Ana L. Milanova, Atanas Rountev, and Barbara G. Ryder. 2002. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *Proceedings of the International Symposium on Software Testing and Analysis, ISSTA 2002, Roma, Italy, July 22-24, 2002*, Phyllis G. Frankl (Ed.). ACM, 1–11. doi:10.1145/566172.566174

[30] Ana L. Milanova, Atanas Rountev, and Barbara G. Ryder. 2005. Parameterized object sensitivity for points-to analysis for Java. *ACM Trans. Softw. Eng. Methodol.* 14, 1 (2005), 1–41. doi:10.1145/1044834.1044835

[31] Biruk Asmare Muse, Mohammad Masudur Rahman, Csaba Nagy, Anthony Cleve, Foutse Khomh, and Giuliano Antoniol. 2020. On the Prevalence, Impact, and Evolution of SQL Code Smells in Data-Intensive Systems. In *MSR '20: 17th*

International Conference on Mining Software Repositories, Seoul, Republic of Korea, 29-30 June, 2020, Sunghun Kim, Georgios Gousios, Sarah Nadi, and Joseph Hejderup (Eds.). ACM, 327–338. doi:10.1145/3379597.3387467

[32] Csaba Nagy and Anthony Cleve. 2018. SQLInspect: a static analyzer to inspect database usage in Java applications. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 93–96. doi:10.1145/3183440.3183496

[33] Yannis Smaragdakis and George Balatsouras. 2015. Pointer Analysis. *Found. Trends Program. Lang.* 2, 1 (2015), 1–69. doi:10.1561/2500000014

[34] Yannis Smaragdakis, George Kastrinis, and George Balatsouras. 2014. Introspective analysis: context-sensitivity, across the board. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 485–495. doi:10.1145/2594291.2594320

[35] Fausto Spoto, Elisa Burato, Michael D. Ernst, Pietro Ferrara, Alberto Lovato, Damiano Macedonio, and Ciprian Spiridon. 2019. Static Identification of Injection Attacks in Java. *ACM Trans. Program. Lang. Syst.* 41, 3 (2019), 18:1–18:58. doi:10.1145/3332371

[36] Manu Sridharan, Shay Artzi, Marco Pistoia, Salvatore Guarnieri, Omer Tripp, and Ryan Berg. 2011. F4F: taint analysis of framework-based web applications. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*. 1053–1068.

[37] Manu Sridharan, Satish Chandra, Julian Dolby, Stephen J. Fink, and Eran Yahav. 2013. Alias Analysis for Object-Oriented Programs. In *Aliasing in Object-Oriented Programming*, David Clarke, Tobias Wrigstad, and James Noble (Eds.). Springer. doi:10.1007/978-3-642-36946-9_8

[38] Manu Sridharan, Stephen J Fink, and Rastislav Bodík. 2007. Thin slicing. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*. doi:10.1145/1250734.1250748

[39] He Su, Feng Li, Lili Xu, Wenbo Hu, Yujie Sun, Qing Sun, Huina Chao, and Wei Huo. 2023. Splendor: Static Detection of Stored XSS in Modern Web Applications. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, René Just and Gordon Fraser (Eds.). ACM, 1043–1054. doi:10.1145/3597926.3598116

[40] Tian Tan and Yue Li. 2023. Tai-e: A Developer-Friendly Static Analysis Framework for Java by Harnessing the Good Designs of Classics. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, René Just and Gordon Fraser (Eds.). ACM, 1093–1105. doi:10.1145/3597926.3598120

[41] Tian Tan, Yue Li, Xiaoxing Ma, Chang Xu, and Yannis Smaragdakis. 2021. Making pointer analysis more precise by unleashing the power of selective context sensitivity. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–27. doi:10.1145/3485524

[42] John Toman and Dan Grossman. 2019. Concerto: a framework for combined concrete and abstract interpretation. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29.

[43] Omer Tripp, Marco Pistoia, Patrick Cousot, Radhia Cousot, and Salvatore Guarnieri. 2013. Andromeda: Accurate and Scalable Security Analysis of Web Applications. In *Fundamental Approaches to Software Engineering - 16th International Conference, FASE 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7793)*, Vittorio Cortellessa and Dániel Varró (Eds.). Springer, 210–225. doi:10.1007/978-3-642-37057-1_15

[44] Omer Tripp, Marco Pistoia, Stephen J Fink, Manu Sridharan, and Omri Weisman. 2009. TAJ: effective taint analysis of web applications. *ACM Sigplan Notices* 44, 6 (2009), 87–97.

[45] Mario Linares Vásquez, Boyang Li, Christopher Vendome, and Denys Poshyvanyk. 2016. Documenting database usages and schema constraints in database-centric applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, Andreas Zeller and Abhik Roychoudhury (Eds.). ACM, 270–281. doi:10.1145/2931037.2931072

[46] Gary Wassermann and Zhendong Su. 2007. Sound and precise analysis of web applications for injection vulnerabilities. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, Jeanne Ferrante and Kathryn S. McKinley (Eds.). ACM, 32–41. doi:10.1145/1250734.1250739

Received 2024-11-14; accepted 2025-03-06