BOOMSLANG: A Modular Framework for Database Transaction Isolation Checking

Jian Zhang
Northeastern University
Boston, USA

Shuai Mu
Stony Brook University
Stony Brook, USA

Cheng Tan
Northeastern University
Boston, USA

Abstract

Checking whether database transactions adhere to isolation levels is a crucial yet challenging problem. Existing checkers are typically designed for a specific combination of isolation level, operation types, and workload assumptions. As a result, supporting a new application scenario often requires building a new checker from scratch. We present BOOMSLANG, an extensible framework for checking the isolation of transactional database executions using only client-side traces, without requiring knowledge of database internals.

BOOMSLANG adopts a front-/back-end separation. As the front-end, it parses a database trace into an *Abstract Semantic Graph*, which is then lowered into a low-level intermediate representation (IR). The back-end converts this IR to a set of constraints for satisfiability modulo theories (SMT) solving. This design is enabled by a key abstraction in the IR, called *superpositions*, which represent heterogeneous forms of uncertainty—including unknown version orders, ambiguous read-from relationships, and physical deletions—as a single first-class IR object. Isolation levels, operation semantics, and database-specific behaviors are implemented as modules that generate this IR over a shared back-end. Our experiments show that with only a few hundred lines of code, the core logic of three prior checkers can be reimplemented as BOOMSLANG modules. Using BOOMSLANG, we also support eight isolation levels, from read committed to strict serializability, identify a new bug in TiDB, audit the metadata layer of the JuiceFS file system, check vendor-specific behavior in MariaDB, and confirm a theoretical result on the correctness of strict serializability. BOOMSLANG is open source: https://github.com/winddd/SOSH [29].

CCS Concepts: • **Information systems → Database management system engines**; • **General and reference → Verification**.

Keywords: Transaction isolation, Database checking

ACM Reference Format:
Jian Zhang, Shuai Mu, and Cheng Tan. 2026. BOOMSLANG: A Modular Framework for Database Transaction Isolation Checking. In *The 2026 ACM SIGOPS Annual Technical Conference (SIGOPS ATC 2026), November 15-18, 2026, Hong Kong, China.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3828523.3857054



This work is licensed under a Creative Commons Attribution 4.0 International License.

SIGOPS ATC 2026, Hong Kong, China
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2818-1/2026/11
https://doi.org/10.1145/3828523.3857054

1 Introduction

Transactional databases play a crucial role in today's digital world, yet ensuring their correctness is challenging due to bugs [45, 47, 54], failures [32, 87], errors [46, 77], and security threats [75, 81, 82]. Now, consider an "oracle" that serves as a decisive mechanism: given a database execution trace, it can either confirm correctness or reject it, providing explanations for any incorrect behavior. This oracle would be extremely valuable across various applications. During testing, it verifies whether a database behaves as expected under different workloads. In auditing, it provides a thorough review of potential errors and failures arising from underlying software and hardware issues. For outsourced computing environments where trust cannot be assumed, the oracle validates data integrity and execution against potential tampering.

In this work, we build BOOMSLANG, an extensible checking framework that approximates such an oracle for transactional key-value store executions, producing sound and complete verdicts with respect to the given trace and isolation specification. We focus on transactional key-value stores (which we refer to as "databases" throughout this paper for brevity) as they are widely used in practice and serve as core building blocks for many complex systems, such as SQL databases [52, 74] and file systems [12].

BOOMSLANG takes two inputs: a trace of a database execution and a target isolation level. The trace is collected at the clients: it records the operations that clients issue to the database, their arguments, and the results the database returns, grouped into transactions. BOOMSLANG then decides whether some execution that is consistent with what the clients observed is also allowed by the target isolation level. If such an execution exists, BOOMSLANG accepts the trace; otherwise, it rejects the trace and reports a counterexample that explains the violation.

What distinguishes BOOMSLANG from existing checkers [38, 41–43, 51, 53, 56, 57, 59, 61, 67, 73, 75, 78, 85] is its focus on extensibility: checking a new isolation level, operation type, or database-specific behavior should not require designing a new checker from scratch. In particular, BOOMSLANG has three key properties:

1. BOOMSLANG features a modular pipeline in which isolation levels, operation semantics, and database-specific behaviors are implemented as modules that generate a shared intermediate representation (IR). Users can check new isolation levels and consistency properties by writing such modules while reusing BOOMSLANG's parsing, optimizations, and its back-end solver.
2. BOOMSLANG supports operations beyond reads and writes that modern transactional key-value stores provide, including range queries, iterators, and physical deletions. Many prior checkers restrict operation types or rely on workload conventions (e.g., unique written values) that make dependencies easy to infer; such conventions do not hold for arbitrary application traces.
3. BOOMSLANG does not require users to understand database internals (e.g., the concurrency control protocols in use) and enables checking with minimal logging. It uses a lightweight logging scheme called *envelope tracing*, which records only the inputs and outputs of database operations at the client side and can be easily adopted without modifying application logic or the database—making it accessible to non-experts.

To achieve these properties, BOOMSLANG adopts a modular pipeline similar to a compiler's workflow: (1) it logs inputs to and outputs from the database from the client's perspective, creating a trace that represents the database execution. (2) It parses the trace into a data structure called the *Abstract Semantic Graph* (ASG). The ASG contains the necessary information to reason about trace correctness. It extends a simple graph with additional metadata needed to check different correctness properties. (3) BOOMSLANG lowers the ASG into a low-level intermediate representation (IR) via semantic analysis. Grounded in prior research [30, 31, 75, 85, 86], the IR logically represents the set of possible direct serialization graphs (DSGs) that could explain the trace, expressing everything the trace leaves unknown as superpositions (described below). Isolation levels are encoded as constraints over these DSGs. BOOMSLANG checks whether any DSG represented by the IR satisfies the target isolation-level constraints. The semantic analysis also accepts user-defined hints to accommodate different setups and assumptions. (4) BOOMSLANG applies optimizations and generates SMT constraints from the IR. In particular, BOOMSLANG introduces three optimizations: reachability pruning, topological prioritization, and unsat search. (5) BOOMSLANG solves the constraints with an SMT solver and reports the verdict.

The key abstraction in this pipeline is the superposition. Reasoning about unknowns itself, however, is not new. Bernstein et al. [35, 36] define (multiversion) serializability through the existence of a suitable version order; Papadimitriou's polygraphs [70] encode alternative orderings as choices; and Seeing is Believing [44] defines isolation through the existence of an execution. Prior checkers instantiate these ideas for the specific forms of uncertainty in their target settings. BOOMSLANG generalizes them: it captures heterogeneous forms of uncertainty—unknown version orders, ambiguous read-from relationships, physical deletions, and keys skipped by range queries and iterators—with a single first-class IR object, called a *superposition*: a set of mutually exclusive possibilities that could explain part of the trace. With superpositions, people can support a new isolation level or operation type by writing modules that generate corresponding IR constraints.

We have implemented BOOMSLANG and evaluated it against state-of-the-art checkers. As evidence of extensibility, the core encodings of Cobra [75], Viper [85], and PolySI [53] are each implemented in BOOMSLANG with 271–798 lines of code.

We further use BOOMSLANG to build checkers for diverse configurations: (1) implementing new encodings for eight isolation levels, from read committed [30, 34] to strict serializability, including repeatable read [30, 34], PL-2+ [30], cursor stability [30], and PL-FCV [30], over traces with range queries, iterators, and physical deletions; (2) testing a popular real-world application—a distributed file system JuiceFS [12]—to audit its metadata management; (3) supporting a vendor-specific behavior of `SELECT FOR UPDATE` in MariaDB [14] that breaks traditional transactions' visibility; (4) confirming the previously theorized timestamp inversion pitfall [62] in a real implementation. In addition, while cross-comparing behaviors across databases, BOOMSLANG discovered a new bug [16] in TiDB.

Beyond these use cases, BOOMSLANG's extensible design offers two further benefits. First, it enables a fair comparison between different checkers. For example, while Viper and PolySI are both SI checkers, and previous results showed Viper to be slower than PolySI, implementing both in BOOMSLANG reveals comparable or better performance for Viper's encoding, indicating that the reported performance gap stemmed from implementation differences rather than inherent checker differences. Second, BOOMSLANG can share optimizations across different configurations. For instance, topological prioritization is enabled by default for all checkers.

Limitations. BOOMSLANG is not without limitations. First, BOOMSLANG does not change the NP-complete worst case of black-box checking; like other black-box checkers, it cannot guarantee verification of all traces within bounded time, and pathological or adversarial traces—for example, from a database that attempts to exhaust the checker—may time out despite the optimizations. Second, BOOMSLANG's guarantees are relative to its inputs: the trace must contain all transactions relevant to the observed data, which generally requires instrumenting all clients, and user-provided hints and modules are trusted. Third, BOOMSLANG does not support full SQL semantics, a limitation common to existing checkers. While BOOMSLANG's design is, in principle, general enough to accommodate SQL semantics, efficiently translating rich SQL semantics to SMT constraints remains an open problem [71].

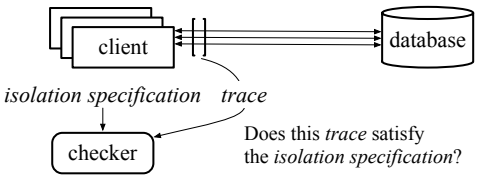


Figure 1. Problem setup.

2 Setup, motivation, and applicability

2.1 Problem setup

In this paper, we consider the following setup. Multiple clients simultaneously send requests to a database. The database operates on the requests in a concurrent manner and sends responses back to the clients. These clients and the external world observe the requests to and responses from the database, which we refer to as a *trace*. A checker then examines this trace and determines whether the database faithfully executes clients' requests. Figure 1 depicts this setup.

We consider transactional operations: all client operations are wrapped in transactions. The operations that today's transactional key-value stores provide are: puts, gets, deletions, range queries, and iterators.

We assume that the trace contains all transactions relevant to the observed data. For a multi-client execution, this generally requires instrumenting all participating clients.

The database claims to uphold some isolation specifications. A canonical example is serializability, an *isolation level* defined by ANSI/ISO and widely adopted in the database community [30, 44]. These specifications are typically configured by the client, establishing an agreement between the client and the database on the expected execution semantics and transaction isolation guarantees.

This agreed-upon isolation specification is verified by a checker. The checker evaluates the trace against the specification and returns a verdict: accept if the database satisfies the isolation guarantees, or reject if it violates them. Crucially, the checking must be sound and complete:

- *Soundness*: If the checker accepts, the trace satisfies the specification. (That is, if the trace violates the isolation guarantees, the checker will reject.)
- *Completeness*: If the trace satisfies the specification, the checker will accept. (That is, if the checker rejects, the trace violates the isolation guarantees.)

Crucially, Boomslang does not require internal scheduler knowledge. When only client-side observations are available, it operates on *observational equivalence* [44]. Boomslang lowers the trace into an IR representing a set of possible DSGs [30, 31]. Thus, our soundness and completeness claims apply strictly to the external trace; Boomslang accepts traces that are observationally equivalent to a valid execution under the target specification, which is a fundamental trade-off of black-box checking. There are checkers [50, 59, 72, 76, 83] that sacrifice either completeness or soundness or both for performance, which is not the target of this paper. We focus on complete and sound database checking.

Existing checkers. Existing checkers make different, well-motivated tradeoffs. Cobra [75], Viper [85], PolySI [53], and Plume [61] identify read-from relationships using unique written values. Elle [56] exposes dependencies through specialized workloads, while Emme [42] recovers version orders and version sets from database internals. We discuss these systems in §9. In each case, the encoding is co-designed with its target isolation level and assumptions. For example, Cobra-style formulations encode unknown orders as pairwise binary choices and couple different dependency types together [4, 53, 85]. Physical deletions and iterators, however, introduce ambiguities with more than two possibilities: if an `iterator` does not return key x, x may never have been inserted, or may have been deleted by one of several `delete` operations. Likewise, weaker levels such as read committed keep some dependency types while excluding others. Extending such a checker to a new operation type, isolation level, or database-specific behavior therefore often requires a new encoding. The underlying challenge is to represent dependency unknowns independently of any single checker encoding.

2.2 Motivating examples

Our goal is to make isolation checking extensible. In practice, traces include advanced operators, vendor-specific isolation behaviors, duplicate values, physical deletions, and combinations of isolation guarantees. Together, these challenges introduce complex semantics that are difficult to formalize and verify, and require a checker that is flexible enough to support diverse operators, semantics, and isolation specifications. Below we highlight four motivating examples.

Case #1: checking real-world workloads and applications. JuiceFS [12], a distributed file system, stores file-system metadata in a transactional key-value store, where metadata inconsistencies can cause severe failures. Auditing its metadata operations with a checker is therefore desirable for strong correctness assurance. However, JuiceFS uses the full operator set, including iterators, which no existing checker supports. Existing checkers [4, 53, 61, 85] typically focus on point queries with relatively easy-to-identify read-from relations. An iterator's missing key, however, may never have existed, have been deleted, or not yet have been reached, resulting in a larger ambiguity. Prior tools lack an abstraction and efficient handling for this greater uncertainty.

Case #2: supporting vendor-specific semantics. Vendor-specific behavior introduces further challenges for isolation checking. For example, under the *Repeatable Read* isolation level [30, 34], many databases implement `SELECT FOR UPDATE` as returning the most recent version of a record rather than a transactionally consistent snapshot [14]. This vendor-supported behavior diverges from textbook definitions and

can make existing checkers reject vendor-expected valid executions. A checking framework should allow such visibility rules to be customized without redesigning the checker.

Case #3: relaxing assumptions that may not hold in practice. Existing checkers make various assumptions about workloads and applications, e.g., all values written are unique to disambiguate read-from dependencies, but applications can write duplicates, and a MariaDB bug appears only when duplicate values are written [26]. Similarly, *tombstones* [85] simplify deletion by replacing physical deletion with a logical marker, but databases do not implement deletion this way. Supporting different workload assumptions requires a common representation of the resulting ambiguities, so that extensions can reuse the checking back-end.

Case #4: checking combined isolation guarantees. In practice, systems expose combinations of isolation guarantees. For example, databases that support both serializability and read committed may differ in whether a transaction can read its own writes. Such variations cannot be handled by simple patches, since these behaviors must be encoded as part of the isolation specification rather than bolted on afterward. As a technicality, serializability relies on *anti-dependencies* [30], a concept that does not exist in read committed; so when modeling read-your-own-write for read committed, one cannot simply "turn off" anti-dependencies without redefining the model itself. Supporting such diversity requires a framework with modular structure and suitable abstractions so that new concurrency constraints can be expressed and checked without redesigning the entire system.

2.3 Application scenarios

An isolation-checking framework provides infrastructure for validating database concurrency correctness. We present three scenarios that illustrate how it can be applied.

Scenario #1: serving as a test oracle. Concurrency testing needs an oracle to check whether diverse interleavings violate isolation guarantees. Frameworks such as Jepsen [8] often rely on specially designed workloads for soundness and completeness. A unified framework instead validates execution traces against specified concurrency semantics across real-world workloads. Such principled validation is impossible without comprehensive support for real-world workloads.

Scenario #2: acting as an auditor for data integrity. Failures are inevitable in both software and hardware. Users need assurance that their data remains consistent across failures, recoveries, and concurrent transactions. Silent hardware corruption is also a concern: recent work [46, 77] shows that even CPUs can silently produce incorrect results. A checker can help detect such corruptions by validating final outcomes without relying on internal information.

Scenario #3: verifying outsourced databases. For outsourced databases, especially those backing critical applications such as finance and healthcare, a checker lets owners validate correctness without trusting the cloud provider.

More potential applications. More broadly, a checker enables unexplored applications: auditing databases against external attacks for security, proving database integrity for forensics, and distinguishing suitable isolation levels for performance tuning. These highlight a chicken-and-egg problem—without a unified, extensible framework, researchers are unable to build checkers for niche use cases, but without demonstrated use cases, no one builds the extensible framework. *We hope our isolation-checking framework, Boomslang, breaks this cycle and opens new research directions.*

3 System overview

Boomslang accepts a trace and checks if the trace satisfies expected isolation levels and other properties. Boomslang provides native support for major isolation levels, including strict serializability, serializability, snapshot isolation, and read committed. In addition, Boomslang exposes an intermediate representation (IR) that enables users to extend trace semantics and define custom isolation properties by implementing modular extensions. Figure 2 depicts the workflow of Boomslang.

System overview. At a high level, Boomslang translates a trace, under a specified isolation level, into a set of logical formulas that are machine checkable. This process mirrors traditional program compilation: the trace corresponds to the program, and different isolation levels correspond to different programming languages, each with its own semantics. Boomslang introduces an abstract semantic data structure and an intermediate representation (IR) that capture the common structure of isolation checking. The IR is then lowered into logical formulas—SMT constraints—that are solved by a specialized SMT solver.

Boomslang's generalizability comes from its separation of front-end and back-end. To support a new isolation level (e.g., PL-2+ [30, §4.1]) or a variant of an existing one (e.g., snapshot isolation variants [44, Fig4]), people need to modify only the relevant pipeline stage—analogous to adding a new programming language or dialect. Meanwhile, if new optimization strategies or specialized SMT techniques become available, they can be integrated into the back-end and automatically benefit all isolation-checking tasks.

This separation is enabled by Boomslang's IR. At its core is a unified abstraction: *superpositions* (§5.2), a concept that captures unknown dependencies in the trace. Superpositions encode every feasible way the trace could have been generated by a database that correctly enforces the specified isolation level. By solving the resulting constraints, the system produces either a witness demonstrating correctness or a counterexample revealing a violation.

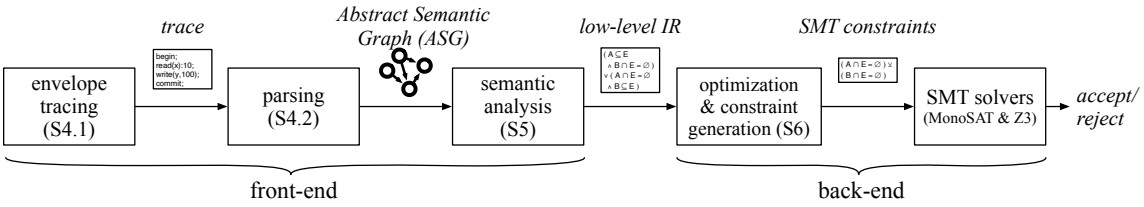


Figure 2. BOOMSLANG pipeline.

Pipeline design of BOOMSLANG. The BOOMSLANG pipeline comprises five phases:

Phase 1: tracing. BOOMSLANG introduces a simple and intuitive tracing mechanism called *envelope tracing* (§4.1). By instrumenting *only* the code immediately before and after each database invocation, the tracer collects all information required for checking without intruding on application logic. The interface is flexible: users can supply additional hints that capture domain-specific knowledge relevant to checking. These hints can later be processed through *user-defined modules* (§5.3) and used to accelerate the checking pipeline.

Phase 2: parsing. In the parsing phase, BOOMSLANG constructs an *Abstract Semantic Graph (ASG)* that captures the structural information embedded in the trace. The ASG consists of three key components: a graph representation, a read lineage, and a write history. This phase effectively converts a trace into a structured representation suitable for analysis.

Phase 3: semantic analysis. The semantic analysis phase transforms the ASG into a low-level intermediate representation (IR) that we designed. This IR includes simple graphs, first-order logic expressions that describe relationships between edges, and superpositions. Superpositions represent "known unknowns"—ambiguities in the trace where certain relationships must exist but cannot be precisely determined from the available information. For example, superpositions can capture the uncertainty in the ordering of concurrent write operations: they are conflicting and therefore must have an order, but without internal scheduling information we cannot determine their order. Meanwhile, if a trace provides complete information without ambiguities, no superpositions will be generated. User-defined modules apply at this phase.

Phase 4: optimization and constraint generation. This phase encodes the low-level IR into a set of constraints suitable for SMT solving. Superpositions are encoded as mutually exclusive options. Optimization passes are applied during this phase to improve solver performance. These optimizations traverse the IR and apply equivalent transformations that preserve correctness while producing constraints that can be solved more efficiently.

Phase 5: solving. In the final phase, BOOMSLANG uses an SMT solver to decide whether the trace satisfies the specified isolation level. By default, it uses MonoSAT [33], whose native graph primitives are well suited to the generated graph constraints; for compatibility, BOOMSLANG can also emit classic SMT clauses for solvers such as Z3 [28]. The solver returns either `sat`, confirming compliance, or `unsat`, with a counterexample explaining the violation.

4 Tracing and parsing

4.1 Client interface and envelope tracing

In practice, clients interact with databases through various interfaces, such as JDBC [68], or language-specific APIs provided by the database. While these interfaces may differ in aspects like function names and arguments, modern transactional key-value stores expose a common core of operations:

- *Transaction control*: `begin`, `commit`, `abort`
- *Point queries*: `put`, `get`, `delete`
- *Predicate-based queries*: `scan`, `iterator`

Most operations are self-explanatory. `scan` returns all items in a key range `[start, end)`, while `iterator` lets clients fetch range-query results incrementally, e.g., for top-K queries. For example, after `it←Iter(key0, key9)`, repeated `it.Next()` calls return successive items.

Although common, `iterator` and combinations such as `delete` with `scan` are unsupported by existing checkers. This limitation is due to several challenges, including verifying that returned items fall within the top-K, and that non-returned items either lie outside the specified range or do not exist—conditions that are often partially unknown in the trace. We will elaborate on these challenges later (§5.1).

Envelope tracing. Logging the necessary information in the trace is essential for correctness checking. However, instrumenting databases is difficult—if not impossible—especially for modern distributed databases that are highly optimized and continuously evolving. Parsing database metadata also requires significant expertise. For example, Emme [42] reconstructs execution traces by parsing change data capture (CDC) logs: it uses the `CHANGEFEED` mechanism for CockroachDB, the Debezium CDC tool for PostgreSQL, and heap scanning for TiDB [42, §7.1].

To enable practical logging, *envelope tracing* instruments the client-side database interface before and after each operation, recording its inputs before invocation and its output afterward:

```
# "ctx" is a dictionary, containing...
# ...context-specified information (e.g., hints)
log_input(op, args, ctx) # log inputs to database
ret = op(args)           # issue op to database
log_output(ret)          # log output from database
```

Envelope tracing does not require modifications to either the application logic or the databases. Developers only need to instrument the interfaces. For BOOMSLANG, we create a set of wrappers for different databases and their APIs. The checked trace combines records from all participating clients, as assumed in §2.1.

Envelope tracing captures only the minimal information necessary for checking. As a result, the checking problem becomes challenging. This trade-off is expected: more detailed traces simplify checking and improve checking performance, but at the cost of more intrusive logging, tighter coupling to specific applications, or reliance on database internals. Existing checkers typically choose a fixed point along this trade-off curve, while BOOMSLANG supports the full spectrum.

BOOMSLANG adopts a modular design. By default, it accepts the minimal trace from envelope tracing. Yet, when additional information—for example, the information extracted from CDC logs—is available, developers can write a module for the semantic analysis phase (§5) and BOOMSLANG can then leverage it to improve checking efficiency. Meanwhile, BOOMSLANG provides common modules to manage trade-offs that have been explored by existing checkers.

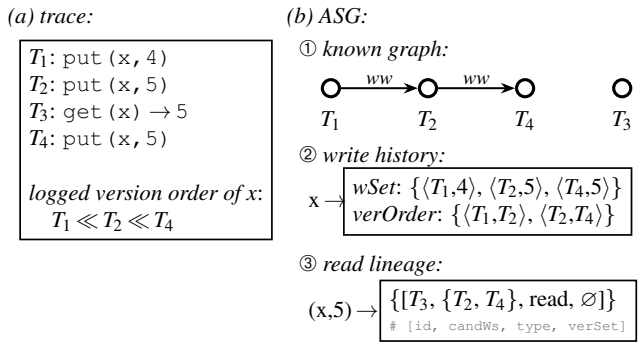


Figure 3. The running example and its Abstract Semantic Graph.
(a) trace: four transactions access key x; T_2 and T_4 write the same value 5, and the version order of x is logged as a hint (begins and commits are omitted for simplicity).
(b) ASG: the known graph has one node per transaction and the ww edges given by the version order. The write history of x records the three writes ("*wSet*") and their version order ("*verOrder*"). The read lineage maps the returned pair (x,5) to T_3's read, whose candidate writers ("*candWs*") are T_2 and T_4; the operation type is "read", so the version set ("*verSet*") is empty. For black-box database traces, some information (e.g., "*verOrder*") may be missing.

4.2 Parsing: building the ASG

With a given trace, BOOMSLANG constructs an *Abstract Semantic Graph (ASG)*: an intermediate structure that enables further lowering. A crucial design goal is to ensure that ASG is both general—capturing all necessary information across different isolation levels—and efficient—allowing quick retrieval of relevant information during the generation of low-level IR.

Observations. To design ASG, we have two key observations. First, isolation levels can be effectively expressed through graph representations as highlighted by prior work [30, 31, 36, 70, 79, 86], suggesting that graphs serve as a natural intermediate representation. This graphical formulation allows us to capture the complex relationships between transactions in an intuitive manner. Second, we have found that all dependencies in different isolation levels can be expressed as a set of conditioned read/write primitives. At their core, the basic units of transaction operations are simply reads and writes, but these are grouped by a potentially complex set of conditions that determine their valid execution given different traced information.

ASG represents a new design that generalizes to different checking problems, in contrast to prior checkers, which target a specific isolation level, or have workload assumptions, and therefore do not require generalization. In particular, unlike previous fixed graph structures (e.g., serialization graphs [30], polygraphs [75], and BC-graphs [86]), ASG's flexible design adapts nodes and edges based on the target isolation level. ASG also generalizes across all configurations, supporting white-, gray-, and black-box traces; range and point queries; logical or physical deletions; and unique or duplicated values.

Running example. We use the trace in Figure 3(a) as a running example from parsing (§4.2) through constraint generation (§6), checking it against serializability. Four transactions access key x: T_1 writes 4, T_2 and T_4 both write 5, and T_3 reads x and observes 5. Because two writes produce the same value, the trace does not reveal which of them T_3 read. For this example, we also assume that the trace contains the version order of x ($T_1 \ll T_2 \ll T_4$), for instance, extracted from CDC logs as additional information (§4.1); §5.2 discusses what changes when this order is also unknown.

ASG design. Based on these observations, we define our Abstract Semantic Graph (ASG) as consisting of three essential components: ①, a *known graph*; ②, a *write history*; and ③, a *read lineage*. Figure 3(b) depicts the ASG of our running example. The known graph captures known relationships between transactions, while the write history and read lineage store detailed information about write/read primitives.

Arrow labels denote dependency types [30]: $T_i \xrightarrow{ww} T_j$ is a write-dependency, where T_j writes a later version of the same key; $T_i \xrightarrow{wr} T_j$ is a read-dependency, where T_j reads a version written by T_i; and $T_i \xrightarrow{rw} T_j$ is an anti-dependency, where T_j writes a later version than the one read by T_i. The label $\xrightarrow{pwr}$ denotes a predicate-read-dependency, from a writer to a predicate read. When present, a key in parentheses, as in $\xrightarrow{\text{ww(x)}}$, identifies the key involved.

The write history groups updates by key:

$$Map\{Key \rightarrow \{wSet : Set\langle id, val\rangle, \\ verOrder : Set\langle id, id\rangle\}\}.$$

Here, *wSet* records writes and deletes with their transaction IDs and values, while *verOrder* records any known version order between writes to the same key. The version order may be only partially known.

The read lineage maps each returned key-value pair to its retrieval operations:

$$Map\{(Key, Val) \rightarrow Set\langle[r_id, candWs, type, verSet]\rangle\}.$$

Each entry records the reader's transaction ID (*r_id*), candidate writes (*candWs*), operation type (*type*: read, range query, or iterator), and, when available, the version set (*verSet*) for predicate queries.

To construct the ASG, BOOMSLANG scans committed transactions and creates graph nodes according to the target isolation level. It collects writes and deletes in the write history, then builds the read lineage by matching returned keys and values against candidate writes. Dependencies determined from the trace are added as edges to the known graph; ambiguous read origins remain as candidate sets for semantic analysis.

For the running example (Figure 3(b)), BOOMSLANG creates one node for each of T_1–T_4. The write history of x records $\langle T_1, 4\rangle$, $\langle T_2, 5\rangle$, and $\langle T_4, 5\rangle$ together with the logged version order, from which the parser adds the known edges $T_1 \xrightarrow{ww} T_2$ and $T_2 \xrightarrow{ww} T_4$. The read lineage maps the returned pair (x,5) to T_3's read, whose candidate writers are $\{T_2, T_4\}$. Since the trace cannot distinguish the two candidates, the parser adds no read dependency for T_3 and leaves the ambiguity to semantic analysis (§5). Had each value been unique, the candidate set would contain only one transaction, for example, T_2.

5 Semantic analysis with superpositions

The semantic analysis phase lowers the ASG to BOOMSLANG's low-level IR, which consists of simple directed graphs, first-order logic expressions, and superpositions—a new abstraction. BOOMSLANG introduces superpositions to represent heterogeneous uncertainty in a shared IR that different semantic modules can generate.

5.1 Challenges of representing heterogeneous uncertainty

Extending isolation checking to different operations and workload assumptions requires a shared representation of "known unknowns": dependencies between transactions that must exist but cannot be fully identified from the trace. Two common cases illustrate the issue:

1. *Phantom ambiguity*: if `scan(x,z)` does not return `y` ∈ `[x,z)`, it is unclear whether `y` was never inserted or deleted, especially when the trace contains multiple `put(y,...)` and `del(y)` operations.
2. *Read lineage uncertainty*: with duplicate writes, e.g., T_2 and T_4 both executing `put(x,5)` in the running example (Figure 3(a)), a later read of $x \rightarrow 5$ does not reveal which write it observes.

These are just two examples. Such uncertainties can also interact, as in traces combining range queries, physical deletions, and duplicate writes. Representing them in a common IR allows modules for new semantics to reuse the same optimizations and constraint-generation back-end.

5.2 Abstracting unknowns using superposition

We introduce *superposition*, a unified representation that abstracts unknowns. Building on established reasoning about unknown orders and dependencies [30, 35, 44, 70], BOOMSLANG makes heterogeneous uncertainty a first-class IR abstraction. We borrow the term from quantum mechanics; superposition indicates a situation where two or more states are added together (superposed). However, when measured, the superposition commits to a single state.

Superposition is defined as follows:

$$\langle possibility_1,\ possibility_2,\ \cdots,\ possibility_n\rangle$$

It is a tuple of *n possibilities*. Each possibility indicates one way to explain the observed events. In reality, one and only one possibility is true, but due to unknowns, it is unclear which one is the ground truth.

For example, if a black-box database is serializable, two concurrent transactions (say T_i and T_j) writing to the same key must be ordered. However, due to the black-box setup, the order of the two writes is unknown, leading to a superposition:

$$\langle T_i \xrightarrow{ww} T_j,\ T_j \xrightarrow{ww} T_i\rangle,$$

reading as, either T_i commits before T_j or vice versa.

A missing key in a range query can create another superposition. Consider a key x that falls within the range queried by T_q and appears elsewhere in the trace but is not returned. The key must either have been absent initially or have been deleted before the query. If T_0 represents an initially absent x and $T_{d_1}, \ldots, T_{d_m}$ are candidate transactions that delete x, the alternatives are:

$$\langle T_0 \xrightarrow{pwr} T_q,\ T_{d_1} \xrightarrow{pwr} T_q,\ \ldots,\ T_{d_m} \xrightarrow{pwr} T_q\rangle.$$

If x existed initially, the T_0 alternative is omitted.

Running example, continued. We use the running example to illustrate why unknowns are challenging and how superposition abstracts unknowns. Figure 4 varies how much the trace reveals about it: whether the version order (*VO*) is available, and whether written values encode their writer IDs using subscripts (e.g., 5_{T2}). The running example as parsed in §4.2 corresponds to Figure 4(c).

We start with no unknowns (Figure 4(a)). The trace is serializable, because there exists a serial execution of transactions: $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_4$. The dependencies between writes ($T_1 \xrightarrow{ww} T_2$ and $T_2 \xrightarrow{ww} T_4$) come from version order, which requires the white-box setup, while $T_2 \xrightarrow{wr} T_3$ follows from

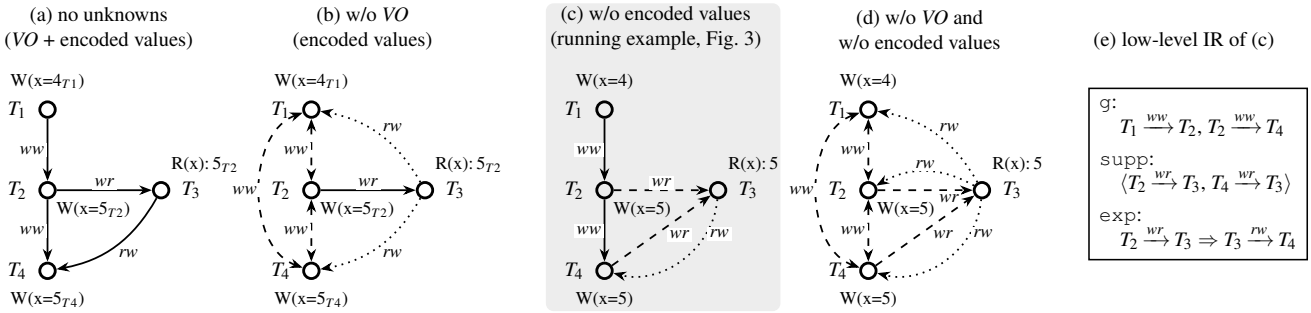


Figure 4. Unknowns and superpositions for the running example (Figure 3(a)) under different trace assumptions for checking serializability. Nodes are transactions; solid edges are known dependencies, dashed edges are possible dependencies from superpositions, and dotted edges are inferred anti-dependencies. **(a)** Both the version order (*VO*) and encoded values (e.g., 5_{T2}) are available, so nothing is unknown. **(b)** excludes the version order; **(c)**, the running example, excludes the encoded values; and **(d)** excludes both. **(e)** is the low-level IR that semantic analysis generates for (c) (§5.3).

the encoded value 5_{T2} read by T_3. Finally, T_3 must be executed before T_4 because otherwise the T_3 should have read T_4 instead of T_2. Such anti-dependencies are considered in serializability and snapshot isolation.

Next, we remove the white-box setup, so the internal scheduling information version order is unknown. Thus, although T_1, T_2, and T_4 must be totally ordered as writes to x, their exact order is unknown. Thus, the unknown write order yields pairwise superposition: $\langle T_1 \to T_2,\ T_2 \to T_1\rangle$, $\langle T_1 \to T_4,\ T_4 \to T_1\rangle$, and $\langle T_2 \to T_4,\ T_4 \to T_2\rangle$, shown as double-arrowed dashed edges in Figure 4(b). They also imply two possible anti-dependencies, shown as dotted edges.

If we restore the white-box setup but remove the encoded values, reading 5 cannot distinguish whether T_3 reads from T_2 or T_4, since both write $x = 5$. This yields superposition $\langle T_2 \to T_3,\ T_4 \to T_3\rangle$, shown as dashed edges in Figure 4(c); the dotted edge denotes the possible anti-dependency. This is exactly the running example: the candidate set $\{T_2, T_4\}$ in the ASG's read lineage (Figure 3(b)) becomes this superposition.

Finally, in a black-box setup without encoded values, all the above superpositions coexist and, more importantly, compound. For example, the dotted edge $T_3 \to T_2$ appears only if T_3 reads from T_4 and T_4 precedes T_2 on x. Although neither is true in the ground truth, both remain possible under the combined unknowns when version order and read dependencies are not available.

5.3 Semantic analysis: generating low-level IR

Intermediate representation (IR). In the semantic analysis phase, BOOMSLANG transforms the ASG into low-level IR. Crucially, lowering the ASG into the IR generates superpositions, which logically represent the set of possible DSGs [30, 31] that would explain the trace. Figure 5 depicts this transformation. BOOMSLANG's low-level IR consists of three components: a simple graph (`g` in Figure 5), logical expressions (`exp`) across edges—treated as boolean variables where true indicates existence and false indicates absence—and superpositions (`supp`) that capture unknowns in the trace.

```
1  global ASG # read only
2
3  def genIR():
4    g,supp,exp = ASG.g,{},{}  # init low-level IR
5    g,supp,exp = genReadDep(g, supp, exp)
6    ... # the other dependency passes
7
8    # user-defined modules
9    userModules = [genSessionDep, ...]
10   # the "ctx" is from envelope tracing interface
11   for f in userModules:
12     g,supp,exp = f(g, supp, exp, ctx)
13
14   return g,supp,exp
15
16 # an example of dependency pass
17 def genReadDep(g, supp, exp):
18   lineage = ASG.readLineage
19   for (k,v) in lineage:
20     for entry in lineage[(k,v)]:
21       if entry.candW.size() == 1:
22         # if write is unique, add a known edge
23         g.Edges.add(...)
24       else:
25         # if multiple possibilities, add a superposition
26         supp.add(...) # e.g., <T2->T3, T4->T3>
27   return g, supp, exp
28
29 # an example of user-defined module
30 def genSessionDep(g, supp, exp, ctx):
31   for (id1, id2) in ctx['session']: # session order
32     g.Edges.add(id1,id2)            # enforce this order
33   return g, supp, exp
```

Figure 5. Transforming ASG to low-level IR. This describes the logical transformation flow. In practice, the implementation combines common dependency passes to improve efficiency.

The IR is also the composition point for encoding isolation definitions. Rather than implementing an isolation level as a monolithic checker, BOOMSLANG builds it by composing dependency-generating modules over this IR and then applying the isolation-specific graph acyclicity condition. Each module contributes one class of facts or unknowns, such as read-dependencies, write-dependencies, or anti-dependencies. For serializability, BOOMSLANG composes the modules that construct the full DSG and checks acyclicity. For read committed, BOOMSLANG reuses the common graph structure but excludes anti-dependency modules. For SI, BOOMSLANG reuses the relevant dependency modules and lowers transaction to a begin-commit granularity. New semantics can therefore be

encoded by reusing existing modules, adding a module for a new dependency type or hint, and changing the final graph acyclicity condition, without changing trace parsing or the SMT backend.

To construct this IR, BOOMSLANG processes the ASG through multiple passes, generating each required dependency type one at a time. Each dependency-generating pass analyzes the ASG to determine whether a dependency exists between two nodes. Crucially, isolation levels define the conditions under which dependencies should exist, but the trace might not offer enough information to definitively establish these conditions. For example, when generating read dependencies in `genReadDep(...)`, if there is only one possible write that could have produced a value, we add a known edge to the graph (line 22, Figure 5). However, when multiple possibilities exist, we create a superposition to represent this uncertainty (line 25, Figure 5). These superpositions, along with logical expressions for anti-dependencies, form the core of our low-level IR's ability to represent complex dependencies and unknowns. For example, suppose T_w writes key x, T_r reads that write, and T_p later overwrites it. The IR records the conditional anti-dependency as:

$$(T_w \xrightarrow{\text{ww(x)}} T_p \wedge T_w \xrightarrow{\text{wr(x)}} T_r) \Rightarrow T_r \xrightarrow{\text{rw(x)}} T_p. \qquad (1)$$

The anti-dependency holds when both the write order and read-from relationship hold.

In the running example, `genReadDep` finds two candidate writers for T_3's read and therefore adds the superposition $\langle T_2 \xrightarrow{wr} T_3,\ T_4 \xrightarrow{wr} T_3\rangle$ (line 25, Figure 5). The anti-dependency pass then instantiates Eq. 1 with T_w=T_2, T_r=T_3, and T_p=T_4; because $T_2 \xrightarrow{ww} T_4$ is already known, the expression reduces to $T_2 \xrightarrow{wr} T_3 \Rightarrow T_3 \xrightarrow{rw} T_4$. If T_3 read from T_4 instead, no anti-dependency arises, since no transaction overwrites T_4's write. Figure 4(e) shows the resulting IR, which represents two possible DSGs, one for each read-from choice.

User-defined modules. BOOMSLANG's modular design enables extension through user-defined modules. Users will write a module function (described in the figure below) whose inputs are the low-level IR after common passes and the hints logged in envelope tracing. User modules also have access to

```
global ASG # read only

def f(g, supp, exp, hints):
  ... # user can access ASG and update g, supp, exp
  return g, supp, exp

registerUserModule(f)  # add f to the list userModules
```

the read-only ASG. Within the module, users can manipulate the low-level IR to describe their own semantics. Figure 5 gives a simple module example supporting session ordering dependencies, `genSessionDep(...)`. User-defined modules and any logged hints they use to constrain the IR are trusted inputs. BOOMSLANG's correctness guarantees depend on their validity. This extensibility ensures that BOOMSLANG can adapt to different isolation levels and incorporate various sources of information while maintaining a consistent overall approach.

6 Optimization and constraint generation

As a unified framework, BOOMSLANG enables sharing optimizations across different encodings and checkers. Below are three optimizations that significantly accelerate solving.

Topological prioritization. Each superposition contains multiple possibilities, and solving involves searching through these possibilities. However, off-the-shelf solving treats all possibilities equally, searching them in a random order. Topological prioritization addresses this inefficiency by prioritizing possibilities that are more likely to be true within the superposition. In particular, we topologically sort the existing graph and use the resulting node order as the priority. For example, consider a superposition $\langle T_i \to T_j,\ T_j \to T_i\rangle$. If T_i is topologically ordered before T_j, the checker then prioritizes exploring $T_i \to T_j$ over the other possibility. When known dependencies already constrain much of the order, this prioritization can find a satisfying assignment without exploring many alternatives. It changes only the search order, without excluding any feasible solution.

Reachability pruning. Node reachability serves as a powerful tool to eliminate superpositions [75, §3.3]. For example, if a path $T_i \rightsquigarrow T_j$ exists in the graph and this path together with the candidate edge $T_j \to T_i$ would form a cycle forbidden by the target isolation level, then $T_j \to T_i$ can be pruned from the superposition $\langle T_i \to T_j,\ T_j \to T_i\rangle$. The superposition then reduces to the known edge $T_i \to T_j$. Pruning this forbidden alternative cannot exclude a valid execution. We precompute the graph's transitive closure to accelerate reachability checks and reduce the search space before SMT solving. Traces with many unresolved, interacting alternatives remain difficult.

Improving rejection performance with unsat search. Proving unsatisfiability is challenging for today's checkers. For example, Viper significantly underperforms with unsat traces, occasionally even timing out [85, Figure 15]. The fundamental reason is that rejecting a non-trivial unsat trace requires proving that no set of possibilities satisfies specifications.

Though challenging, proving unsatisfiability for real-world database executions exhibits some characteristics. We observe that unsat traces often have small "unsat cores": a small set of transactions that already violate the specifications. This matches Cui et al.'s study of 140 real-world transaction bugs, where 93.6% involved at most three transactions and all involved at most five [45, Finding 4]. We hypothesize that violations usually arise from corner cases under high concurrency, so the faulty transactions tend to occur close together in time.

Leveraging this observation, unsat search partitions a trace into smaller sub-traces and checks them individually. If a sub-trace retains a small unsat core, the checker can prove

the violation without resolving unrelated ambiguities in the full trace. This can substantially reduce rejection time, but depends on keeping the bug-inducing transactions together.

The guarantee of unsat search is that if it rejects, then the original trace is unsat. This is because the unsat core identified during the unsat search is also a valid unsat core for the original trace. However, failing to find an unsat core does not necessarily imply that the original trace satisfies the specifications, as unsat cores may span across sub-traces. In this case, it therefore still requires solving the complete encoding. Here, our earlier observation becomes critical: if unsat cores are typically small, the likelihood of all these cores being split across different sub-traces is low. Our current implementation assigns graph nodes to subgraphs at random, then checks each induced subgraph. It does not group nodes using concurrency information. Designing and evaluating deterministic or learned grouping heuristics is future work.

Constraint generation. After optimizations, BOOMSLANG emits constraints based on the low-level IR. It creates a Boolean variable for each edge: *True* represents the existence of an edge; *False* means otherwise. For edges in the IR's graph `g`, BOOMSLANG assigns them to be *True*. For logical expressions `exp` like implications (e.g., Eq. 1), BOOMSLANG emits corresponding SMT formulas directly. For each superposition in the set `supp`, say $\langle e_1, e_2, \ldots, e_n \rangle$, BOOMSLANG ensures that one and only one of the possible expressions is *True*, by $\bigvee_{i=1}^{n} \left(e_i \wedge \bigwedge_{j \neq i} \neg e_j \right)$. Finally, BOOMSLANG adds isolation correctness constraints—typically graph acyclicity—based on the isolation level definition being verified. For the running example (Figure 4(e)), BOOMSLANG sets $T_1 \xrightarrow{ww} T_2$ and $T_2 \xrightarrow{ww} T_4$ to *True*, requires exactly one of $T_2 \xrightarrow{wr} T_3$ and $T_4 \xrightarrow{wr} T_3$ to be *True*, emits $T_2 \xrightarrow{wr} T_3 \Rightarrow T_3 \xrightarrow{rw} T_4$, and requires the resulting graph to be acyclic. Either choice yields an acyclic graph (e.g., $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_4$ when T_3 reads from T_2), so the solver returns `sat` and BOOMSLANG accepts the trace.

Why BOOMSLANG works well in practice. Checking black-box traces is NP-complete in general [38, 39, 70], and BOOMSLANG does not change the worst case. However, practical traces are far from adversarial: read values restrict candidate writers to matching writes, and known dependencies and session order constrain transaction order. For correct traces, dependencies are structured by the database's concurrency control protocol: for example, although serializability permits a much later transaction to precede an earlier one in the serialization order, such large inversions are rare. Reachability pruning and topological prioritization exploit this regularity. For buggy traces, a violation usually involves only a few concurrent or nearby transactions, so a small unsatisfiable portion of the trace suffices to reject it; unsat search looks for such portions. These optimizations preserve soundness and completeness, but pathological or adversarial traces can still cause BOOMSLANG to time out.

7 Implementation

We implement BOOMSLANG in Java, consisting of approximately 14K lines of code (LOC). This includes around 0.6K LOC of building the ASG, 1.7K LOC of converting ASG to low-level IR, 1.1K LOC of optimizations, 1.9K LOC of translating IR to SMT clauses, and other utilities.

8 Experimental evaluation

In this section, we evaluate BOOMSLANG along three dimensions: (1) its extensibility, demonstrated through support for diverse configurations, checker implementations, semantic customizations, and module reuse; (2) its effectiveness in practical use cases; (3) its performance. The evaluated artifact is publicly available.[1]

Setup. Unless otherwise noted, all experiments are run on a 12-core AMD Ryzen 5900 machine, with 64GB of memory and Ubuntu 20.04. The checker uses at most 40GB of memory, and we run 24 concurrent clients by default.

8.1 BOOMSLANG supports all existing configurations

Configuration space. We evaluate BOOMSLANG across all known configurations, in terms of isolation levels, benchmarks, and databases.

Isolation levels. We experiment with six well known isolation levels: strict serializability (*SSER*), serializability (*SER*), snapshot isolation (*SI*), repeatable read (*RR*), read committed (*RC*), and *PL-2+* defined by Adya [30, §4.1].

Benchmarks. We experiment with nine existing benchmarks from prior work [75]: RUBiS [20], Twitter [2], TPC-C [24], BlindW-C [3, 4] from Cobra [75]; ListAppend-J from Jepsen; BlindW-T [84] (to distinguish from Cobra's BlindW-C benchmark), Retwis [84] from Tapir [84], BlindW-E from Port [64]; and the built-in filesystem performance testing benchmark from JuiceFS [12].

We also reimplement BlindW (to distinguish from BlindW-C and BlindW-E). Further, we implement two additional benchmarks focusing on range queries, iterators, and physical deletions, without the unique value assumption.

- *RandomBench*: this benchmark includes all the operations that we support, including reads, writes, range queries, true deletions, and iterators. No prior black-box checkers support this benchmark. Written values are non-unique, and each transaction has 8 operations.
- *RandomRange*: this benchmark evaluates range queries. Each transaction has 8 operations, with 20% range queries and 80% other operations (reads, writes, deletions, and read-modify-writes).

[1] https://doi.org/10.5281/zenodo.23203856

Iso.	Database	#Txns	App	Runtime
SSER	Tapir [84]	13.2K	BlindW-T	1.14s
		9.1K	Retwis	0.63s
SER	CockroachDB 25.1.4 [74]	10.5K	BlindW-C	4.45s
		17.3K	TPC-C	18.54s
SER	PostgreSQL 15.2 [19]	9.8K	RUBiS	2.81s
		10.1K	Twitter	2.43s
SER	YugabyteDB 2.25.1.0 [27]	10.4K	BlindW-C	4.4s
		15.3K	TPC-C	10.65s
SI	TiKV 7.6.0 [23]	5.8K	JuiceFS 1.2.3	10.03s
		10K	RandomBench	370.81s
SI	YugabyteDB 2.25.1.0 [27]	10.3K	RUBiS	4.08s
		10.4K	Twitter	4.42s
SI	TiDB 8.5.4 [22]	10K	RandomRange	16.76s
RR	MariaDB 12.1.2 [14]	10K	BlindW	3.25s
RR	MySQL 9.5.0 [15]	10K	BlindW	2.47s
PL-2+	Port [64]	0.8K*	BlindW-E	0.2s
RC	PostgreSQL 15.2 [19]	10.1K	BlindW-C	2.99s
		14.6K	TPC-C	5.94s
RC	MariaDB 12.1.2 [14]	10K	BlindW	5.33s
RC	MySQL 9.5.0 [15]	10K	BlindW	5.63s

Figure 6. Boomslang can check various combinations of isolation levels, databases, and workloads.
* Checking PL-2+ is challenging, and Boomslang does not scale to 10K transactions as it does for other isolation levels.

Checker	Gen. IR	Pruning	Gen. SMT	Total LOC
Boomslang SSER	55	10†	180$^+$	245
Boomslang SER	51*	10†	180$^+$	241
Boomslang RR	59	10†	180$^+$	249
Boomslang SI	95	10†	180$^+$	285
Boomslang RC	48	10†	180$^+$	238
Boomslang PL-2+	51*	36$^\bullet$	178	265
Boomslang CS	51*	36$^\bullet$	277	364
Boomslang PL-FCV	51*	36$^\bullet$	226	313

Figure 7. Lines of code (LOC) for each checker component: IR generation (Gen. IR), pruning, and SMT encoding.
†,*,$^\bullet$,$^+$ means this component is shared across checkers. For example, Boomslang SER/PL-2+/CS share the same component of converting ASG to low-level IRs.

Databases. We experiment with nine databases: PostgreSQL [19], YugabyteDB [27], CockroachDB [74], TiKV [23], Tapir [84], Port [64], MariaDB [14], MySQL [15], and TiDB [22].

Experimental results. We evaluate the generalizability of Boomslang by applying it to check all mentioned isolation levels across a diverse set of benchmarks and databases. Figure 6 shows a subset of the results. Note that we present only a single trace for each configuration for demonstration purposes, not for performance evaluation. RandomRange and RandomBench can only be checked by Boomslang since they have operations that existing checkers cannot handle. We only present the result of Port on a relatively small trace. Even though both PL-2+ and Port are related to the notion of transactional causal consistency, it is unclear whether Port provides PL-2+, but its traces pass our checking. The benchmarks are all run as-is, without any modifications.

Checker	Original LOC	LOC in Boomslang
Cobra [75]	16.6K	271
Viper [85]	2.9K	281
PolySI [53]	3.4K	798

Figure 8. Lines of code (LOC) for Cobra/Viper/PolySI's original system and their implementations in Boomslang.

8.2 Implementing new checkers made easy

Boomslang is an extensible framework. To demonstrate this, we implement eight different checkers and measure their lines of code (LOC) for different phases. Figure 7 shows the results.

The eight checkers cover strict serializability, serializability, snapshot isolation, repeatable read [30, 34], read committed [30, Section 3.2.2], PL-2+ [30, Section 4.1], cursor stability [30, Section 4.6], and PL-FCV [30, Section 4.6]. Prior systems already support some of them under specific workload assumptions. For example, Plume [61] checks read committed assuming unique written values for each key. Elle's list-append analysis [56] uses uniquely appended elements to infer version order and read-from relationships and achieve soundness. Our new encodings lift all those assumptions and support range queries, iterators, and physical deletions. Figure 7 shows that all the checkers are implemented with <370 lines of code. Boomslang makes module reuse easy: four checkers—Boomslang SER, PL-2+, CS and PL-FCV—share the same conversion code from ASG to IR, because their IR needs the same types of superpositions and first-order expressions. Another five checkers (Boomslang SSER, SER, RR, SI, RC) share the same pruning strategies because they all require no-cycle property in a graph with all dependencies.

8.3 Replicating existing checkers

To further illustrate Boomslang's flexibility, we reimplement three existing checkers: Cobra, Viper and PolySI. Figure 8 shows the results. We also include the LOC of their original implementations. The original implementations require approximately 4.3–61 times as many lines of code as their counterparts in Boomslang.

8.4 Case Study

Case #1: discovering a new bug in TiDB. While running benchmarks on TiDB 8.5.4, Boomslang detected an isolation-level violation in pessimistic mode under repeatable read. After another transaction commits an update, a transaction that writes the same value to the same column of the same row may subsequently read its old snapshot value instead of its own write. We reported the issue [16], and the TiDB developers acknowledged it as a moderate-severity bug.

Case #2: catching bugs in PostgreSQL 12.3 and MariaDB 10.6. Boomslang can serve as a test oracle for concurrency testing. We evaluate this by running our benchmarks on two known-buggy databases, PostgreSQL 12.3 and MariaDB 10.6, and using Boomslang as the oracle. Boomslang reliably

catches bugs within tens of runs. For PostgreSQL, it identifies the expected G2-item anomaly and additionally detects a general G2 cycle involving range queries under serializability, which existing checkers cannot capture. On MariaDB, it rediscovers a repeatable-read anomaly [26] where a read after an update observes the transaction's initial snapshot; this bug requires non-unique values, another case unsupported by most existing checkers.

Case #3: auditing file system metadata operations. JuiceFS uses TiKV as its metadata storage, making metadata integrity dependent on TiKV's transactional guarantees. The metadata integrity is critical to file system correctness, and developers can use Boomslang to audit if the underlying metadata storage actually provides the claimed guarantees. Notably, JuiceFS uses the full APIs—`Get`/`Set`/`Iter`/`Delete`—beyond the support of existing checkers. Figure 6 reports one such result for JuiceFS 1.2.3.

Case #4: customizing checkers for vendor-specific behavior. MariaDB's repeatable read mixes MVCC snapshot reads with locking reads/writes (e.g., `SELECT FOR UPDATE`) that observe the latest committed state, so its behavior does not match any standard isolation level and cannot be modeled by existing checkers. With Boomslang, we support this semantics by partitioning each physical transaction into logical sub-transactions: locking reads and writes start new sub-transactions with fresh views, while snapshot reads remain in earlier ones. This modification requires only 85 LOC changes in front-end, where Boomslang parses and structures transactions accordingly. With this customization, Boomslang accepts traces permitted by MariaDB's repeatable-read semantics but rejected under the standard definition. Note that this partitioning strategy is not a tight checker, as it intentionally relaxes certain intra-transaction constraints.

Case #5: supporting mixed isolation guarantees. Beyond standard isolation levels, databases often provide additional guarantees such as read-your-own-write (RYOW). In our experiments, we observe that databases differ in whether and how they support RYOW within a transaction. Boomslang accommodates these variations by implementing three RYOW policies within 38 LOC changes: must read your own writes, must read externally, and may read from either. These policies can be inferred automatically or specified by the user, enabling flexible support for mixed isolation guarantees.

Case #6: confirming a theoretical hypothesis: timestamp inversion pitfall. Lu et al. [62] identify a potential issue in many timestamp-based transaction protocols, named *timestamp inversion pitfall*. They provided a counterexample in the appendix of their paper's extended version [63]. Using Boomslang, we successfully confirm this phenomenon in practice: we found a trace from Tapir that is serializable but not strictly serializable, demonstrating a real instance of the timestamp inversion pitfall.

Case #7: enabling fair comparison across checker encodings. Multiple encodings have been proposed for checking the same isolation level, such as snapshot isolation. We reimplement these encodings in Boomslang to enable a fair performance comparison, removing differences due to implementation-specific factors. Figure 9 compares two encodings—originally from Viper and PolySI—on the same traces: the BlindW-C benchmark with 10K to 40K transactions. We ran this experiment on a 12th Gen Intel Core i5-12400 processor. Viper's original implementation is slower than PolySI's due to its Python implementation and PolySI's specialized optimizations. However, when comparing only the encodings, Viper's encoding outperforms PolySI's—an insight made possible only by a unified framework like Boomslang.

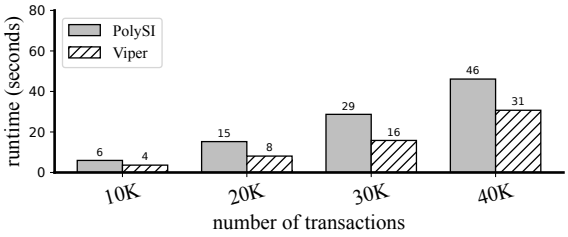


Figure 9. Checking SI: a fair comparison between PolySI's and Viper's encodings

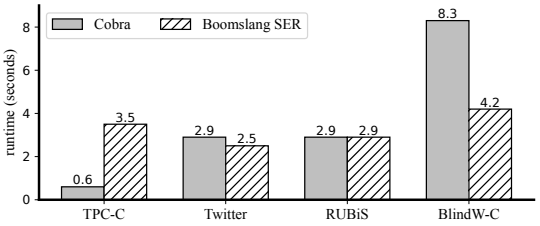


Figure 10. Checking serializability: Boomslang and Cobra.

8.5 Performance evaluation

Baselines. We compare Boomslang with four state-of-the-art checkers: Cobra [4], Viper [25, 85], PolySI [18, 53], and Elle [56]. They are the most closely related work. The first three work in a black-box setup and assume unique value: Cobra targets serializability, while Viper and PolySI target SI; Viper also supports range queries with tombstone. Elle is a grey-box checker that supports multiple isolation levels; its ListAppend mode exposes version order for sound checking.

We compare Boomslang's performance with baselines using CobraBench [3]. Benchmarks are executed on PostgreSQL with session orders, as required by the baselines.

Checking serializability: Boomslang versus Cobra. We compare Boomslang with Cobra on the problem of checking serializability. We run CobraBench with 10K transactions for each benchmark. Cobra relies on GPU acceleration, and thus its scalability is fundamentally constrained by GPU memory. On our 3060 Ti GPU, Cobra cannot handle 20K transactions and hence 10K transactions are used for the comparison.

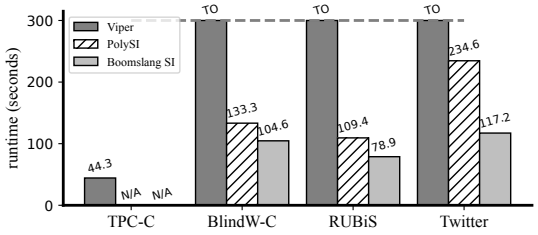


Figure 11. Checking SI: BOOMSLANG, Viper, and PolySI.

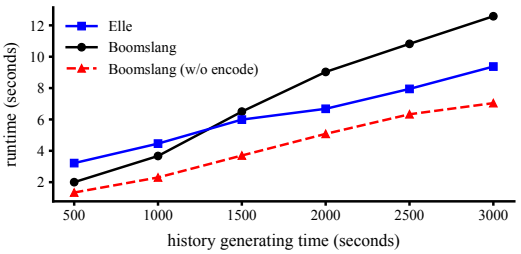


Figure 12. Checking serializability: BOOMSLANG and Elle.

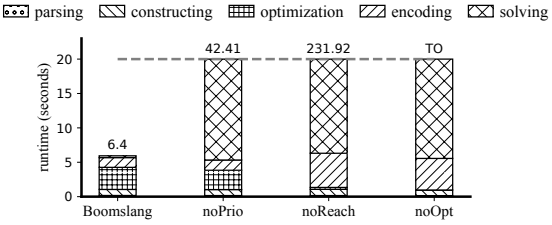


Figure 13. Decomposition of BOOMSLANG runtime of BlindW-C (2K keys, 32 threads, 10K txns) under different settings.

Figure 10 shows the results. Compared with Cobra, BOOMSLANG achieves comparable or better results, except for TPC-C. BOOMSLANG has additional topological prioritization that Cobra does not have, which can guide the process of assigning values to edge variables, and hence BOOMSLANG is faster than Cobra on all the traces that have superpositions. This comparison uses CobraBench's TPC-C benchmark, which encodes unique values to satisfy Cobra's assumptions, making read-from relationships directly identifiable. Moreover, all writes are read-modify-writes, so these read-from relationships also determine the write order for each key. Together, these properties leave no ambiguity in read-from relationships or write orders, and hence no superpositions in these traces. BOOMSLANG consequently pays the bookkeeping overhead of its generalized IR without benefiting from its ambiguity-handling optimizations, explaining the slowdown relative to Cobra.

Checking SI: BOOMSLANG versus Viper and PolySI. Figure 11 compares the runtime performance of BOOMSLANG against Viper and PolySI across four workloads with 50K transactions. Viper reaches the 300-second timeout (TO) on BlindW-C, RUBiS, and Twitter, indicating Viper fails to scale computationally for large workloads with many superpositions. On these workloads, BOOMSLANG completes the checks faster than PolySI, including a 2x speedup on Twitter (117.2s vs. 234.6s) and a reduction from 109.4s to 78.9s on RUBiS. Both BOOMSLANG and PolySI run out of memory on the 50K-transaction TPC-C trace, whereas Viper completes the check. Repeated updates to the same keys in TPC-C produce long version chains. Although read-modify-writes reveal their order, BOOMSLANG's modular dependency generation relies on write-dependencies between nonconsecutive versions to derive anti-dependencies. Explicitly materializing these transitive write-dependencies and anti-dependencies exhausts memory. Viper instead retains only consecutive version-order edges and completes the check. PolySI's generalized constraints also encode transitive write-dependencies, exhausting memory during pruning.

Checking serializability: BOOMSLANG versus Elle. In this experiment, we run Jepsen's ListAppend-J benchmark and vary the history collection time, which is approximately proportional to the number of transactions. We compare the checking time of BOOMSLANG with Elle, used out of the box. We adopt 24 sessions. Figure 12 shows the results. The performance is comparable, yet BOOMSLANG has a greater slope than Elle. This difference reflects BOOMSLANG's SMT encoding overhead as expected. Excluding encoding but retaining solving time, BOOMSLANG performs similarly to Elle.

Ablation study. We measure how pipeline phases and optimizations contribute to runtime. We evaluate BOOMSLANG with BlindW-C benchmark on PostgreSQL under serializability. We vary the configurations of serializability checkers with various optimizations: (a) *BOOMSLANG*: BOOMSLANG with both optimizations; (b) *noPrio*: BOOMSLANG serializability checker without topological prioritization; (c) *noReach*: BOOMSLANG serializability checker (§6) without reachability pruning. (d) *noOpt*: BOOMSLANG serializability checker with no optimizations. We measure the runtime of BOOMSLANG's different phases: parsing, constructing, optimization, encoding, and solving. Figure 13 shows the results. BOOMSLANG times out (>600s) without optimizations. With topological prioritization, BOOMSLANG (the noReach bar) becomes solvable because topological prioritization has a better guided search in the solving process. With reachability pruning, BOOMSLANG (the noPrio bar) is faster than the noReach bar because many superpositions have been pruned.

Runtime vs. #superpositions. In this experiment, we study the relationship between checking time and the number of superpositions. Figure 14 shows that checking time grows superlinearly with the number of superpositions, as expected since serializability checking is NP-complete [70].

Detecting real-world violations. To confirm BOOMSLANG's ability to catch real-world anomalies, we check 15 known violation traces from online repositories [5, 8, 17, 21, 58] and

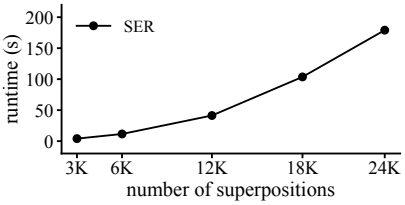


Figure 14. Checking serializability: runtime grows superlinearly with the number of superpositions.

Iso.	Anomaly	Database	#Txns	Time
SER	G2-anomaly [7]	Yugabyte† 1.3.1.0	37.2K	1.6s
	Lost updates [1]	Yugabyte 1.1.10.0	2.8K	25.55s
	G2-anomaly [6]	CockroachDB beta.	446	0.08s
	Read uncommitted [13]	CockroachDB 2.1	20⋆	0.05s
	Read skew [9]	FaunaDB 2.5.4	8.2K	14.05s
	Fractured reads [49]	Tapir	547	0.07s
	Fractured reads [49]	Tapir	480	0.06s
	G1-anomaly [30]	Tapir	13156	0.68s
	G2-anomaly [30]	Tapir	9082	0.66s
SI	Lost updates [10]	MongoDB 4.2.6	23.2K	0.28s
	Aborted read [10]	MongoDB 4.2.6	2.2K	0.08s
	G1c-anomaly [10]	MongoDB 4.2.6	1.1K	0.07s
	Read future writes [10]	MongoDB 4.2.6	4.6K	0.14s
	Read skew [11]	TiDB 2.1.7	9.1K	0.2s
	Non-repeatable read [48]	MariaDB* 10.4.22	1K	0.16s
	G-SI-anomaly [30]	Yugabyte 2.11.1	20	0.05s
SSER	Time inversion [62]	Tapir	68	0.05s
PL-2+	Thin-air read [66]	AntidoteDB 0.2.2	120	0.03s

Figure 15. Boomslang detects real-world violations. "CockroachDB beta." represents CockroachDB-beta 20160829.
⋆ The report only contains a small fragment of the original trace.
* This is short for MariaDB-Galera.
† This is short for YugabyteDB.

three new Tapir traces we collected by running a slightly modified BlindW-T benchmark on a single machine. These traces cover four isolation levels: serializability, snapshot isolation, strict serializability, and PL-2+. Figure 15 shows the results. Boomslang detects all the violations within 26s.

9 Related work

Database checkers. Prior isolation checkers make different tradeoffs among observability, workload generality, and extensibility. Jepsen [8] is a complementary testing framework that runs workloads, injects faults, and records histories for analysis with Elle [56]. Elle is a highly influential checker that uses ListAppend to expose hidden dependencies for bug finding, while some modes trade soundness for coverage. Cai et al. [40] extend Elle to workloads that write duplicate values by tracking finer-grained, operation-level dependencies. Cobra [75], Viper [85], and PolySI [53] provide sound and complete checking in their target settings using specialized graph encodings, with unique values identifying read-from relationships in read/write traces. VeriStrong [41] checks SER and SI without the unique-value assumption. Boomslang emphasizes extensibility, allowing users to implement custom modules that reproduce the functionality of above checkers.

dbcop [37, 38] supports multiple isolation levels by checking whether a read/write history with unique values admits a commit order satisfying the target model's axioms. Plume [61] uses anomaly characterizations and vector and tree clocks to efficiently check read committed, cut isolation, read atomicity, and transactional causal consistency on read/write histories with unique values. Boomslang builds on insights from these approaches and incorporates some of their optimizations.

Ouyang's checker [69] uses internal execution information to efficiently check MongoDB's SI variants. Emme [42] and its extensions [43] use version certificate recovery to obtain database-internal version orders and version sets for efficient checking of predicate operations. Chronos [57] uses database-provided timestamps for fast checking of SI and SER, and MTC [78] restricts workloads to mini-transactions so that SER and SI can be checked in near-linear or quadratic time. Users can provide the above internal information as hints to Boomslang to improve checking efficiency.

Reasoning about unknowns. Reasoning over unobserved orders and dependencies has a long history. Bernstein and Goodman [35] characterize multiversion serializability through the existence of a version order yielding an acyclic serialization graph. Papadimitriou's polygraphs [70] represent choices among alternative orderings. Adya's predicate dependencies [30] depend on the versions selected for a predicate read and whether later writes change which objects match it. Seeing is Believing [44] specifies isolation through the existence of an execution satisfying per-transaction commit tests. Boomslang builds on these foundations with a first-class IR unifying heterogeneous uncertainty, so semantic modules can reuse optimizations and constraint generation.

Testing and other consistency checking. Bug-finding systems such as TxCheck [54], Troc [47], and Leopard [59, 60, 80], as well as cloud-storage consistency testers [55, 65, 72, 76, 83], often trade soundness or completeness for performance, and may instrument user workloads to expose database states.

10 Conclusion

We presented Boomslang, an extensible framework for checking transaction isolation. Its superposition-based IR unifies heterogeneous uncertainty, supporting non-unique values, range queries, iterators, and physical deletions. We demonstrate extensibility with eight new isolation-level encodings and reimplementations of Cobra [75], Viper [85], and PolySI [53]. Module and optimization reuse reduces implementation effort and enables fair comparisons across encodings. Our evaluation demonstrates database bug finding and file-system metadata checking, with comparable or better performance on shared benchmarks. Nevertheless, guarantees require complete relevant traces and pathological traces may still time out.

11 Acknowledgement

Jian Zhang and Cheng Tan were supported in part by NSF CAREER #2237295. Mu's lab was supported in part by NSF awards CNS-2321725, CNS-2238768, and CNS-2130590.

References

[1] Acknowledged inserts can be present in reads for tens of seconds, then disappear. https://github.com/YugaByte/yugabyte-db/issues/824.

[2] Big data in real time at Twitter. https://www.infoq.com/presentations/Big-Data-in-Real-Time-at-Twitter.

[3] Cobra bench. https://github.com/DBCobra/CobraBench.

[4] Cobra verifier. https://github.com/DBCobra/CobraVerifier.

[5] Cobralogs. https://github.com/DBCobra/CobraLogs/tree/12e4421a1eb7f276d2de36608980dd3e535de64d.

[6] G2: anti-dependency cycles. https://github.com/cockroachdb/cockroach/issues/10030.

[7] G2-item anomaly with master kills. https://github.com/YugaByte/yugabyte-db/issues/2125.

[8] Jepsen. https://jepsen.io/.

[9] Jepsen: FaunaDB 2.5.4. http://jepsen.io/analyses/faunadb-2.5.4.

[10] Jepsen: MongoDB 4.2.6. https://jepsen.io/analyses/mongodb-4.2.6.

[11] Jepsen: TiDB 2.1.7. https://jepsen.io/analyses/tidb-2.1.7.

[12] JuiceFS. https://github.com/juicedata/juicefs.

[13] Lessons learned from 2+ years of nightly jepsen tests. https://www.cockroachlabs.com/blog/jepsen-tests-lessons/.

[14] MariaDB. https://mariadb.org/.

[15] MySQL. https://www.mysql.com/.

[16] Pessimistic transaction: Select returns old snapshot after update to the same value. https://github.com/pingcap/tidb/issues/64888.

[17] Plume: Efficient and complete black-box checking of weak isolation levels. https://github.com/dracooooooo/Plume.

[18] PolySI GitHub repo. https://github.com/hengxin/PolySI-PVLDB2023-Artifacts.

[19] PostgreSQL official website. https://www.postgresql.org/.

[20] RUBiS. https://rubis.ow2.org/.

[21] Tapircorrectnesstest. https://github.com/lucamul/TapirCorrectnessTest.

[22] TiDB. https://www.pingcap.com/.

[23] TiKV is a highly scalable, low latency, and easy to use key-value database. https://tikv.org/.

[24] TPC-C. http://www.tpc.org/tpcc/.

[25] Viper github repo. https://github.com/Khoury-srg/Viper.

[26] Weird select view when a record is modified to the same value by two transactions. https://jira.mariadb.org/browse/MDEV-26642.

[27] YugabyteDB GitHub repo. https://github.com/yugabyte/yugabyte-db.

[28] The Z3 theorem prover. https://github.com/z3prover/z3.

[29] Boomslang: A modular framework for database transaction isolation checking. https://doi.org/10.5281/zenodo.23203856, 2026.

[30] A. Adya. *Weak consistency: a generalized theory and optimistic implementations for distributed transactions*. PhD thesis, Massachusetts Institute of Technology, 1999.

[31] A. Adya, B. Liskov, and P. O'Neil. Generalized isolation level definitions. In *Proceedings of 16th International Conference on Data Engineering*, 2000.

[32] A. Alquraan, H. Takruri, M. Alfatafta, and S. Al-Kiswany. An analysis of network-partitioning failures in cloud systems. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 51–68, 2018.

[33] S. Bayless, N. Bayless, H. H. Hoos, and A. J. Hu. SAT modulo monotonic theories. In *Proc. AAAI*, Jan. 2015.

[34] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *Proc. SIGMOD*, May 1995.

[35] P. A. Bernstein and N. Goodman. Multiversion concurrency control—theory and algorithms. *ACM Transactions on Database Systems*, 8(4):465–483, 1983.

[36] P. A. Bernstein, D. W. Shipman, and W. S. Wong. Formal aspects of serializability in database concurrency control. *TSE*, SE-5(3), May 1979.

[37] R. Biswas and C. Enea. dbcop source code. https://gitlab.math.univ-paris-diderot.fr/ranadeep/dbcop.

[38] R. Biswas and C. Enea. On the complexity of checking transactional consistency. In *Proc. OOPSLA*, Oct. 2019.

[39] A. Bouajjani, C. Enea, R. Guerraoui, and J. Hamza. On verifying causal consistency. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 626–638, 2017.

[40] Z. Cai, S. Liu, H. Wei, and Y. Chen. Extending Elle for transaction workloads with duplicate values. arXiv:2607.17515, 2026.

[41] Z. Cai, S. Liu, H. Wei, Y. Chen, and A. Pan. Fast verification of strong database isolation. *Proc. VLDB Endow.*, 19(4):563–575, 2025.

[42] J. Clark, A. F. Donaldson, J. Wickerson, and M. Rigger. Validating database system isolation level implementations with version certificate recovery. In *Proc. EuroSys*, 2024.

[43] J. Clark, M. Rigger, J. Wickerson, and A. F. Donaldson. Bringing order to practical validation of database isolation levels. *ACM Trans. Comput. Syst.*, 44(3), 2026.

[44] N. Crooks, Y. Pu, L. Alvisi, and A. Clement. Seeing is believing: a client-centric specification of database isolation. In *Proc. PODC*, July 2017.

[45] Z. Cui, W. Dou, Y. Gao, D. Wang, J. Song, Y. Zheng, T. Wang, R. Yang, K. Xu, Y. Hu, et al. Understanding transaction bugs in database systems. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, pages 1–13. IEEE Computer Society, 2024.

[46] H. D. Dixit, S. Pendharkar, M. Beadon, C. Mason, T. Chakravarthy, B. Muthiah, and S. Sankar. Silent data corruptions at scale. *arXiv preprint arXiv:2102.11245*, 2021.

[47] W. Dou, Z. Cui, Q. Dai, J. Song, D. Wang, Y. Gao, W. Wang, J. Wei, L. Chen, H. Wang, et al. Detecting isolation bugs via transaction oracle construction. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1123–1135. IEEE, 2023.

[48] A. Eisenberg and J. Melton. SQL standardization: The next steps. *SIGMOD Rec.*, 29(1):63–67, 2000.

[49] S. Ghasemirad, S. Liu, C. Sprenger, L. Multazzu, and D. A. Basin. VerIso: Verifiable isolation guarantees for database transactions. *Proc. VLDB Endow.*, 18(5):1362–1375, 2025.

[50] W. Golab, M. R. Rahman, A. AuYoung, K. Keeton, and I. Gupta. Client-centric benchmarking of eventual consistency for cloud storage systems. In *Proc. ICDCS*, June 2014.

[51] L. Gu, S. Liu, T. Xing, H. Wei, Y. Chen, and D. Basin. IsoVista: Black-box checking database isolation guarantees. *Proc. VLDB Endow.*, 17(12):4325–4328, 2024.

[52] D. Huang, Q. Liu, Q. Cui, Z. Fang, X. Ma, F. Xu, L. Shen, L. Tang, Y. Zhou, M. Huang, et al. TiDB: a Raft-based HTAP database. *Proceedings of the VLDB Endowment*, 13(12):3072–3084, 2020.

[53] K. Huang, S. Liu, Z. Chen, H. Wei, D. Basin, H. Li, and A. Pan. Efficient black-box checking of snapshot isolation in databases. *Proceedings of the VLDB Endowment*, 16(6):1264–1276, 2023.

[54] Z.-M. Jiang, S. Liu, M. Rigger, and Z. Su. Detecting Transactional Bugs in Database Engines via Graph-Based Oracle Construction. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 397–417, 2023.

[55] B. H. Kim and D. Lie. Caelus: Verifying the consistency of cloud services with battery-powered devices. In *Proc. S&P*, May 2015.

[56] K. Kingsbury and P. Alvaro. Elle: Inferring isolation anomalies from experimental observations. *arXiv preprint arXiv:2003.10554*, 2020.

[57] H. Li, H. Wei, H. Ouyang, Y. Chen, N. Yang, R. Zhang, and A. Pan. Online timestamp-based transactional isolation checking of database systems. In *Proceedings of the 41st IEEE International Conference on Data Engineering (ICDE)*, pages 3738–3750, 2025.

[58] H. Li, H. Wei, H. Ouyang, Y. Chen, N. Yang, R. Zhang, and A. Pan. Online timestamp-based transactional isolation checking of database systems (extended version). https://github.com/FertileFragrance/TimeKiller, 2025.

[59] K. Li, S. Weng, P. Liu, L. Ni, C. Yang, R. Zhang, X. Zhou, J. Lou, G. Huang, W. Qian, et al. Leopard: A black-box approach for efficiently verifying various isolation levels. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*, pages 722–735. IEEE, 2023.

[60] P. Liu, S. Weng, K. Li, L. Ni, C. Yang, R. Zhang, W. Qian, and D. Qiao. Leopard: A general test suite for isolation level verification. In *14th Conference on Innovative Data Systems Research (CIDR)*, 2024.

[61] S. Liu, L. Gu, H. Wei, and D. A. Basin. Plume: Efficient and complete black-box checking of weak isolation levels. *Proc. ACM Program. Lang.*, 8(OOPSLA2):876–904, 2024.

[62] H. Lu, S. Mu, S. Sen, and W. Lloyd. NCC: Natural concurrency control for strictly serializable datastores by avoiding the timestamp-inversion pitfall. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, 2023.

[63] H. Lu, S. Mu, S. Sen, and W. Lloyd. NCC: Natural concurrency control for strictly serializable datastores by avoiding the timestamp-inversion pitfall (extended version). arXiv:2305.14270, https://arxiv.org/abs/2305.14270, 2023.

[64] H. Lu, S. Sen, and W. Lloyd. Performance-optimal read-only transactions. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, OSDI'20, USA, 2020. USENIX Association.

[65] H. Lu, K. Veeraraghavan, P. Ajoux, J. Hunt, Y. J. Song, W. Tobagus, S. Kumar, and W. Lloyd. Existential consistency: measuring and understanding consistency at Facebook. In *Proc. SOSP*, Oct. 2015.

[66] J. Manson, W. Pugh, and S. V. Adve. The java memory model. *SIGPLAN Not.*, 40(1):378–391, Jan. 2005.

[67] L. Møldrup and A. Pavlogiannis. AWDIT: An optimal weak database isolation tester. *Proc. ACM Program. Lang.*, 9(PLDI):1540–1564, 2025.

[68] Oracle. JDBC documentation. https://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/.

[69] H. Ouyang, H. Wei, Y. Huang, H. Li, and A. Pan. Verifying transactional consistency of mongodb. *arXiv preprint arXiv:2111.14946*, 2021.

[70] C. H. Papadimitriou. The serializability of concurrent database updates. *JACM*, 26(4), Oct. 1979.

[71] L. Pick, A. Xu, A. Desai, S. A. Seshia, and A. Albarghouthi. Checking observational correctness of database systems. *Proceedings of the ACM on Programming Languages*, 9(OOPSLA1):1661–1688, 2025.

[72] M. R. Rahman, W. Golab, A. AuYoung, K. Keeton, and J. J. Wylie. Toward a principled framework for benchmarking consistency. In *Proc. HotDep*, Oct. 2012.

[73] W. Sun and Z. Zou. Vbox: Efficient black-box serializability verification. arXiv:2503.05163, 2025.

[74] R. Taft, I. Sharif, A. Matei, N. VanBenschoten, J. Lewis, T. Grieger, K. Niemi, A. Woods, A. Birzin, R. Poss, P. Bardea, A. Ranade, B. Darnell, B. Gruneir, J. Jaffray, L. Zhang, and P. Mattis. Cockroachdb: The resilient geo-distributed SQL database. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, 2020.

[75] C. Tan, C. Zhao, S. Mu, and M. Walfish. Cobra: Making transactional key-value stores verifiably serializable. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020.

[76] H. Wada, A. Fekete, L. Zhao, K. Lee, and A. Liu. Data consistency properties and the trade-offs in commercial cloud storage: the consumers' perspective. In *Proc. CIDR*, Jan. 2011.

[77] S. Wang, G. Zhang, J. Wei, Y. Wang, J. Wu, and Q. Luo. Understanding silent data corruptions in a large production cpu population. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 216–230, 2023.

[78] H. Wei, J. Xiao, N. Yang, S. Liu, Z. Yin, Y. Chen, and A. Pan. Boosting end-to-end database isolation checking via mini-transactions. In *Proceedings of the 41st IEEE International Conference on Data Engineering (ICDE)*, pages 3998–4010, 2025.

[79] G. Weikum and G. Vossen. *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery*. Elsevier, 2001.

[80] S. Weng, K. Li, Z. Hu, L. Ni, C. Yang, R. Zhang, P. Cai, X. Zhou, W. Qian, and J. Leng. A high-performance black-box bug-finding tool for database isolation levels. *IEEE Trans. Knowl. Data Eng.*, 38(6):3349–3364, 2026.

[81] Y. Xia, X. Yu, M. Butrovich, A. Pavlo, and S. Devadas. Litmus: Towards a practical database management system with verifiable acid properties and transaction correctness. In *Proceedings of the 2022 international conference on management of data*, pages 1478–1492, 2022.

[82] X. Yang, Y. Zhang, S. Wang, B. Yu, F. Li, Y. Li, and W. Yan. Ledgerdb: A centralized ledger database for universal audit and verification. *Proceedings of the VLDB Endowment*, 13(12):3138–3151, 2020.

[83] K. Zellag and B. Kemme. How consistent is your cloud application? In *Proc. SoCC*, Oct. 2012.

[84] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. K. Ports. Building consistent transactions with inconsistent replication. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, pages 263–278. ACM, 2015.

[85] J. Zhang, Y. Ji, S. Mu, and C. Tan. Viper: A Fast Snapshot Isolation Checker. In *Proc. EuroSys*, 2023.

[86] J. Zhang and C. Tan. Simplifying snapshot isolation: A new definition, equivalence, and efficient checking. In *Proceedings of the 11th Workshop on Principles and Practice of Consistency for Distributed Data*, pages 23–29, 2024.

[87] M. Zheng, J. Tucek, D. Huang, F. Qin, M. Lillibridge, E. S. Yang, B. W. Zhao, and S. Singh. Torturing databases for fun and profit. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 449–464, 2014.