Litmus: Towards a Practical Database Management System with Verifiable ACID Properties and Transaction Correctness

Yu Xia
Massachusetts Institute of Technology
Cambridge, USA
yuxia@mit.edu

Xiangyao Yu
University of Wisconsin–Madison
Madison, USA
yxy@cs.wisc.edu

Matthew Butrovich
Carnegie Mellon University
Pittsburgh, USA
mbutrovi@cs.cmu.edu

Andrew Pavlo
Carnegie Mellon University
Pittsburgh, USA
pavlo@cs.cmu.edu

Srinivas Devadas
Massachusetts Institute of Technology
Cambridge, USA
devadas@mit.edu

Abstract

Existing secure database management systems (DBMSs) focus on security and privacy of data but overlook semantic properties, such as the correctness and ACID properties of transactions. Enforcing these properties is crucial to the functionality of applications. If these guarantees do not hold, catastrophic losses could result.

To address this issue, we present Litmus, a DBMS that can provide verifiable proofs of transaction correctness and semantic properties including atomicity and serializability. Litmus features a co-design of both the database and the cryptographic parts. We evaluate a proof-of-concept prototype of Litmus on the YCSB and TPC-C benchmarks and show that under reasonable cryptographic assumptions it can process more than 17,000 transactions per second (txn/s) *verifiably*. Our result shows a promising practical direction considering that PayPal runs on average 115 txn/s and VISA 2000-4000 txn/s. The proof is about 30kB per verification batch and verifies with a constant time of 300 seconds. Litmus can extend to verify consistency as well.

CCS Concepts

• **Security and privacy** → **Database and storage security**; • **Theory of computation** → *Theory of database privacy and security*.

Keywords

Database Security, Verifiable Computation, Cloud Database

1 Introduction

Organizations are increasingly moving important databases to public cloud platforms. For example, state and local governments use Amazon Web Services to host databases for criminal records [2]. Such outsourcing can reduce hardware and labor costs, but also exposes an organization to data-integrity risks. An attacker that breaches the DBMS can tamper with its contents. In the case of a voter-registration database, an attacker could selectively modify registration data for voters from one political party. An even more problematic scenario is if the organization is unable to detect that a breach has occurred and thus it does not know that it needs to restore the database from backup. Unfortunately, there is ample evidence that such breaches occur often [1, 15] and that cleaning up from them is costly [39].

An additional risk of database outsourcing is the cloud provider's DBMS not actually providing the atomicity, consistency, isolation, and durability (ACID) properties that the provider claims to provide. Software bugs [65] are not the only source of such correctness failures. It has been reported that Machine-Learning-as-a-Service (MLaaS) providers have incentives to lower the service quality as observed in [27]. Similarly, for Database-as-a-Service (DBaaS), risk could also originate from dishonest attempts by the cloud providers to cut costs at the expense of database integrity. For example, running the TPC-C benchmark at a lower isolation level can yield 2.5× better throughput compared to that with serializability [23]. Such ACID failures are commonplace, even in widely deployed database systems [33, 35], and they sometimes even lead to business bankruptcy [53].

Existing solutions test whether a database provides serializability by analyzing the log history [55], or the internal scheduler choices [16, 31, 41, 52, 71]. They either include an independent trusted verifier that is powerful enough to run SAT/SMT solvers and report to the clients, or assume the client itself is capable of handling the analysis.

We present **Litmus**, a *verifiable outsourced DBMS* that provides verifiable atomicity and serializability. It allows data owners to outsource data storage and query processing to the cloud without exposing them to the risk of data-corruption attacks or semantic property violations. With Litmus, the cloud provider will (as it does today) maintain an outsourced database on behalf of the owner. But the Litmus client additionally maintains a small cryptographic digest of the database state. Whenever the owner issues queries, the provider will execute the query and then *prove* to the owner that the query's result is consistent with the owner's digest. If the database state changes while executing a query (e.g., the balance of an account is increased), the cloud will also provide a new digest along with a proof that the new digest accurately represents the state of the old digest with the query applied. To exploit parallelism, the owner can submit multiple transactions (a verification batch) and get a single digest reflecting the new data states, and a succinct aggregated proof that these transactions were executed correctly at the designated isolation level. Verifying such a proof is computationally cheap. With this type of verifiable DBMS, an attacker who compromises the server can at best mount a denial-of-service attack (and the owner will notice). To break data integrity, the attacker must compromise the owner itself, which is equivalent to the no outsourcing scenario. Hence, the promise of verifiable DBMSs is that they can give the same level of integrity protection as a local database with the cost savings and convenience of the cloud.

We target the use case of critical cloud computing scenarios where mistakes could have catastrophic consequences. Compared to a local cluster, a cloud service, even with the overhead of verification, can provide both elasticity and robustness at a lower cost. We eval-

uated a proof of concept prototype of Litmus with YCSB and TPC-C workloads. Litmus with multiple parallel provers is able to verifiably process over 17k txn/sec for simple workloads (YCSB) and 280.6 txn/sec for more complex workloads (TPC-C). We believe Litmus has practical applications in the real world, given that Paypal handles on average 115 transactions per second and the VISA network has a demand of around 2,000-4,000 transactions per second[1].

In this paper, we make the following contributions.

- We present Litmus, a practical and general verifiable database system that provides cryptographic guarantees on data integrity, execution correctness, and transaction semantic properties. Using Litmus blocks the type of attacks described in ACIDRain [65].
- We propose, and use in Litmus, a lightweight authenticated dictionary (AD) scheme based on RSA accumulators that supports key non-existence proofs, which may be of independent interest.
- We improve the DBMS's performance over naive schemes by orders of magnitude by co-designing the DBMS and cryptography. For example, batching non-conflicting transactions enables aggregation of cryptographic proofs.

2 Background and Goals

We introduce the goals for our verifiable database and where we must extend existing work to achieve them. A transaction is a sequence of operations (e.g., read, write, insert, or delete) that a client sends to a database. A database guarantees ACID for transaction processing, which refers to the following four properties [46]:

- **Atomicity**. Either all or no operations of a transaction occur in the database (i.e., *all or nothing*).
- **Consistency**. Any given database transaction must obey semantic invariants including constraints, cascades, and triggers. A transaction cannot leave the database in an invalid state.
- **Isolation**. An isolation level defines when a transaction's effects can be observed by another concurrent transaction. Verifying isolation is more difficult than the other three properties since it involves the coordination of multiple transactions.
- **Durability**. Effects of committed transactions will survive permanently, even if the system crashes.

It is non-trivial to enforce the isolation level since the DBMS can choose any transaction interleaving. Identifying correct interleavings from the exponentially large space of interleavings is proven to be NP-complete [7, 43]. We choose to utilize cryptography to force the DBMS to provide a proof of behaving honestly. Finally, we note that enforcing durability without special hardware is almost impossible because whether or not the storage is physically permanent is not discernible by software.

2.1 Challenges

There are many challenges in designing such a verifiable DBMS: (1) It is not clear how to provide proofs of inter-transaction properties like serializability. Theoretically, sending the logic of the whole DBMS into the verification framework can solve the problem assuming the source code is carefully reviewed. Reality is more challenging because modern DBMSs are complex and cryptography has special requirements on the input logic. (2) Existing cryptographic tools are computationally heavyweight, posing a practicality challenge. (3) To justify the motivation of database delegation, the client is assumed to be lightweight with limited memory.

For the first challenge, we observe that it is not necessary to verify the entire DBMS. We can decouple runtime execution details from the information that needs to be verified. We achieve atomicity and serializability proofs by encoding transactions one-by-one into crypto-friendly formats, adding extra constraints to ensure data integrity.

To address the second challenge, Litmus features a co-design of both the database part and the cryptographic part. We select a batch-based concurrency control (CC) algorithm that identifies a subset of non-conflicting transactions. Witnesses of correctly executing non-conflicting transactions can aggregate into a single succinct one. When the contention level of the underlying workload is not high, this improves the computational overhead of the proving system by orders of magnitude. Further, we parallelize the provers at the task level. This further improves the prover throughput while maintaining a blackbox use of the verifiable computation technique. Finally, for the third challenge, our design lets the client only keep a constant-sized digest, and verify a succinct proof.

2.2 Building Blocks

We now discuss the building blocks of Litmus. We use a verifiable computation framework to prove that transactions execute correctly and their atomicity and isolation properties are guaranteed. We propose a lightweight weakly-binding authenticated dictionary scheme to verifiably track the changes on the data. Finally, we specifically design the CC algorithm to process transactions in batches [59], which enables aggregation of cryptographic computations and witnesses. We discuss how these building blocks work together in Sec. 4.

Verifiable Computation (VC): This is a cryptographic protocol that enables a (usually computationally limited) client to delegate expensive computation to an untrusted server.

The computation is described as a cryptographic circuit. Formally, we define a gate to be a tuple $(d_i, d_o, f : \mathbb{F}^{d_i+d_o} \rightarrow \{\text{yes,no}\})$, where d_i is the in-degree, d_o is the out-degree, and f is a function representing the semantic evaluation of the gate. For example, an AND gate on boolean values will have in-degree 2, out-degree 1, and a function $f(x_1, x_2, y_1)$ that outputs yes if and only if $y_1 = x_1 \wedge x_2$. Given a set of gates $\mathcal{G}$ (e.g., AND, OR, NOT, and 2-FANOUT for a boolean circuit, or ADD and MUL for an arithmetic circuit), a *cryptographic circuit* is a directed acyclic graph where each node is a gate in $\mathcal{G}$, and the edges connecting two gates are *wires*. During evaluation, we assign a value to each wire such that for all the gates, the values of the wires satisfy $f = \text{yes}$. There are always two special gates in $\mathcal{G}$, the INPUT gate and the OUTPUT gate, where an INPUT gate does not have inward edges, but emits the corresponding circuit input as its output. Similarly, an OUTPUT gate absorbs values from other gates and semantically reports these values as the output of the whole circuit. For simplicity, we use $C(\mathbf{x}) = \mathbf{y}$ to indicate that when the INPUT gates emit values in $\mathbf{x}$, the OUTPUT gates will report the values $\mathbf{y}$ as the output.

Given a circuit C known by both the server and client, the verifiable computation scheme proceeds as follows (after a trusted setup as necessary): (1) the client sends $\mathbf{x}$ to the server; (2) the server computes $\mathbf{y} = C(\mathbf{x})$ and generates a proof π, and sends it to the client; (3) The client verifies π against $\mathbf{x}$, $\mathbf{y}$ and C efficiently.

A verifiable computation is *correct* if and only if, when the claimed

[1] Source: https://en.bitcoin.it/wiki/Scalability

output **y** equals $C(\mathbf{x})$, the proof verification always passes. It is *sound* if and only if, when the verification passes, there is only an exponentially small chance that the server could cheat by not computing correctly. Litmus generates program code (e.g., in the C language) of a function that will return false if the transaction semantic properties are violated. Then, it compiles the function into crypto circuits using compilers like Frigate [40], extracts interleaving hints from CC algorithms, and applies a VC scheme on the circuit and the interleaving hints. The VC scheme guarantees that if the function returns true and the proofs pass verification, the client knows that the transactions were executed correctly and the semantic properties are preserved.

Weakly-Binding Authenticated Dictionaries: An authenticated dictionary (AD) scheme enables a client to securely outsource a dictionary to an untrusted server. The client only keeps a succinct digest of the dictionary. The server is able to provide verifiable key-value pair lookups for the client. When the dictionary changes, the digest gets updated accordingly. A weakly-binding AD guarantees the correctness and soundness properties (Sec. 6.1.1) if the digest updating is trusted. In contrast, a *strongly-binding AD* works against a malicious updater. In Litmus, we let the client as well as the delegated computation maintain an AD to track the database state. The client itself is trusted and the delegated computation is guaranteed correct by the VC framework. Therefore, a weakly-binding AD is sufficient.

Deterministic Reservation: This is a CC algorithm that processes transactions by batches [9, 59], which we call *processing batches* to distinguish from the *verification batch* (the number of transactions submitted by the client). It identifies a maximal non-conflicting subset of transactions, as described in Section 7. In our design, deterministic reservation helps the authenticated dictionary scheme "merge" the non-conflicting transactions and provide aggregated proofs of data integrity. This reduces the workload of the VC framework.

2.3 Related Work

Verifiable computation (VC) is a powerful technique to prove the correctness of a program execution where a client offloads the computation to an untrusted computer, while being able to efficiently verify the result. By using general-purpose VC tools [5, 6, 8, 10, 14, 19, 49–51, 62, 63], we can, in theory, construct a verifiable database system that satisfies the cryptographic properties of Sec. 3 by compiling the whole DBMS into a giant circuit. Even though it shows promising asymptotic results [47], it would still incur an impractical computational overhead due to large constant factors. Litmus only verifies essential parts of the database and parallelizes the provers to achieve a practical throughput.

Authenticated Data Structures: Cryptographic accumulators [25], multiset hashes [20], vector commitment [18], and authenticated dictionaries [60] are widely used in verifiable data storage in various settings. For example, [32] uses Merkle trees to keep track of the data on the server. vChain [64, 69] proposes new *authenticated data structures* based on *bilinear mapping groups* to verify queries on blockchains. Similar to vChain, Litmus also allows batched verification and utilizes aggregation to boost the performance. However, different from vChain, Litmus targets OLTP transactions on a cloud database, while vChain allows expressive queries on blockchains, where the blocks are immutable (read-only). Besides data integrity, works like [68] use authenticated data structures and attribute-based signatures to authenticate queries with fine-grained access control and protect data privacy against unauthorized users.

Verifiable databases in the single-transaction setting: vSQL [73] and IntegriDB [74] construct VC schemes that handle processing of a non-trivial subset of SQL, one query at a time, while Litmus focuses on concurrent transactions with *read* and *write* operations.

Verification of concurrent systems: Recent work considers the task of verifying general-purpose computations in a concurrent setting [54]. Orochi [54] is a system for verification of PHP web applications. Spice [48] addresses the verifiable concurrent execution problem, and provides low-level mutual-exclusion primitives. Since none of these works consider verification of a transaction's ACID properties, they are orthogonal to our work.

Checking serializability: Recent work has proposed using external programs to verify the serializability of transactions in a DBMS [16, 31, 36, 41, 52, 55, 56, 56, 70**?** , 71]. These programs take the traces from a transactional database and verify whether the execution is serializable. These techniques either require the DBMS to both generate these traces and send them to a verifier (which is likely impractical), or use SAT or SMT solvers to compute a possible sequential interleaving [56]. Works like Elle [35] require the database to make data acesses into list operations to keep track of the history of the tuples; we evaluate Elle in Sec. 8.3. Other approaches that provide verifiable serializability include [30, 32]. Haeberlen et al. applies to general distributed systems settings, where the nodes are uniform and at least one node is honest [30]. Compared to the classic Merkle-tree approach, [32] is novel in decoupling the data owner and clients, and introducing postponed verification; this enables the server to process transactions in parallel. In particular, [32] resembles the Merkle tree baseline in our evaluation (c.f. Sec 8) and their evaluation is consistent with our observation (<20 txn/s at 100% verification level).

3 Cryptographic Formalization

We now present a sketch of the cryptographic framework that we use to formalize ACID properties. We focus on verifying serializability, the highest level of isolation, and atomicity, though the formalism naturally extends to other isolation levels as well as consistency (as defined by invariants before and after applying transactions). We discuss durability separately in Sec. 9.

Formally, we model the database state as a bitstring $D \in \{0,1\}^*$ (as some encoding of a dictionary). We model a transaction $T: \{0,1\}^* \rightarrow \{0,1\}^* \times \{0,1\}^*$ as a function that maps the old database state D to a new database state D' and an output value v. For example, the output value v could be the result of a transaction that updates a database row and then executes a `SELECT` query. A *verifiable database scheme* is then a tuple of algorithms as follows:

- $\text{Digest}(D) \rightarrow \delta$. Compute a constant-sized cryptographic digest δ of the database D.
- $\text{Execute}(D,\mathcal{T}) \rightarrow (D',\mathcal{V},\pi)$. Given a database state D and a list of transactions $\mathcal{T}$, apply the transactions to the database (in some order) to produce a new database D', a list of output values $\mathcal{V}$ (one per transaction), and a proof π of computation correctness.
- $\text{Verify}(\mathcal{T},\delta,\delta',\mathcal{V},\pi) \rightarrow \{0,1\}$. Given a list of transactions $\mathcal{T}$, a digest δ of the old database state, a digest δ' of the new database state, a list of output values $\mathcal{V}$, and a claimed proof of correctness π, check the proof and output "1" if and only if the proof is valid.

For simplicity, we omit the cryptographic security parameter and the public parameters of the scheme. For a verifiable database system to be useful, it should be *correct* and *sound*. Informally, correctness states that an honest database server is able to convince an honest client that it has correctly executed a list of transactions.

Definition 1 (Correctness)*:* A verifiable database scheme (Digest, Execute, Verify) is *correct* if for all lists of transactions $\mathcal{T}$ and all database states $D \in \{0,1\}^*$,

$$\Pr\left[\mathsf{Verify}(\mathcal{T},\delta,\delta',\mathcal{V},\pi)=1 : \begin{array}{r}\delta \leftarrow \mathsf{Digest}(D),\\ (D',\mathcal{V},\pi) \leftarrow \mathsf{Execute}(D,\mathcal{T}),\\ \delta' \leftarrow \mathsf{Digest}(D')\end{array}\right] = 1.$$

Informally, a verifiable database scheme is sound for serializability if, for all lists of transactions $\mathcal{T} = \{T_1,...,T_n\}$, whenever an adversary produces digests δ, δ', a list of outputs $\mathcal{V} = \langle v_1,...,v_n\rangle$, and a proof π that the verifier accepts, this adversary "must know" corresponding databases D_0 and D_n and a permutation σ on $\{1,...,n\}$ that "explain" the new digest δ' of the database state and the outputs in $\mathcal{V}$. Namely,

(a) $\delta = \mathsf{Digest}(D_0)$,
(b) for $i = 1, ..., n$: $(D_i, v_i) \leftarrow T_{\sigma(i)}(D_{i-1})$, and
(c) $\delta' = \mathsf{Digest}(D_n)$.

We formalize this notion of "knowledge" with an extractor $\mathcal{E}^{(\mathcal{A})}$ with a oracle access to the adversary $\mathcal{A}$ as follows.

Definition 2 (Soundness – Serializability)*:* A verifiable database scheme (Digest, Execute, Verify) is *sound for serializability* if there exists a probabilistic polynomial time (p.p.t.) extractor $\mathcal{E}$ s.t. for any p.p.t. adversarial database server $\mathcal{A}$, for all lists of transactions $\mathcal{T} = \{T_1,...,T_n\}$ that $\Pr[\mathsf{Verify}(\mathcal{T}, \mathcal{A}(\mathcal{T})) = 1]$ is non-negligible, the extractor $\mathcal{E}^{(\mathcal{A})}$ outputs databases (D_0, D_n) and a permutation σ on $\{1,...,n\}$ such that the following quantity is *negligibly* close to 1 in the (implicit) security parameter:

$$\Pr\left[\begin{array}{r}\mathsf{Digest}(D_0) = \delta\\ \text{for all } i \in \{1,...,n\}:\\ (D_i, v_i) \leftarrow T_{\sigma(i)}(D_{i-1})\\ \mathsf{Digest}(D_n) = \delta'\end{array} : \begin{array}{r}(\delta,\delta',\mathcal{V},\pi) \leftarrow \mathcal{A}(\mathcal{T}),\\ (D_0, D_n, \sigma) \leftarrow \mathcal{E}^{(\mathcal{A})}(\mathcal{T})\end{array}\right].$$

Note that this definition also implies atomicity because no transactions are partially executed. Extensions of this definition allow capturing other isolation levels and consistency.

4 System Overview

This section presents an overview of the verifiable database system. We assume a single client that interacts with a single database server. In the DBaaS setting, the single client is the organization that delegates the database, which might be the proxy of millions of real users and submit many transactions. In Sec. 7, we introduce concurrency where the client submits a batch of transactions.

In this section, we focus on the verification of *Atomicity* and *Isolation* (namely, *Serializability*) of the database. We will discuss *Consistency* and *Durability* in Sec. 9. To provide verifiable isolation, we need a global scope for transactions because the isolation property places constraints on the interleaving of transactions. We introduce the concept of **wrapped transaction** to help the VC scheme handle the interleavings. A wrapped transaction is a set of transactions "glued" together, with the logic of the memory integrity checker (c.f. Section 6.1.1) plugged into each transaction. Specifically, before every transaction starts to run its own logic, it first runs the checker to see if the provided memory digest along with the memory modifications are consistent with its local digest. If the check passes, the transaction continues, otherwise it aborts by directly returning 0.

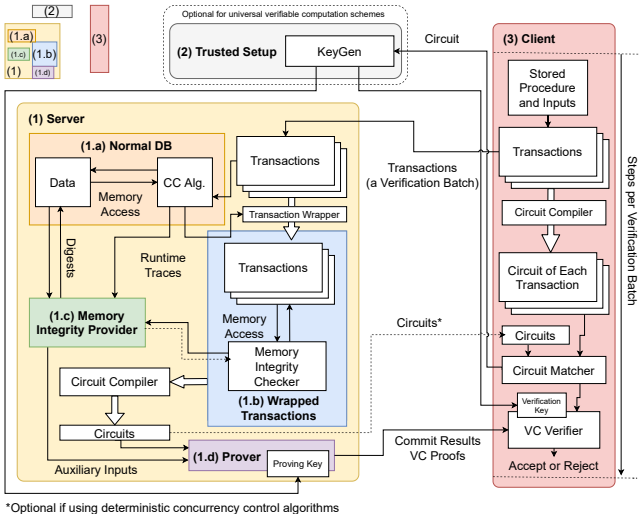


Figure 1: Overview of the Proposed Verifiable Database — The system contains three modules as shown in the top left corner: (1) the verifiable DBMS with (1.a) a normal DBMS system, (1.b) the wrapped transactions, (1.c) the memory integrity provider, and (1.d) the VC prover; (2) the (optional) trusted third-party setup; and (3) the client.

As shown in Figure 1, the system contains three key modules: (1) The **server** hosts the **normal DBMS**, receives transactions from the clients, and creates a **wrapped transaction**; The server also contains the **memory integrity provider** and the **VC prover**. (2) The optional **key generator** can be the client itself or a trusted third-party, or be implemented by another VC instance if the client wants the server to carry the heavy computation (as the key generation logic is fixed, there is no circular dependency on the key generation). Alternatively, we can use a *universal VC scheme*, where the keys do not depend on the circuits, making key generation a one-time cost; (3) The **client** is the organization or program that submits transactions.

We now describe the modules in greater detail. The **client** is a commodity machine that does not have to be computationally strong. Before the system starts, we assume that the client has stored enough information to define a group of transactions, e.g., a stored procedure with a set of input parameters. The client first sends the transactions to the server. Then, it passes the transactions to the circuit compiler to obtain the circuit representation of each of the transactions. Upon receiving the circuit of the wrapped transaction (c.f. (1b)) from the server, the client tries to match the local circuits and the circuit of the wrapped transaction sent from the server. Then, the client sends the circuit to the key generator. The client also contains the verifier part of the VC framework, which takes in the verification key from the key generator and the proof generated from the server, and outputs a single bit indicating whether the proof is accepted or rejected.

The optional **key generator** takes in the circuit and a sufficient amount of randomness, and produces the proving key σ and the verification key τ. The proving key is sent to the server and the verification key is sent to the client.

We now describe the four components of the server.

(1a) Normal DBMS: This is a full-fledged database system. Theoretically, it can run any valid CC algorithm like two-phase locking (2PL)

or optimistic concurrency control (OCC). However, we choose to use the *deterministic reservation* CC algorithm in Litmus (see Section 7.1) to reduce the size of the circuit by aggregating the cryptographic witnesses of a set of non-conflicting transactions into a single succinct witness. Because this CC algorithm is deterministic, the client by itself may be able to infer the transaction interleaving and produce the whole circuit. The key generation may start even before the server finishes running all the transactions, escaping the critical path of performance. The normal DBMS also generates runtime traces, namely, the transaction interleaving and data updates, which closely resembles logging records. Just like a DBMS could support data logging and command logging, the traces could be as small as a few bytes indicating the transaction order and their inputs (as in command logging), or an extensive list of all the data changes (as in data logging).

(1b) Transaction Wrapper and Circuit Compiler: The transaction wrapper is a tool to plug a memory integrity check into the starting point of every transaction in a group, and merge them into a single transaction. It takes in a group of transactions as well as the specification of the memory integrity checker, and outputs a circuit representing the wrapped transactions. This is similar to compilers adding instrumentation to source code to add features to the program (e.g., producing traces for debuggers). The wrapped transaction generated by the transaction wrapper is then compiled into logic circuits acceptable to the verifiable computation scheme. The circuit is analogous to the binary program produced by a compiler. It expresses the same logic but in a more low-level representation. The circuit is then used by the key generator to generate the proving key and the verification key, and by the prover to provide cryptographic proofs of the circuit being evaluated correctly.

(1c) Memory Integrity Provider: The memory integrity provider helps the circuit keep track of data changes and provides proofs on the values read from the database. It listens to runtime traces of concurrency control algorithms and generates a sequence of memory digests. The difference between each consecutive pair of memory digests reflects a modification by a subset of transactions.

(1d) Prover: The VC prover takes in the proving key, the circuit generated by the transaction wrapper, and the inputs supplied by the memory integrity provider. It outputs a proof indicating the transaction's output is correct with respect to the input transactions. Generating the proof is usually computationally heavy and the running time depends on the size of the circuit. However, the size of the final proof is not necessarily long, and the client can verify the proof even in constant time for some existing VC constructions.

5 Authenticated Dictionary Scheme from RSA

Before diving into the details of the system, we discuss a useful cryptographic primitive to guarantee data integrity in Litmus.

We propose a new weakly binding AD scheme that only needs a constant length of storage and a constant number of arithmetic operations for each verification. The AD scheme is based on an RSA accumulator [13]. One might argue that it is trivial to build a weakly binding AD from an RSA accumulator by simply hashing the key-value pairs into distinct primes. However, we believe the naive construction is not suitable for databases because it does not efficiently support key non-existence proofs. Transactions only visit a few spots in the memory compared to the vast memory space. If we adopt the naive approach, the client has to encode *all* memory values into the accumulator, which is catastrophic in terms of running time. Our approach efficiently supports key non-existence proofs so that the server can prove that the requested key was not previously accessed, and provide an initial value, say 0, previously agreed with the client.

5.1 Prime Categorization

At a high level, our construction relies on *categorization of prime numbers* to accumulate multiple types of information at the same time. Specifically, we use a dynamic universal RSA accumulator as the building block. We categorize prime numbers into three categories that each contains an infinite number of primes. We use the first category to encode keys, the second category to encode values, and the last category to encode the relationship between keys and values. This construction enables us to produce constant-sized proofs of lookups and to verify such proofs with a constant number of operations. These properties make our AD scheme circuit-friendly.

Categorization of Prime Numbers: A categorization of prime numbers is a set of disjoint subsets $\text{cat} = (\mathbb{P}_1, \mathbb{P}_2, \ldots, \mathbb{P}_l)$, such that $\bigcup_i \mathbb{P}_i = \mathbb{P}$, where $\mathbb{P}$ is the set of all the primes. A categorization scheme consists of two deterministic algorithms (Sample,Verify) that satisfy the following:

- $\text{Sample}(\lambda, i, nonce)$ takes in the bit-length λ and a categorization index $i \in [l]$. It returns a prime number $p \in \mathbb{P}_i$ with λ bits.
- $\text{Verify}(p, i)$ takes in a number p and a categorization index $i \in [l]$. It returns a bit yes/no indicating whether or not $p \in \mathbb{P}_i$.

We say a categorization cat is *feasible* if and only if Sample and Verify are probabilistic polynomial time bounded by λ, Sample outputs a unique prime number for a nonce. The following holds.

Definition 3 (**Correctness**)*:* For any bit-length λ, a categorization index $i \in [l]$, and any nonce, we have

$$\Pr[\text{Verify}(\text{Sample}(\lambda, i, nonce), i) = \text{yes}] = 1.$$

Definition 4 (**Soundness**)*:* For any adversary A, any bit-length λ and a categorization index $i \in [l]$, we have

$$\Pr[\text{Verify}(p, i) = \text{yes} \wedge p \notin \mathbb{P}_i : (p, i) \leftarrow A(1^\lambda)] = 0.$$

A simple way to construct a finite number of prime categories is by modulo residue. For example, $\mathbb{P}_1 := \{\pm 1 \pmod 8\} \cap \mathbb{P}$, $\mathbb{P}_2 := \{3 \pmod 8\} \cap \mathbb{P}$, and $\mathbb{P}_3 := \{5 \pmod 8\} \cap \mathbb{P}$ form three categories. The reader might worry that proving modulo operations involves expensive range proofs on the residues. We let the circuit include dedicated wires for all the possible residues $\{1, 3, 5, 7\}$ for an odd prime. To show that a residue is well-defined, we assert it to be equal to one of $\{1,3,5,7\}$. Hence, the server can simply provide the quotient q and residue r. A single constraint $p = 8q + r$ suffices to show the category (we handle primality tests separately). Examples include $17 \in \mathbb{P}_1$, $11 \in \mathbb{P}_2$, and $13 \in \mathbb{P}_3$.

5.2 Assumption and Interfaces

The AD scheme relies on the *strong RSA assumption* [3].

Definition 5 (Strong RSA Assumption)*:* Given two primes p and q of bit-length λ, let $N := pq$. It holds for all p.p.t. adversary $\mathcal{A}$ that

$$\Pr[u^x \equiv a \pmod N : (u, x) \leftarrow \mathcal{A}(a, N), a \leftarrow_\$ \mathbb{Z}_N^*] \leq \texttt{negl}(\lambda).$$

Before we dive deeper, we provide the definition of ADs.

Definitions: An AD scheme consists of the following APIs.
$\text{Setup}(1^\lambda) \rightarrow (\text{pk}, \text{vk})$. Returns the *proving* and *verification keys*.

Commit(pk,D) $\to d$. Returns a digest d of the dictionary D.

Update(pk,D,d,K,V) $\to d'$. Update the digest d by setting the value of the key $k \in K$ to be $V(k)$. Return the new digest d'.

ProveLookup(pk,d,D,V,K) $\to \pi$. Returns a *lookup proof* π that each $k \in K$ has value $V(k)$.

VerLookup(vk, d, K, V, π) $\to$ {yes, no}. Verifies the proof that each $k \in K$ has value $V(k)$ in the dictionary with digest d.

ProveNoKey(pk,d,D,K) $\to \pi$. Returns a *non-membership proof* π that there does not exist any key value pair (k,v) with $k \in K$ in the dictionary with digest d.

VerNoKey(vk,d,K,π). Verifies the proof π that each $k \in K$ does not exist in the dictionary with digest d.

A weakly binding authenticated scheme observes two properties, namely, *correctness* and *weak key binding*.

Definition 6 (**Correctness**)*:* An authenticated dictionary scheme is correct if, $\forall$ public parameters (pk,vk) $\leftarrow$ Setup(1^λ), $\forall$ dictionaries D with digest $d \leftarrow$ Commit(pk,D), the following hold:

- Lookup correctness: $\forall$ sets of keys K, if π = ProveLookup(pk, D, K, V) and $V(k) = D(k)$, $\forall k \in K$, then VerLookup(vk, d, K, V, π) = 1.
- Key Non-membership correctness: $\forall$ K, if π = ProveNoKey(pk, d, D, K) and $\forall k \in K$, k is not in D, then VerNoKey(vk, d, K, π) = 1.

Definition 7 (**Weak Key Binding**)*:* $\forall$ adversaries $\mathcal{A}$ running in time poly(λ), there exists a negligible function negl($\cdot$), such that the following inequalities hold:

Lookup Soundness:

$$\Pr\left[\begin{array}{c}(\mathsf{pk},\mathsf{vk}) \leftarrow \mathsf{Setup}(1^\lambda),\\ (D,K,K',V,V',\pi,\pi') \leftarrow \mathcal{A}(1^\lambda,\mathsf{pk},\mathsf{vk}):\\ d = \mathsf{Commit}(\mathsf{pk},D)\\ \mathsf{VerLookup}(\mathsf{vk},d,K,V,\pi)=1 \wedge\\ \mathsf{VerLookup}(\mathsf{vk},d,K',V',\pi')=1 \wedge\\ \exists k \in K \cap K' \text{ s.t. } V(k) \neq V'(k)\end{array}\right] \leq \mathsf{negl}(\lambda).$$

Key Non-membership Soundness:

$$\Pr\left[\begin{array}{c}(\mathsf{pk},\mathsf{vk}) \leftarrow \mathsf{Setup}(1^\lambda),\\ (D,K,K',V,\pi,\pi') \leftarrow \mathcal{A}(1^\lambda,\mathsf{pk},\mathsf{vk}):\\ d = \mathsf{Commit}(\mathsf{pk},D)\\ \mathsf{VerLookup}(\mathsf{vk},d,K,V,\pi)=1 \wedge\\ \mathsf{VerNoKey}(\mathsf{vk},d,K',\pi')=1 \wedge\\ \exists k \in K \cap K'\end{array}\right] \leq \mathsf{negl}(\lambda).$$

5.3 Extending RSA Accumulators to AD

Now we are ready to extend an existing dynamic universal RSA accumulator scheme (e.g. [11]) to a weakly binding authenticated dictionary scheme, given a *feasible* categorization of prime numbers. For simplicity, we first define

$$H(k,v) = \mathsf{Sample}(\lambda,0,k)\cdot\mathsf{Sample}(\lambda,1,v)\cdot\mathsf{Sample}(\lambda,2,h(k,v)),$$

where $h(k,v)$ is a collision-resistant hash function.

Setup(1^λ) $\to$ (pk,vk). Sample an RSA group $\mathbb{G}_?$ with generator g. Set pk = g and vk = g. The order of $\mathbb{G}_?$ remains secret.

Commit(pk,D) $\to d$. For each key-value pair $(k,v) \in D$, we sample three primes from each of the categories. The primes correspond to the key, the value, and the relationship of the key and value (represented by a hash $h(k,v)$). Formally, we compute

$$d \leftarrow g^{\prod_{(k,v)\in D}[H(k,v)]}.$$

ProveLookup(pk,D,K) $\to \pi$. Denote the dictionary after removing the key-value pairs with keys $k \in K$ as $D \backslash K$. We produce a digest of this sub-dictionary, serving as the proof. Similar to Commit, the proof equals

$$\pi \leftarrow g^{\prod_{(k,v)\in D\backslash K}[H(k,v)]}.$$

VerLookup(vk,d,K,V,π) $\to$ {yes,no}. Checks whether

$$\pi^{\prod_{(k,v)\in(K,V)}[H(k,v)]} = d.$$

Update(pk, D, d, K, V) $\to d'$. We first compute $\pi \leftarrow$ ProveLookup (pk,D,K), then build the new digest d' based on π:

$$d' \leftarrow \pi^{\prod_{(k,v)\in D\backslash K}[H(k,v)]}.$$

ProveNoKey(pk, d, D, K) $\to$ $\{A,B\}$. We compute (A,B) = *Bezout*(S, $\prod_{k\in K}$Sample(λ,0,k)), where *Bezout*(x,y) returns the Bezout coefficients A,B s.t. $Ax+By=1$ for x,y with $gcd(x,y)=1$.

VerNoKey(vk,d,K,A,B) $\to$ {yes,no}. Checks whether

$$d^A \cdot \left(g^{\prod_{k\in K}\mathsf{Sample}(\lambda,0,k)}\right)^B = g.$$

One important property that enables our optimization in Sec. 7 is *aggregability*. As shown in the ProveLookup and VerLookup interfaces, we can aggregate a number of lookups into a set of keys and provide a single proof for all of them. This means we can batch non-conflicting transactions and prove/verify the memory access once for all their lookups. This property is widely used in RSA accumulators [12]. Our AD scheme inherits this property.

As an example, suppose we want to compute a digest d of dictionary D where $D[1] = 2$, $D[3] = 4$. Then, $d = g^{H(1,2)\cdot H(3,4)}$. To prove $D[1] = 2$, the server needs to compute $\pi = g^{H(3,4)}$ as the proof. To verify, the client can check if $\pi^{H(1,2)}$ equals d. If the dictionary otherwise does not contain the key-value pair (1,2), the server has to compute π from d itself. By the Strong RSA Assumption, it is difficult to compute $d^{1/H(1,2)}$. Therefore, even if the server is malicious, it is not likely to produce a proof passing the verification.

The reader might wonder why we only used Sample so far but not Verify. The reason is that, given that the circuit is computationally weak and deterministic, it is impractical for the circuit to sample prime numbers on its own. It needs the prover to supply candidates of Prime numbers as auxiliary inputs, and it calls Verify to test primality and its category (e.g., via Pocklington [42, 45]). *Pocklington* adds extra conditions to the Fermat primality test to make the conditions sufficient. Specifically, if there exists an integer a and a prime p such that $a^{N-1} \equiv 1 \pmod N$, $p|N-1$, $p > \sqrt{N}-1$, and $\gcd(a^{(N-1)/p}-1, N) = 1$, then N is prime. With this result, the server can provide a small prime number p_0 and prove its primality through a deterministic primality test, and "boost" it up by providing (r,a) s.t. $r < p_0$, $\gcd(a^r - 1, rp_0+1) = 1$, $a^{rp_0} \equiv 1 \pmod{rp_0+1} \Rightarrow N = rp_0+1$. This is a prime number by Pocklington. This process can be repeated multiple times to reach a prime number that is large enough. Whenever the circuit calls Sample on a nonce, the server needs to provide the prime number and $p_0, \pi_{p_0}, \{a_i, r_i, \pi_i\}$ to the circuit, where π_{p_0} is the deterministic primality proof for p_0, and $\{\pi_i\}$ are the proofs that a_i and r_i satisfy the conditions. To make the whole process deterministic, the choice of certificates a_i, r_i depends on the nonce. The nonce could be a key, a value, or a hash value from a key-value pair. As the length of the prime number only depends on the security parameter λ, we only need to boost $O(\lambda)$ times. For example, if we already know $p_0 = 59$ is a prime, to prove 827 is also a prime, we can pick $a = 2$, $r = 14$ as our certificate.

6 Single-Threaded 2PL Baseline

We present a baseline system that provides verifiability of atomicity and isolation properties. Although Litmus's server can work with any CC algorithm if it can access transactions' interleaving information, we use single-threaded 2PL to simplify our discussion. We also extend the design to multiple threads in Sec. 7.

6.1 Server

We first present Litmus's memory integrity model. Next, we discuss its transaction wrapper and circuit compiler. Lastly, we explain how a normal database interacts with these components.

6.1.1 Memory Integrity. The memory integrity scheme uses authenticated dictionaries to maintain data integrity. The core design philosophy of our memory integrity model is to make the circuit as small as possible. Existing works do not suit our purpose because they either need a variable length digest (e.g., Merkle Tree) or require significant time to verify. We use the same approach as Pantry to enable a memory access interface for circuits [14]. As shown in Algs. 1 and 2, the memory integrity model consists of two parts — the *provider* and the *checker*. The provider runs natively on the server and generates proofs for values read by the transactions. The checker runs as a part of the circuit and checks the proofs. The server and the client agree on the initial state (g_0, D_0) before the protocol starts. We require an *consistent* initial digest, namely $g_0 = \mathsf{Commit}(\mathsf{pk}, D_0)$. The initial digest does not have to cover all the possible memory addresses. For example, D_0 could be empty.

The memory integrity provider maintains two variables, S and acc. The former (S) tracks the product of the elements, and acc stores the latest AD digest. The provider also maintains a dictionary to keep track of the memory changes. It initializes S to be the product of the hashes of the key-value pairs in D_0 and acc to g_0. It holds that $g_0 = g^S$. Due to our AD scheme, the digest g_0 and the initial product s_0 do not have to include initial values for all the memory addresses since initializing the memory is a significant cost for the client.

When the system calls `GenReadProof` (Alg. 1), Litmus computes the corresponding proofs for the client and returns them. It first checks if the memory address is in the local cache D. If yes, it returns a lookup proof $\pi = g^{S/H(k,v)}$. Otherwise, it returns a non-existence proof of the key k, indicating that no values have been written to the address k, and the circuit should accept an initial value.

The `UpdateWrite` operation is simpler (Alg. 1). It reads the old value at address k from the dictionary D, and computes the lookup proof $\pi = g^{S/H(k,v')}$. Then, it updates the digest to be $\pi^{H(k,v)}$, and the product S accordingly. Finally, it updates D. Note that if we assume no blind writes, it gets the value of π for free.

The memory integrity checker consists of three interfaces, `MemInit`, `MemCheck`, and `MemUpdate`. Before a transaction starts, the circuit first invokes `MemCheck` (Alg. 2) to determine whether the read values are correct according to its local digest, namely, $\pi^{H(k,v')} = \mathsf{acc}$. If this fails, it checks whether the proof indicates that the address was not accessed before so the value should be an initial value. If both fail, the function returns 0, indicating the integrity is compromised.

After the transaction finishes execution, the circuit runs `MemUpdate` (Alg. 2). The server provides auxiliary inputs: the address k, the old value v', the new value v, and the lookup proof π of the pair (k, v'). It first checks if π is valid. The circuit can skip the verification if we assume no blind writes since π is already verified in the read operation.

Algorithm 1: Memory Integrity Provider (on the Server Side) with initial database state agreed as (g_0, D_0).

Input: `AgreedInitState` = $\{g_0, D_0\}$
$S = \prod_{(k,v)\in D_0} H(k,v)$; acc $= g_0$; $D = D_0$; /* initialization */
Func `GenReadProof` *(k, v)*:
  if *k is in D* **then**
    return $\pi = g^{S/H(k,v)}$; /* generate the lookup proof */
  else
    return $(A,B) = Bezout(S, \mathsf{Sample}(\lambda, 0, k))$; /* non-existence proof */
Func `UpdateWrite` *(k, v)*:
  $v' = D[k]$; /* old value */
  $\pi = \mathsf{GenReadProof}(k, v')$; /* must be a lookup proof */
  $\mathsf{acc} = \pi^{H(k,v)}$; /* update the digest */
  $S = S/H(k,v') \cdot H(k,v)$; /* update the product */
  $D[k] = v$; /* update the dictionary */
  return

Algorithm 2: Memory Integrity Checker (inside the Wrapped Transaction) with initial database state (g_0, s_0).

Input: `AgreedInitState` = (g_0, s_0)
Global variable acc maintained by a dedicated wire;
Func `MemInit` *()*:
  acc $= g_0$; /* initialize the local digest */
Func `MemCheck` *(k, v, π, A, B)*:
  if $\pi^{H(k,v)} = \mathsf{acc}$ *or* $(\mathsf{acc}^A \cdot g^{B \cdot H(k,v)} = g$ *and* $v = 0)$ **then**
    return 1 ; /* verification passes */
  return 0
Func `MemUpdate` *(k, v′, v, π)*:
  if $\pi^{H(k,v')} ! = \mathsf{acc}$ **then**
    return 0 ; /* verification failure */
  acc $= \pi^{H(k,v)}$; /* update the local digest */
  return 1

Lastly, the circuit updates acc to be $\pi^{H(k,v)}$.

There are no loops in the pseudo-code in the memory integrity checker. Every variable has a fixed length (only depending on the security parameter) except A and B. The only concern here is computing large exponentiation. We can address this by letting the server provide the result directly with a Proof-of-Exponent [11]. This technique can shrink down the size of A and B to be *constants*. Overall, the memory integrity checker only contributes a constant number of gates to the circuit per memory access in the transaction[2].

6.1.2 The Transaction Wrapper. This component takes in a list of transactions $\{T_i\}$ and the runtime traces `RuntimeTraces`, and builds a single **wrapped transaction** (represented as a function) from $\{T_i\}$ by chaining them sequentially and inserting memory integrity checking code before each transaction starts. It constructs a graph representing transactions and their partial orders decided by the CC algorithm. Then, it performs a topological sort on the transactions such that the partial orders are all satisfied. Next, it builds the wrapped transaction. The wrapped transaction takes in inputs of (1) read values passed by the memory and (2) the proofs of the memory digests. It initializes the local memory digest. Then, it runs the transactions one by one and for each transaction T_i in the list, it checks if the corresponding read values provided by the input are correct by using the memory integrity proofs. It runs the transaction with the read values. While running the transactions, it collects the written values and updates the local memory digest accordingly. If any of the memory integrity checks fail, the return value of the wrapped transaction will start with a 0 bit if evaluated correctly by the server.

[2] The memory integrity checker performs exactly three exponentiations, two multiplications, three comparisons, and two boolean operations per request. This logic produces a constant number of gates in the circuit.

Algorithm 3: The Serial Transaction Wrapper

```
1  Func TransactionWrapper (A set of transactions {T_i}, runtime traces RuntimeTraces):
2      /* Construct the wrapped transaction                                  */
3      Construct a graph T with nodes {T_i} and edges (T_i → T_j) ∈ RuntimeTraces;
4      Perform topological sort on T and get a list of transactions (T_i)_<;
5      Func WrappedTransaction (ReadVals, memproof):
6          MemInit ();
7          AllCommit = 1;
8          for each T_i in (T_i)_< do
9              if MemCheck (ReadVals, memproof) then
10                 CommitFlag_i, WriteVals = T_i.run (ReadVals);
11                 CommitFlag_i = CommitFlag_i * MemUpdate (WriteVals);
12                 AllCommit = AllCommit * CommitFlag_i;
13             else
14                 AllCommit = 0;
15         return AllCommit;
16     return the function code of WrappedTransaction;
```

The wrapped transaction does not need to represent a sequential chain of transactions. As we will discuss in Sec. 7, forming a wide network of transactions is better for performance; the shape of the returned wrapped transaction is critical for exploiting parallelism.

6.1.3 Circuit Compiler and Circuit Matcher. The circuit compiler compiles the wrapped transaction into a monolithic circuit on the server side, and compiles each transaction into separate circuits on the client side. The client runs the circuit matcher to determine if the circuit claimed by the server matches the local sets of circuits.

The compiler converts the description of the wrapped transaction (in high-level programming languages or in LLVM-like representations) into a Rank-1 Constraint System. A carefully designed circuit compiler can optimize the structure of the circuit without changing the underlying logic. This is similar to what modern compilers do to re-order instructions for multi-issue architectures.

A malicious server is free to generate any circuit it prefers and pass it to the client and the prover. Therefore, the client must check whether the wrapped transaction circuit is valid or not. First, it checks whether the logic of the transactions is consistent with that in the wrapped transaction. Secondly, the circuit matcher checks if the memory integrity checker in the circuit is correctly plugged in. One can reduce these two tasks to pattern matching problems as the same transaction logic would be compiled into the same circuit, and a known memory integrity checker also results in a known circuit description. Later, in Sec. 7, we will show how to extend Litmus to use a deterministic CC algorithm with fixed batch sizes. In this case, the client does not need to perform circuit matching since it can locally produce the transaction interleaving (if the writesets do not depend on the read set). Instantiations of such deterministic circuit compilers exist and can be re-purposed for our verifiable DBMS [17, 40].

6.1.4 Normal Database and Concurrency Control. To verify the isolation property, we need to track runtime traces (namely, transactional dependencies) and guarantee memory integrity. We track transactional dependencies because the circuit should follow the interleaving of real transaction execution (otherwise the read values and proofs provided by the memory integrity provider are inconsistent). Furthermore, the dependency information can serve as hints to the VC prover to prepare the proofs faster. We make the following changes to the 2PL algorithm to obtain transactional dependencies, and prepare memory integrity proofs and memory digest updates.

We add two metadata fields to the data items, `LastReader` and `LastWriter`, indicating the set of last readers and last writers, respectively. The server initializes all of the data items in the database,

Algorithm 4: The Workflow of the Server

```
   Input: A previously agreed initial statue of the database AgreedInitState
1  Initialize DB.Data[*].LastReader = DB.Data[*].LastWriter = ∅;
2  Upon receiving message (MSG_TXN, {T_i}) from the client:
3      Initialize the memory accumulator acc = Acc.Accumulate(AgreedInitState);
4      Initialize the proof list proofList ← [ ], the read values readList ← [ ];
5      DB.Run ({T_i});
6          When T_i reads an item DB.Data[k]:
7              Acquire the shared read lock on DB.Data[k];
8              Append (DB.Data[k].LastWriter → T_i) to RuntimeTraces;
9              Append T_i to DB.Data[k].LastReader;
10             Append (k,DB.Data[k]) to T_i.read_set;
11             Append GenReadProof(acc,k,DB.Data[k]) to T_i.proofs;
12         When T_i writes an item DB.Data[k] with a new value v:
13             Acquire the exclusive write lock on DB.Data[k];
14             Append (DB.Data[k].LastWriter → T_i) and
                 (DB.Data[k].LastReader → T_i) to RuntimeTraces;
15             DB.Data[k].LastWriter = T_i;
16             DB.Data[k].LastReader = ∅;
17             Call UpdateWrite(acc,k,v);
18         When a transaction T_i commits:
19             Append T_i.proofs to proofList;
20             Append T_i.read_set to readList;
21         When the database finishes execution, get the runtime traces
             RuntimeTraces = {(T_i → T_j)}_{i,j} and the final output y;
22     C = CircuitCompiler(TransactionWrapper({T_i},RuntimeTraces));
23     Send (MSG_WRTXN,C) to the client;
24 Upon receiving message (MSG_PKEY,σ) from the third-party:
25     Initiate the prover VC.Prove on C and σ;
26     Feed readList and proofList as circuit inputs to the prover;
27     Feed RuntimeTraces as auxiliary information to the circuit to the prover;
28     Get the proof π from the prover and send (π, y) to the client;
```

and sets `DB.Data[*].LastReader` and `DB.Data[*].LastWriter` to empty. Upon receiving a message (`MSG_TXN`, $\{T_i\}$) from the client, the server initializes acc, the memory digest on the server side. It then sets `RuntimeTraces` to be an empty set. In addition, the server also initializes the list of proofs of memory integrity to be empty. Then, the normal DBMS starts processing the transactions.

Upon a read request on address k from the transaction T_i, the server first runs normal 2PL to fetch the shared read lock on `DB.Data[`k`]`. Then, it infers a partial transaction order (`DB.Data[`k`].LastWriter` $\rightarrow T_i$) by looking at the `LastWriter` field of the data item, and enforces Read-After-Write dependencies. The server adds T_i to the `LastReader`, and a tuple (k, `DB.Data[`k`]`) to the read set of T_i. The server generates a memory integrity proof and appends it to T_i.proofs.

For a write request to address k from the transaction T_i, the server first runs the 2PL logic to fetch the write lock on `DB.Data[`k`]`. Then, the server creates the partial order information by appending (`DB.Data[`k`].LastWriter` $\rightarrow T_i$) and (`DB.Data[`k`].LastReader` $\rightarrow T_i$) to `RuntimeTraces`. It then resets the `LastReader` and `LastWriter` fields, and notifies the memory integrity provider to update the digest with a new write on k with value v.

When a transaction T_i commits, the proofs and read set of T_i are appended to the `proofList` and `readList`, respectively. After all the transactions are finished, the server stores `RuntimeTraces` and the final output of the transaction set, denoted as y. The output y will start with a bit that indicates if any memory integrity check has failed. Also, y can include other information depending on the application. Then, the server calls `TransactionWrapper` with $\{T_i\}$ and `RuntimeTraces`, and sends back the compiled circuit C to the client. Recall that the client will check this circuit with its local circuits corresponding to each transaction. If the check succeeds, the client notifies the key generator to produce keys. Upon receiving the message (`MSG_PKEY`, σ), the server then starts the prover with the `readList`

and `proofList`, as well as the `RuntimeTraces` as auxiliary information to speed up the proving process. When the prover produces the proof π, the server sends it along with the output to the client.

6.2 Client

When Litmus's client sends a batch of transactions $\{T_i\}$ to the server, it also compiles each transaction T_i into a small circuit c_i. The client then waits for the circuit C of the wrapped transaction from the server. The client tries to match the circuit C with its local circuits $\{c_i\}$ by pattern matching. If successful, the client sends C to the key generator and gets the verification key back. Upon receiving the proof π and the commit result y, it checks if the proof π is valid with respect to y (i.e., contains an 1 bit, indicating whether all the memory checks passed), the circuit C, and the transactions $\{T_i\}$, using the verification key τ. If the verification passes, the client accepts the output and the proof; otherwise, the client rejects.

6.3 Verifying Atomicity and Isolation

We contend that Litmus's design guarantees atomicity and isolation. For atomicity, the wrapped transaction in Sec. 6.1.2 chains the transactions sequentially, and so evaluating the circuit implies the effect on the database is equivalent to running the transactions one by one. The DBMS cannot partially execute a transaction if the wrapped transaction evaluates honestly. For isolation, the memory integrity model guarantees that the server can never cheat on the data values. Every read operation will return the latest written value to the designated memory address. Therefore, the result that the client receives is exactly the same as what comes from an honest server in an ideal world, which runs the transactions sequentially.

7 Extension to Multi-Threading

In Sec. 6 we described a single-threaded baseline. In this section, we present an extended design and optimizations to make Litmus practical. Parallelism provided by modern multi-core architecture is crucial to efficient verifiable DBMSs. However, it is non-trivial for VC frameworks to work in parallel. Existing works on VC require that the whole circuit to be evaluated is ready before the protocol starts. Apart from this, they are designed and described for a single-threaded machine because parallelizing the process has limited theoretical interest. There is little work that parallelizes the steps inside the VC framework efficiently outside of [58, 72].

7.1 Concurrency Control

We use the *deterministic reservation* protocol that processes transactions by batches as the CC algorithm for the normal DBMS part [9, 67]. For each batch, it runs two phases in parallel. Alg. 5 shows the pseudocode for the algorithm. It allocates a global array of reservation R and sets it to be infinity. Given a set of transactions $\mathcal{T}$ and a batch size m, the entry function Process first assigns a deterministic and unique priority $T.\rho$ for each transaction T. Then, it processes them in batches. For every batch, it first calls Reserve(T) in parallel on each transaction T in the batch. Any for-loop parallel framework like OpenMP would be sufficient. Then, it computes the return value r_T of Commit(T) for each transaction T in parallel. Lastly, it collects all the transactions with r_T = yes as the non-conflicting batch, adds it to `RuntimeTraces`, and removes these transactions from $\mathcal{T}$.

The Reserve function runs the transaction and collects its read set and write set. Additionally, for every write operation, it also attempts to reserve the key by setting $R[x]$ to be the priority of T_i, if T_i has a higher priority ($T_i.\rho$ is smaller), where x is the key. The Commit function first checks if all the reservations are still valid. If any other transaction overwrites the reservation, the function returns no right away as there is a conflict. Otherwise, it applies the batch to the database. Finally, the function returns yes.

Algorithm 5: Deterministic Reservation

```
   Input: A set of transactions 𝒯 = {T_i} and the batch size m
1  R = [∞,∞,...,∞] ;                                          /* size n */
2  Func Reserve (T_i):
3      Run T_i;
4      Whenever T_i reads DB.Data[x]:
5          T_i.read_set.insert(x);
6          if x is in T_i.WriteVals then
7              return T_i.WriteVals[x]
8          return DB.Data[x]
9      Whenever T_i writes val to DB.Data[x]:
10         T_i.WriteVals.insert(x,val);
11         Atomic do:
12             if T_i.ρ < R[x] then
13                 /* smaller number means higher priority      */
14                 R[x] ← T_i.ρ
15 Func Commit (T_i):
16     /* Check the reservations                                 */
17     for x ∈ T_i.read_set ∪ T_i.WriteVals do
18         if T_i.ρ < R[x] then
19             return no
20     /* Apply the updates                                      */
21     for x,val ∈ T_i.WriteVals do
22         DB.Data[x] ← val
23     return yes
24 Func Process (𝒯,m):
25     Generate priorities T.ρ for every transaction T ∈ 𝒯;
26     do
27         Reset R to be [∞];
28         Take m transactions as 𝒯' from 𝒯;
29         In parallel call Reserve(T) for all T ∈ 𝒯';
30         In parallel compute r_T = Commit(T) for all T ∈ 𝒯';
31         Provide a non-conflicting batch B = {T | r_T = yes};
32         𝒯 ← 𝒯 \ B
33     while 𝒯 ≠ ∅;
```

Deterministic reservation identifies subsets of transactions that can be executed in parallel without conflicts. These transactions are perfect for the transaction wrapper since they can be merged together. And this brings several advantages:*(a) Reduce the number calls to the memory integrity checker.* Exploiting the aggregability of our AD scheme, we can prove and verify a processing batch of non-conflicting transactions with a single proof. This reduces the size of the circuit and the number of auxiliary inputs. Both contribute to reducing the prover computation. *(b) Simplify circuit matching.* The client can locally compute the same interleaving as that on the server thanks to the determinism, and generate fewer circuit pieces. *(c) Flatten the circuit by reducing its depth.* The depth of the circuit is critical for the prover efficiency for some VC frameworks [28]. Moreover, if the transactions are generated from the same template or stored procedure and are similar to each other, we have parallel repetitions of similar structures in the circuit. This repeated structure pattern can be utilized to apply a specially designed proving algorithm [57] that improves the prover efficiency.

When the contention level of the underlying benchmark workload is not high, the improvement in terms of throughput is significant. As we will show in Sec. 8, enabling batching yields a throughput gain of around 10×. Because the CC algorithm is deterministic, the client is able to produce the same batch of transactions if their write targets do not depend on the read values (as in YCSB and the subset

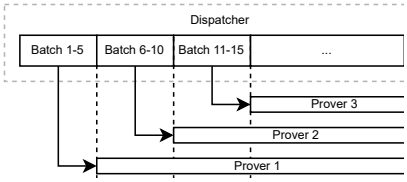


Figure 2: Litmus Pipelining — We start multiple provers with each producing on a number of batches.

of TPC-C we evaluated). If the transactions' writeset depends on the read values, our current design lets the server send the circuit as well as the read sets and write sets to the client. The client can validate the correctness of interleaving by checking whether the transactions in the same batch are non-conflicting or not using a hash table that maps accessed keys to transaction IDs. Alternatively, we can encode the non-conflicting property as a check in the circuit. Given two variables X and Y, the relationship $X \neq Y$ can be encoded using an auxiliary input Z provided by the server s.t. $Z \cdot (X - Y) = 1$. This trick helps the server prove the transactions access different places.

Although Litmus also supports non-deterministic CC algorithms (Sec. 6), we justify our choice of deterministic CC algorithms here. For a non-deterministic algorithm, the client cannot produce the wrapped circuit itself because the interleaving on the server is likely different and therefore, the circuit would be different. The server has to send the circuit to the client for it to perform pattern matching. This adds to communication cost and increases latency. However, batching techniques are still feasible as long as the CC algorithm, which is not necessarily deterministic, can produce batches of non-conflicting transactions. Note that when the CC algorithm is not deterministic and does not work by exploiting non-conflicting batches, like the 2PL baseline in Sec. 8, our multi-core optimization does not apply.

7.2 Pipelining Provers

Following the determinism of the CC algorithm, the transaction wrapper and the memory integrity provider are able to work on their own and do not have to wait for the traces from the normal database part. This not only enables the merging of non-conflicting transactions, but also directly increases parallelism and reduces interaction between components inside the server.

We enable parallel proving without modifying the underlying VC framework. Namely, we *break* the whole circuit into sequential pieces that each consist of multiple transaction batches, and let a single thread work on the proof of a single part of circuit. As shown in Figure 2, we use a dedicated thread (dispatcher) to read runtime traces from the normal database module. Once the dispatcher gathers enough transactions for a circuit piece (e.g., Batch 1-5), it launches a new thread to work on generating the circuit for the transactions and generating the proofs. If all the proofs generated by the prover threads are valid, and the values passing through connecting circuit parts are consistent, the client is convinced of the correctness and semantic properties. Fortunately, the values between connecting parts are only the memory digest and the single bit indicating if all the previous checks are successful. The cost of checking consistency is minimal. Enabling multiple provers yields an extra gain of around 25×.

8 Evaluation Results

We have built a preliminary verifiable database system [66] with prover pipelining and evaluated it against the Yahoo Cloud Serving Benchmark [21] and the TPC-C benchmark [37] OLTP workloads.

The YCSB benchmark mimics a cloud database service with a table of 10 million rows with each row storing 1kB data. In total, the database system hosts 10G in-memory data if not stated otherwise. We also tested a larger YCSB table (see Sec. 8.2). The access pattern of the rows follows the Zipfian distribution with the Zipfian parameter $\theta = 0.6$. Each transaction accesses two rows where each access has a 50% chance to be a *write* operation or otherwise is a *read* operation.

The TPC-C benchmark simulates 64 data warehouses and performs entry orders on them. We include two types of transactions Payment and New Order, which cover around 90% of all the TPC-C transactions per the specification. Moreover, we further assume that customers are selected based on IDs only and the transactions do not insert into the HISTORY table because no transactions read from this table. In this way, the writing targets of transactions do not depend on the read values. Therefore, the server does not have to send the interleaving to the client, which can produce the circuit by itself.

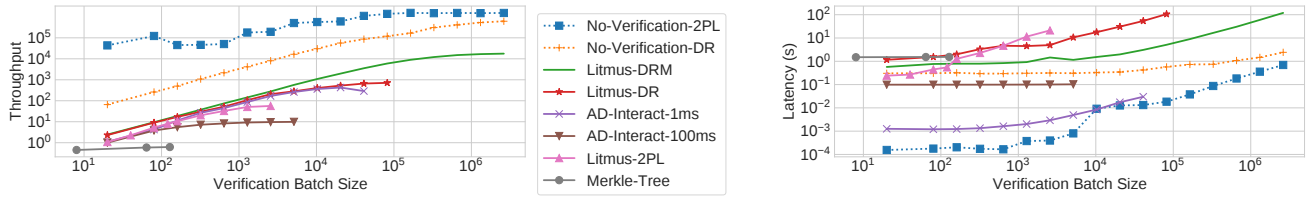
We tested both of the benchmarks with a real DBMS, PostgreSQL, by BenchBase [24]. For YCSB, PostgreSQL has a throughput of 5759 txn/sec. For TPC-C New Order and Payment, PostgreSQL reaches a throughput of 506 txn/sec and 1337 txn/sec, respectively.

We instantiate the VC framework with Pequin [51]. We bypassed the compiler, and hand-wrote the circuits of the transactions and memory integrity checker to guarantee efficiency and determinism. The backend of Pequin is a zero-knowledge succinct non-interactive argument of knowledge (zk-SNARK) protocol that produces the final proof showing that the arithmetic constraints are actually satisfied. Specifically, the backend is built based on the libsnark project, an optimized version of the Pinocchio scheme [44].

Our implementation serves as a proof of concept. It consists of only the server side software as we determined that the server efficiency decides the throughput of the DBMS. We include key generation on the critical path, which can be done in parallel as our CC algorithm is deterministic. For interaction between the client and the server, we simulate a thread sleep of 1 ms or 100 ms. The implementation assumes *division-intractability* for large integers of shape $x+P$, where P is a pre-sampled large prime number. The underlying curve we use in the proving system is BN-128 [4].

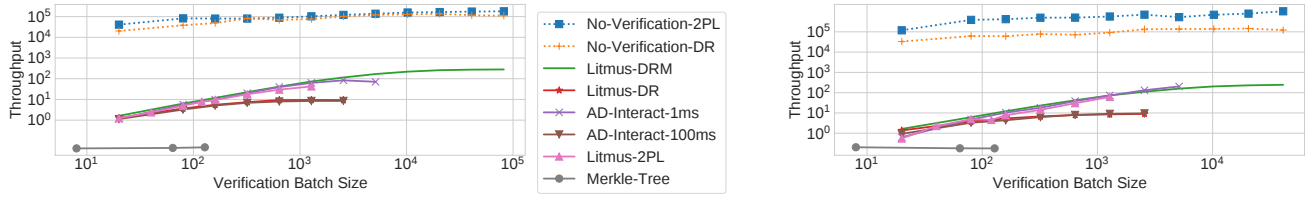
We run Litmus on a machine with two Intel Xeon 5218R CPUs (20 cores per CPU, 2× with hyperthreading). The machine has a 173 GB RAM. We include the following baselines for comparison.

- *Litmus-DRM* is the Litmus system using Deterministic Reservation with Multiple Provers. It uses a processing batch size of 81,920 for the CC algorithm. If not explicitly specified, this baseline uses 4 threads for the normal database component, 1 thread for the runtime tracing, and 75 threads for the provers.
- *Litmus-DR* is the same as Litmus-DRM except with a single prover.
- *Litmus-2PL* is the Litmus system using the 2PL algorithm in Sec. 6.
- *AD-Interact-1ms/100ms* is an interactive baseline that follows the vSQL style interaction between the server and the client for every transaction. The client maintains a digest and performs lookup proof verification locally. By issuing the transactions sequentially, serializability and atomicity are guaranteed. The simulated la-



(a) Throughput **(b)** Latency

Figure 3: Throughput and Latency vs Verification Batch Size - YCSB



(a) New Order **(b)** Payment

Figure 4: Throughput vs Verification Batch Size - TPC-C

tency for the network roundtrip is set to 1 ms to mimic a LAN connection, and 100 ms to mimic a connection across countries (e.g., from Los Angeles to Tokyo), respectively.

- *Merkle Tree* is the folklore approach to realize authenticated data delegation. For every lookup and every update, the server needs to supply a Merkle path consisting of $O(\log n)$ hashes. The client maintains the root of the Merkle tree. We use SHA-256 as the underlying hash algorithm. To make sure the experiment finishes in a reasonable time, we use a smaller table with only 1024 rows.
- *No-Verification-2PL/DR* runs 2PL / Deterministic Reservation at 64 threads without any verification or any logging/traces collection. They serve as performance upper bounds.

8.1 Throughput and Latency

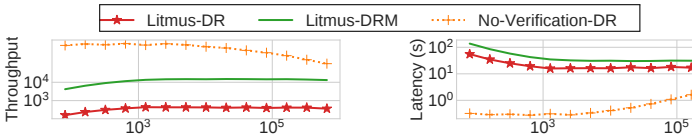
The first experiment evaluates the runtime performance of Litmus by measuring the throughput when the verification batch size changes. We run the baselines with a single verification batch.

For all the baselines, the results in Figure 3a show that throughput increases when the verification batch is larger because the verifiable framework has overhead that grows sublinearly with the number of constraints (namely, the circuit size). When the circuit is larger, the amortized overhead becomes smaller. Litmus-DRM reaches 17,638 txn/sec when the number of transactions in a batch is 2.6m. This is two orders of magnitude slower than the no-verification baseline, and 24.7× faster than Litmus-DR, which uses a single prover thread achieving 714.2 txn/sec at 82k transactions. The peak performance of Litmus-DR is 12.6× faster than Litmus-2PL because it exploits aggregation and transaction parallelism. Litmus-2PL is slower than the deterministic reservation baselines due to less parallelism.

The interactive baselines plateau when the number of transactions is larger than 320. The network latency becomes the bottleneck. Further, the interactive baseline with simulated network latency of 1ms starts to perform worse when the total number of transactions increases. This is due to the computational overhead of the AD scheme. Every single update of the digest invalidates all the proofs. The server has to either compute the witnesses from scratch when needed, or cache the proofs and update them for every digest update. Both methods become more expensive when the number of elements is larger. For the Merkle Tree baseline, the computation overhead of SHA-256 degrades throughput. Our data collection process stops when the lines start to plateau as the slow baselines take an unacceptable amount of time to finish on large workloads.

The second experiment results in Figure 3b show the average latency of the transactions for each of the baselines. The latency covers the time period from the point when the user sends the transaction to the server to the point when the transaction commits and the user receives the proof(s). The latency for Litmus baselines is comparatively higher since the proving algorithm of the VC framework has a significantly long critical path. Among these baselines, Litmus-2PL has the highest latency since all the transactions not only compile into a deep circuit, but also go into a single proof. On the contrary, Litmus-DRM generates a smaller circuit and utilizes multiple provers, with each prover processing a smaller circuit piece. The transactions in those pieces that finish earlier have a smaller latency. The deterministic reservation no-verification baseline starts with a higher latency than the 2PL counterpart, as the CC algorithm needs to wait for a large batch. The latency of the interactive baselines remain stable because the latency is dominated by the simulated network roundtrip when the number of transactions is smaller than 1,000. Then, the latency starts to get dominated by the computation, as we can see clearly for the interactive-1ms baseline.

Figures 4a and 4b show the performance of Litmus and baselines on TPC-C New Order transactions and Payment transactions. We scanned the processing batchsize for deterministic reservation and found that a smaller processing batch is preferable for both TPC-C transactions. The performance of both no-verification baselines are stable for New Order and Payment, because deterministic reservation has a small batch size. For New Order transactions, the peak performance of Litmus-DRM is only 280.6 txn/sec. This is because New Order transactions execute more queries, leading to more cryptographic gates. The results are similar for Payment transactions.



(a) Throughput **(b)** Average Latency

Figure 5: Throughput and Average Latency vs Deterministic Reservation Processing Batch Size

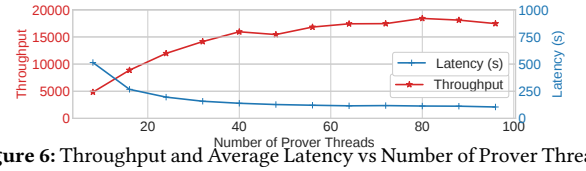


Figure 6: Throughput and Average Latency vs Number of Prover Threads for Litmus-DRM

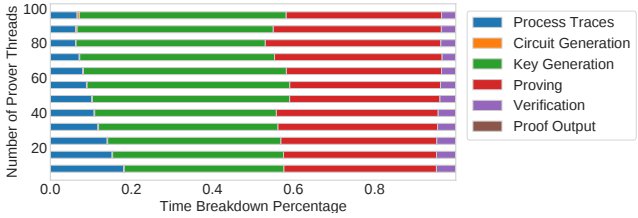


Figure 7: Time Breakdown vs Number of Prover Threads

8.2 Sensitivity Study

We next discuss the sensitivity of Litmus to parameters including processing batch size and the number of prover threads.

Figure 5a shows how the throughput of the deterministic reservation baselines change when the processing batch size changes. The x-axis is the batch size and the y-axis is the throughput. Both of them are in log scale. We can observe that the no-verification baseline remains stable with batch size, because the bottleneck of no-verification is the underlying workload contention. However, for the Litmus baselines, we can see that the throughput grows as the batch size increases due to the larger batch size enabling better exploitation of parallelism and thus the system incurs less prover computational cost. Finally, the throughput decreases because the prover capacity gets saturated while a too large batch harms the performance of CC. We can see a factor of up to 36.2× between the Litmus-DRM and Litmus-DR because of prover pipelining. Figure 5b presents the latency information. When the processing batch size is extremely small, the deterministic reservation CC scheme degrades to a sequential scheduler, incurring significant latency. The latency improves with larger batch sizes, and plateaus when the batch size increases beyond 10^4. The latency of the no-verification baseline increases when the batch size is large, because the too large batch size slows down the synchronized portion of the deterministic reservation algorithm.

Figure 6 shows the throughput and latency of Litmus-DRM when the number of prover threads changes. We see that the throughput scales well up until 40 threads and starts to plateau when there are more than 60 prover threads. As for the average latency, it drops quickly from 514.3 seconds to around 100 seconds when there are more than 40 prover threads.

Figure 7 shows the time breakdown of Litmus-DRM running a verification batch of 2.6m transactions while varying the number of prover threads. We can observe that with a smaller number of prover threads, runtime trace processing (including computing the memory integrity witnesses) takes around 18% of the time. However, as we increase the number of prover threads, key generation and proving gradually take a greater percentage of the time, ending up with 51% and 38%, respectively. The verification takes a modest and stable proportion regardless of the number of prover threads. Since we hand-wrote the circuit, the circuit generation always takes minimal time (not noticeable in the figure). The size of the proofs are constant, namely, 312 bytes per prover thread and 30 KB in total. It takes the client around 300 sec to verify each proof. The key pair has a large size, but we can use universal VC schemes (Sec. 9) where key pairs are not necessary.

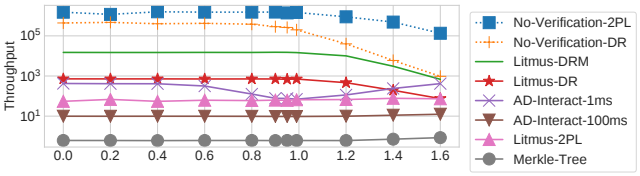


Figure 8: Throughput vs Contention Level

Table Size	10G	20G	40G	80G
Performance (txn/s)	17538	16394	14909	12818

Figure 9: Performance of Litmus vs Table Size

Figure 8 shows how the throughput changes with the contention level of the workload. We observe that all three deterministic reservation baselines are impacted heavily. Since a higher contention level makes more transactions conflict with each other, each round of deterministic reservation produces a smaller non-conflicting batch. Therefore, it needs more rounds to finish processing the transactions. This directly affects the performance of Litmus with deterministic reservation as it cannot benefit from aggregating a large number of transactions. Note that, the proving capability depends only on the circuit size, which does not change with the contention level. When the contention level is high, the performance of Litmus is close to the no-verification baseline with DR because the performance is bounded by the CC algorithm. Comparatively, the 2PL baselines are less sensitive to the contention level. The interactive baselines performance increases since a higher contention level brings better cache utilization.

Figure 9 shows how Litmus performs for varying table sizes. We observe that throughput decreases slowly as the table size doubles. However, we ran out of memory at a 160G YCSB table, as our machine only has 173 GB RAM and we need to allocate space for the traces. We can project that Litmus has promising performance for even larger in-memory databases.

8.3 Comparison with Elle

To understand how Litmus performs compared to alternatives, we evaluated Elle [35] with our codebase. Elle verifies serializability by inferring from the transaction read values and write values. Specifically, it changes all the write operations into list appends to get a history of value versions. It looks for inconsistency between transaction commit orders and the actual value histories.

We ran the no-verification baseline with the YCSB benchmark and

stored the list appending traces into the RAM disk to avoid performance impacts from storage I/O. Elle reads and analyzes the traces, and outputs the result. For fairness, we exclude the time of Elle reading the traces, and include only the actual analyzing time. With 3.5m txns[3], Elle spent 576 sec, reaching a throughput of ~ 5.5k txn/sec. This is at the same level as reported in [35].

Both Cobra [55] and Elle are trace-based, which means they must expose the trace to a trusted entity (a strong verifier able to infer the dependency graph) or the client itself. In contrast, Litmus's client only needs to obtain a single constant-sized proof and verify it in constant time. Moreover, Elle requires changes to the table schema (replacing fixed-length values into var-length lists) to make accurate inferences, and is designed to perform offline tests for software bugs, but not for a continuous/growing history. In the case where an active adversary is involved, it might exploit the incompleteness of Elle and perform violations with an irrecoverable history. Different from Elle, Litmus checks correctness properties on the fly and provides protection on the exact transactions submitted by the client. Meanwhile, Elle relies on the server to honestly provide a full history, and the client to run an inference algorithm to look for serializability violations.

9 Discussion

We next discuss some of our design insights and future directions. As mentioned in Sec. 2, the motivation of Litmus is untrusted cloud DBaaS services. One use case is critical cloud computing scenarios where mistakes could have catastrophic consequences. Examples include financial institutions and criminal records. Our design supports large databases because the digest is constant-sized and verification takes only constant time. The prover running time depends only on the complexity of transaction logic, but not directly on the data size.

Why Cryptography: Compared to the interactive baseline, Litmus lets the client delegate the "interactive verifier" onto the server. Cryptography ensures that the verifier is working correctly. This delegation enables (1) aggressive exploitation of parallelism and aggregability because the server now has the freedom to re-shape the verifier circuit for better performance, and the sequential order is not necessarily materialized within the server (e.g., deterministic reservation only produces batch-by-batch order); (2) network communication becomes internal data exchange, saving significant overhead; and (3) lightweight clients, which do not need to perform heavy transaction replays, (e.g., scanning the whole database requires sending the whole database to the client in [32]).

Universal Verifiable Computation: We currently use Groth16 [29] to instantiate the verifiable computation primitive. However, this comes with a trade-off regarding the trusted setup. Namely, we assume a trusted third-party needs to know the circuit before the proving starts. In our evaluation, we only implemented the server side that performs the trusted setup for the client; there is a security issue because the server might generate a malicious setup that allows it to cheat. This is not a problem for the situations in our evaluation, where the transactions' logic (i.e., circuits) is fixed and the setup only needs to be done once.

However, if the transactions are not generated from a fixed template, the client has to generate the setup for every new circuit. This is computationally expensive, violating our assumption that the client is lightweight. A better alternative is to replace the instantiation with a universal verifiable computation framework like Plonk [26], whose setup is circuit-independent.

Real-time Transactions and Hybrid Approach: Our current design has a fundamental issue of having long latency. Due to the current status of cryptographic tools, the latency of verification is inevitable for batched verification. To address this, we propose two solutions: (1) we can include a hybrid mode, where Litmus can switch between batch verification and interactive verification in real-time. The memory digest of these two modes are compatible. Whenever a client needs faster responses, it can mark the transactions so that the DBMS executes them in the interactive mode that has a lower throughput because it cannot take advantage of aggregations, but it will have a lower latency. (2) We can decouple the transaction results and the cryptographic proof, i.e., Litmus can issue the results to the client as soon as they are ready. The client receives the proof from the server asynchronously. There also could be special transactions that check for application invariant properties.

Consistency and Durability: A *consistent* transaction changes the database only in certain ways. In a bank system, the rule could be that the sum of all balances remains the same after a transfer transaction. To verify consistency, we apply similar methods, but specializing the memory integrity checker into customized checkers. *Durability* guarantees that once a transaction is committed, it will remain committed even if the system crashes. To provide verifiable durability, we have to rely on external shared storage because there is no way to guarantee that the server has written to the disk without letting the client have access to it. One approach would be to design new hard drives with secure enclaves [22, 38]. We believe recent advances in in-storage computation that enable data storage to perform programmable tasks [61] may be promising.

10 Conclusion

We proposed a potential solution to data integrity issues and attacks on transaction semantic properties in database outsourcing. The Litmus system not only prevents a malicious server from returning wrong results, but also provides an answer to 'ACIDRain attacks' [65] by preventing attackers from exploiting isolation levels.

Acknowledgments

This work was supported (in part) by the National Science Foundation (IIS-1846158, SPX-1822920, SPX-1822933, 2028818), Google DAPA Research Grants, and the Alfred P. Sloan Research Fellowship program.

References

[1] Amazon AWS. 2017. *AWS Cloud Database Leak: California Voter Data Exposed, Held for Ransom - MSSP Alert.* Retrieved 09/05/2020 from https://www.msspalert.com/cybersecurity-news/aws-database-leak-california-voter-data-exposed-held-for-ransom/

[2] Amazon AWS. 2020. *State & Local Government.* Retrieved 09/05/2020 from https://aws.amazon.com/stateandlocal/

[3] Niko Barić and Birgit Pfitzmann. 1997. Collision-free accumulators and fail-stop signature schemes without trees. In *International conference on the theory and applications of cryptographic techniques*. Springer, 480–494.

[4] Paulo SLM Barreto and Michael Naehrig. 2005. Pairing-friendly elliptic curves of prime order. In *International Workshop on Selected Areas in Cryptography*. Springer, 319–331.

[3] We could not push it further because Elle exhausted our server's 173 GB memory.

[5] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. 2013. SNARKs for C: Verifying program executions succinctly and in zero knowledge. In *Annual Cryptology Conference*. 90–108.

[6] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. 2014. Succinct non-interactive zero knowledge for a von Neumann architecture. In *USENIX Security Symposium*. 781–796.

[7] Philip A Bernstein and Nathan Goodman. 1983. Multiversion concurrency control—theory and algorithms. *ACM Transactions on Database Systems (TODS)* 8, 4 (1983), 465–483.

[8] Nir Bitansky, Alessandro Chiesa, Yuval Ishai, Omer Paneth, and Rafail Ostrovsky. 2013. Succinct non-interactive arguments via linear interactive proofs. In *Theory of Cryptography Conference*. 315–333.

[9] Guy E Blelloch, Jeremy T Fineman, Phillip B Gibbons, and Julian Shun. 2012. Internally deterministic parallel algorithms can be fast. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*. 181–192.

[10] Andrew J Blumberg, Justin Thaler, Victor Vu, and Michael Walfish. 2014. Verifiable computation using multiple provers. *IACR Cryptology ePrint Archive* 2014 (2014), 846.

[11] Dan Boneh, Benedikt Bünz, and Ben Fisch. 2019. Batching Techniques for Accumulators with Applications to IOPs and Stateless Blockchains. In *Advances in Cryptology – CRYPTO 2019*, Alexandra Boldyreva and Daniele Micciancio (Eds.). 561–586.

[12] Dan Boneh, Benedikt Bünz, and Ben Fisch. 2019. Batching techniques for accumulators with applications to iops and stateless blockchains. In *Annual International Cryptology Conference*. Springer, 561–586.

[13] D. Boneh, E. Kushilevitz, R. Ostrovsky, and W. E. Skeith. 2007. Public Key Encryption That Allows PIR Queries. In *CRYPTO (LNCS, Vol. 4622)*. pp.50–67.

[14] Benjamin Braun, Ariel J Feldman, Zuocheng Ren, Srinath Setty, Andrew J Blumberg, and Michael Walfish. 2013. Verifying computations with state. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 341–357.

[15] Thomas Brewster. 2015. *191 Million US Voter Registration Records Leaked In Mystery Database.* Retrieved 09/05/2020 from https://www.forbes.com/sites/thomasbrewster/2015/12/28/us-voter-database-leak/#171b25735b98

[16] Lucas Brutschy, Dimitar Dimitrov, Peter Müller, and Martin Vechev. 2017. Serializability for eventual consistency: criterion, analysis, and applications. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. 458–472.

[17] Niklas Büscher and Stefan Katzenbeisser. 2017. *Compiling ANSI-C Code into Boolean Circuits.* 15–28.

[18] Matteo Campanelli, Dario Fiore, Nicola Greco, Dimitris Kolonelos, and Luca Nizzardo. 2020. Vector Commitment Techniques and Applications to Verifiable Decentralized Storage. *IACR Cryptol. ePrint Arch.* 2020 (2020), 149.

[19] Alessandro Chiesa, Eran Tromer, and Madars Virza. 2015. Cluster computing in zero knowledge. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. 371–403.

[20] Dwaine Clarke, Srinivas Devadas, Marten Van Dijk, Blaise Gassend, and G Edward Suh. 2003. Incremental multiset hash functions and their application to memory integrity checking. In *International conference on the theory and application of cryptology and information security*. Springer, 188–207.

[21] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. 143–154.

[22] Victor Costan, Ilia Lebedev, and Srinivas Devadas. 2016. Sanctum: Minimal hardware extensions for strong software isolation. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*. 857–874.

[23] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. 2013. Oltp-bench: An extensible testbed for benchmarking relational databases. *Proceedings of the VLDB Endowment* 7, 4 (2013), 277–288.

[24] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudré-Mauroux. 2013. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *PVLDB* 7, 4 (2013), 277–288. http://www.vldb.org/pvldb/vol7/p277-difallah.pdf

[25] Nelly Fazio and Antonio Nicolosi. 2002. Cryptographic accumulators: Definitions, constructions and applications. *Paper written for course at New York University: www. cs. nyu. edu/nicolosi/papers/accumulators. pdf* (2002).

[26] Ariel Gabizon, Zachary J Williamson, and Oana Ciobotaru. 2019. PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge. *IACR Cryptol. ePrint Arch.* 2019 (2019), 953.

[27] Zahra Ghodsi, Tianyu Gu, and Siddharth Garg. 2017. Safetynets: Verifiable execution of deep neural networks on an untrusted cloud. In *Advances in Neural Information Processing Systems*. 4672–4681.

[28] Shafi Goldwasser, Yael Tauman Kalai, and Guy N Rothblum. 2015. Delegating computation: interactive proofs for muggles. *Journal of the ACM (JACM)* 62, 4 (2015), 1–64.

[29] Jens Groth. 2016. On the size of pairing-based non-interactive arguments. In *Annual international conference on the theory and applications of cryptographic techniques*. Springer, 305–326.

[30] Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. 2007. PeerReview: Practical accountability for distributed systems. *ACM SIGOPS operating systems review* 41, 6 (2007), 175–188.

[31] Christian Hammer, Julian Dolby, Mandana Vaziri, and Frank Tip. 2008. *Dynamic Detection of Atomic-Set-Serializability Violations.* Association for Computing Machinery, New York, NY, USA, 231–240. https://doi.org/10.1145/1368088.1368120

[32] Rohit Jain and Sunil Prabhakar. 2013. Trustworthy data from untrusted databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 529–540.

[33] Jepsen. 2020. *Analyses.* Retrieved 09/10/2020 from https://jepsen.io/analyses

[34]]gretchen Kyle Kingsbury. [n. d.]. Gretchen. https://github.com/aphyr/gretchen

[35] Kyle Kingsbury. 2019. Black-Box Serializability Verification.

[36] Kyle Kingsbury and Peter Alvaro. 2020. Elle: Inferring Isolation Anomalies from Experimental Observations. *Proc. VLDB Endow.* 14, 3 (Nov. 2020), 268–280.

[37] W Kohler, A Shah, and F Raab. 1991. Overview of TPC Benchmark C: The Order-Entry Benchmark. *Transaction Processing Performance Council, Technical Report* (1991).

[38] Ilia Lebedev, Kyle Hogan, and Srinivas Devadas. 2018. Secure boot and remote attestation in the sanctum processor. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. IEEE, 46–60.

[39] Linode. 2022. *Recovering from a System Compromise | Linode.* Retrieved 09/05/2020 from https://www.linode.com/docs/security/recovering-from-a-system-compromise/

[40] Benjamin Mood, Debayan Gupta, Henry Carter, Kevin Butler, and Patrick Traynor. 2016. Frigate: A validated, extensible, and efficient compiler and interpreter for secure computation. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 112–127.

[41] Kartik Nagar and Suresh Jagannathan. 2018. Automated detection of serializability violations under weak consistency. *arXiv preprint arXiv:1806.08416* (2018).

[42] Alex Ozdemir, Riad Wahby, Barry Whitehat, and Dan Boneh. 2020. Scaling Verifiable Computation Using Efficient Set Accumulators. In *USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2075–2092.

[43] Christos H Papadimitriou. 1979. The serializability of concurrent database updates. *Journal of the ACM (JACM)* 26, 4 (1979), 631–653.

[44] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. 2013. Pinocchio: Nearly practical verifiable computation. In *2013 IEEE Symposium on Security and Privacy*. 238–252.

[45] Henry C Pocklington. 1914. The determination of the prime or composite nature of large numbers by Fermat's theorem. *Proc. Cambridge Philosophical Society, 1914* 18 (1914), 29–30.

[46] Raghu Ramakrishnan and Johannes Gehrke. 2000. *Database management systems.* McGraw Hill.

[47] Noga Ron-Zewi and Ron D Rothblum. 2021. Proving as Fast as Computing: Succinct Arguments with Constant Prover Overhead. *Cryptology ePrint Archive* (2021).

[48] Srinath Setty, Sebastian Angel, Trinabh Gupta, and Jonathan Lee. 2018. Proving the correct execution of concurrent services in zero-knowledge. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 339–356.

[49] Srinath Setty, Benjamin Braun, Victor Vu, Andrew J Blumberg, Bryan Parno, and Michael Walfish. 2013. Resolving the conflict between generality and plausibility in verified computation. In *Proceedings of the 8th ACM European Conference on Computer Systems*. 71–84.

[50] Srinath Setty, Victor Vu, Nikhil Panpalia, Benjamin Braun, Andrew J Blumberg, and Michael Walfish. 2012. Taking proof-based verified computation a few steps closer to practicality. In *USENIX Security Symposium*. 253–268.

[51] Srinath TV Setty, Richard McPherson, Andrew J Blumberg, and Michael Walfish. 2012. Making argument systems for outsourced computation practical (sometimes).. In *NDSS*, Vol. 1. 17.

[52] Arnab Sinha and Sharad Malik. 2010. Runtime checking of serializability in software transactional memory. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*. 1–12.

[53] Emin Gün Sirer. 2014. *NoSQL Meets Bitcoin and Brings Down Two Exchanges: The Story of Flexcoin and Poloniex.* Retrieved 09/10/2020 from https://hackingdistributed.com/2014/04/06/another-one-bites-the-dust-flexcoin/

[54] Cheng Tan, Lingfan Yu, Joshua B Leners, and Michael Walfish. 2017. The efficient server audit problem, deduplicated re-execution, and the web. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 546–564.

[55] Cheng Tan, Changgeng Zhao, Shuai Mu, and Michael Walfish. 2020. Cobra: Making transactional key-value stores verifiably serializable. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*. 63–80.

[56] Cheng Tan, Changgeng Zhao, Shuai Mu, and Michael Walfish. 2020. Making Cloud Key-Value Databases Verifiably Serializable. *USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2020).

[57] Justin Thaler. 2013. Time-optimal interactive proofs for circuit evaluation. In *Annual Cryptology Conference*. Springer, 71–89.

[58] Justin Thaler, Mike Roberts, Michael Mitzenmacher, and Hanspeter Pfister. 2012. Verifiable Computation with Massively Parallel Interactive Proofs.. In *HotCloud*.

[59] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J Abadi. 2012. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. 1–12.

[60] Alin Tomescu, Yu Xia, and Zachary Newman. 2020. Authenticated Dictionar-

ies with Cross-Incremental Proof (Dis) aggregation. *IACR Cryptol. ePrint Arch.* 2020 (2020), 1239.

[61] Mahdi Torabzadehkashi, Siavash Rezaei, Vladimir Alves, and Nader Bagherzadeh. 2018. CompStor: An In-storage Computation Platform for Scalable Distributed Processing. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 1260–1267. https://doi.org/10.1109/IPDPSW.2018.00195

[62] Riad S Wahby, Srinath TV Setty, Zuocheng Ren, Andrew J Blumberg, and Michael Walfish. 2015. Efficient RAM and control flow in verifiable outsourced computation.. In *Annual Network and Distributed System Security Symposium (NDSS)*.

[63] Riad S Wahby, Ioanna Tzialla, Abhi Shelat, Justin Thaler, and Michael Walfish. 2018. Doubly-efficient zkSNARKs without trusted setup. In *2018 IEEE Symposium on Security and Privacy (SP)*. 926–943.

[64] Haixin Wang, Cheng Xu, Ce Zhang, and Jianliang Xu. 2020. vChain: A Blockchain System Ensuring Query Integrity. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2693–2696.

[65] Todd Warszawski and Peter Bailis. 2017. Acidrain: Concurrency-related attacks on database-backed web applications. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 5–20.

[66] Yu Xia. 2022. *LitmusDB Preliminary Build.* Retrieved 03/30/2022 from https://github.com/yuxiamit/LitmusDB

[67] Yu Xia, Xiangyao Yu, William Moses, Julian Shun, and Srinivas Devadas. 2019. LiTM: A Lightweight Deterministic Software Transactional Memory System. In *Proceedings of the 10th International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM)*. 1–10.

[68] Cheng Xu, Jianliang Xu, Haibo Hu, and Man Ho Au. 2018. When query authentication meets fine-grained access control: A zero-knowledge approach. In *Proceedings of the 2018 International Conference on Management of Data*. 147–162.

[69] Cheng Xu, Ce Zhang, and Jianliang Xu. 2019. vchain: Enabling verifiable boolean range queries over blockchain databases. In *Proceedings of the 2019 international conference on management of data*. 141–158.

[70] Min Xu, Rastislav Bodík, and Mark D Hill. 2005. A serializability violation detector for shared-memory server programs. *ACM Sigplan Notices* 40, 6 (2005), 1–14.

[71] Kamal Zellag and Bettina Kemme. 2014. Consistency anomalies in multi-tier architectures: automatic detection and prevention. *The VLDB Journal* 23, 1 (2014), 147–172.

[72] William Zhang and Yu Xia. 2021. Hydra: Succinct Fully Pipelineable Interactive Arguments of Knowledge. Cryptology ePrint Archive, Report 2021/641. https://eprint.iacr.org/2021/641.

[73] Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. 2017. vSQL: Verifying Arbitrary SQL queries over Dynamic Outsourced Databases. In *2017 IEEE Symposium on Security and Privacy (SP)*. 863–880.

[74] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. 2015. IntegriDB: Verifiable SQL for outsourced databases. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 1480–1491.