Towards a Cryptographically Verifiable Database Management System

by

Yu Xia

B.S., Tsinghua University (2016)
S.M., Massachusetts Institute of Technology (2018)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2022

Author .....
Department of Electrical Engineering and Computer Science
August 8, 2022

Certified by .....
Srinivas Devadas
Edwin Sibley Webster Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Towards a Cryptographically Verifiable Database Management System

by

Yu Xia

Submitted to the Department of Electrical Engineering and Computer Science
on August 8, 2022, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

Database-as-a-Service (DBaaS), like Amazon Web Service Redshift and Microsoft Azure SQL Database, is becoming increasingly popular. These services provide high performance and on-demand elasticity without heavy maintenance costs. However, as with all online applications, DBaaS is prone to malicious attacks ranging from server compromises to cheating providers.

We believe that database security is more than just data privacy.

Existing secure DBMSs focus on the 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. A hacker compromising the server gains complete control of the operating system. The hacker can tamper with the data, perform arbitrary computation, violate transaction properties, or return wrong results to the client to pursue external incentives like financial benefits. Protecting data privacy does not eliminate all the incentives to initiate attacks. For example, the hacker can short the stock price of the data owner while forcing the server to run wrong transactions and return incorrect results, potentially creating business chaos. Besides the correctness of the transactions and results, ACID properties are also critical. For example, two cryptocurrency exchanges went bankrupt due to hackers double-spending their coins through isolation-level attacks.

To address this issue, this dissertation presents Litmus, a database management system 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. We show that under certain cryptographic assumptions, Litmus can process up to thousands of transactions per second (txn/s) *verifiably*. Our results show a promising practical direction considering that PayPal runs on average 115 txn/s and VISA 2000-4000 txn/s. The proof is about tens of kilobytes per verification batch and verifies with a constant time of a few hundred seconds. Moreover, Litmus

can extend to verify consistency as well.

Thesis Supervisor: Srinivas Devadas
Title: Edwin Sibley Webster Professor of Electrical Engineering and Computer Science

Acknowledgments

First and foremost, I would like to thank my advisor, Srini Devadas, for his dedicated support in my research freedom and years of kindness and patience. Srini encouraged me to work on anything I found interesting. During my six years of graduate studies, I mainly worked on database management systems and applied cryptography, two distant fields with different focuses, mindsets, and paradigms. My effort in these two areas has been separated for years until this thesis, which lies in the disciplinary area of both. It will never be possible without Srini's unique technological breadth. Srini's great aesthetics in research shaped my taste. His craftsmanship spirit, from fine-grained details to overall structure, sets a standard for my future work.

I am grateful to the rest of my committee members, Andrew Pavlo and Henry Corrigan-Gibbs. As my long-term collaborator and database Jedi, Andy generously shares his expertise and insights in the database area and offers help throughout the projects. Henry proposed the initial idea of the verifiable database formalization and provided numerous suggestions and help on the cryptography side. I am also grateful to Charles Leiserson for his advice on learning and life in general.

Next, I would like to thank Xiangyao Yu. He brought up the question of defending transaction semantic attacks to me and led me to the ACIDRain paper by Warszawski et al. [162]. During my first few years at MIT, he guided me through several database projects, and I always learned new things from discussions with him.

I was fortunate to have many wonderful collaborators and friends. This thesis is based on joint work with Alin Tomescu, Zachary Newman, William Moses, Julian Shun, Matthew Butrovich, and William Zhang. Beyond this thesis, I want to thank Jun Wan, Shumo Chu, Zhenfei Zhang, Andrew Miller, Zhuolun Xiang, Ling Ren, Elaine Shi, Dawn Song, and Yael Kalai for all the collaboration and discussion.

I would specially thank my friend Yifan Xu and Ruisi Su, as well as their parents, for taking good care of me when I was in deep trouble early this year. Most of the thesis writing was done when I was living in their places. My deepest gratitude goes to my parents. They are always there whenever I need them.

Contents

List of Figures

List of Tables

Chapter 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 [9]. In addition, financial institutions delegate data to the cloud to conduct data analysis, dispute management, and fraud investigation [114].

Such outsourcing can reduce energy [106, 89], hardware, and labor costs [163], but also exposes an organization to data-integrity risks [90]. An attacker that breaches the DBMS can tamper with its data. For example, in the case of a voter-registration database, an attacker could selectively modify registration data for voters from one political party. As another example, falsifying financial account data can result in significant asset loss. An even more problematic scenario is that 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 backups. Unfortunately, there is ample evidence that such breaches often occur [8, 36] and that cleaning up from them is costly [102].

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 [162] 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 [67]. 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 exam-

ple, running the TPC-C benchmark at a lower isolation level can yield 2.5× better throughput compared to that with serializability [57]. Such ACID failures are commonplace, even in widely deployed database systems [85, 92], and they sometimes even lead to business bankruptcy [137].

Existing solutions test whether a database provides serializability by analyzing the log history [140] of the transactions or the internal scheduler choices of the concurrency control algorithms [37, 78, 112, 176, 136]. They either include an independent, trusted verifier that is powerful enough to run SAT/SMT solvers and report the results to the clients or assume the client itself is capable of handling the analysis. Analogous to fuzzing techniques in detecting bugs, works like Jepsen [93] generate test suites and analyze the results and the log records to protect the users against faulty DBMS servers. Jepsen operates in two modes. The first one offers incomplete protection, which does not always catch serializability violations, but it does not change the data schema. The second one provides complete protection against property violations but requires the DBMS to support list operations. It changes the database schema — fixed-length numbers and values become variable-length lists — to embed history versions of data items into the lists. If the test suite analysis passes, the system is deemed secure. However, successful offline tests do not protect the real-time requests from the client. If the source code of the database is available, formal verification can also yield a proof of serializability guarantee [105]. However, formal methods require massive human effort to build logic deductions and guide the theorem proving [124], and often results do not generalize. Moreover, the remote server might run tampered codes instead of the formally verified version.

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 increases), 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 — the proof is constant-sized, and the verification takes constant time. 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, equivalent to the no outsourcing scenario. Hence, verifiable DBMSs promise that they can give the same level of integrity protection as a local database with the cost savings and convenience of the cloud. The reader might wonder why we do not replicate the database to a number of providers and check that they all return the same answers. We note that DBMSs are long-running services in the real world. With enough time, an attacker can secretly compromise all the servers before executing the attack — letting all the servers return the same wrong result. Therefore, we do not trust any of the servers in this work. Lastly, even if all the servers are honest, the results might differ for different servers when the concurrency control algorithm is not deterministic (in fact, due to performance reasons, most concurrency control algorithms are not deterministic). Therefore, such a replicate-and-check approach suffers from significant false positive and false negative cases.

We target the use case of critical cloud computing scenarios where mistakes could have catastrophic consequences. Examples include financial institutions and criminal records. For now, Litmus assumes a single server interacting with a single client. In the DBaaS setting, the client is the organization that delegates the database to the cloud and interacts with the server on behalf of many clients. Compared to a local cluster, a cloud service, even with the verification overhead, can provide both elasticity and robustness at a lower cost. The limitations of Litmus mainly lie in the

following parts: (a) the batch-based design incurs long verification latency; (b) the optimizations in Chapter 8 do not apply to general transactions. (c) Litmus assumes a single client interacting with a single server, although the client could be a delegation of many users. We defer detailed discussion of limitations to Section 12.3. Compared to a previous one-transaction-by-another verification scheme like vSQL [179], Litmus focuses on low-level data interfaces and exploits parallelism in the workload by processing large transaction batches.

We evaluated a proof-of-concept prototype of Litmus with YCSB and TPC-C workloads. Litmus with multiple parallel provers is able to verifiably process up to 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].

1.1 Contribution Overview

In this dissertation, we make the following contributions.

- We propose formal definitions of database properties in the language of cryptography.

- We present Litmus, a practical and general verifiable database system that provides cryptographic guarantees on data integrity, execution correctness, and transaction semantic properties. Specifically, Litmus protects atomicity and isolation level properties. Using Litmus blocks the type of attacks described in ACIDRain [162].

- We propose, and use in Litmus, a lightweight authenticated dictionary (AD) scheme based on RSA accumulators that supports key non-existence proofs. We further generalize the AD scheme to a new cryptographic primitive, *wild-*

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

card accumulator, and show two constructions. These contributions 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.

- We present Hydra, a parallel and pipelineable interactive argument of knowledge. Hydra enables parallel proving of a Goldwasser-Kalai-Rothblum (GKR) verifiable computation protocol. This protocol can potentially improve the performance of verifiable databases on modern parallel architectures.

The vast majority of this thesis is an extended version of the Litmus conference paper [165] [2]. Chapter 7 comes from the LiTM project [166]. Section 5.3 is co-authored with Alin Tomescu and Zachary Newman. Chapter 13 comes from the Hydra project [177] [2], with edits to keep the notations consistent with other parts of the thesis.

[2]Licensed under Creative Commons Attribution International 4.0 License

Chapter 2

Preliminaries and Related Topics

This chapter first helps the readers with background necessary for transactions and the ACID properties. Then, it introduces the goals for our verifiable database and where we must extend existing work to achieve them. Litmus provides verifiable atomicity and isolation levels. In Section 3.3 and Section 11.2, we will discuss how to support consistency as well.

2.1 Transactions and ACID Properties

A transaction is a sequence of operations (e.g., read, write, insert, or delete) that a client sends to a database to complete some tasks. For example, in a banking system, a typical transfer transaction moving 100 USD from an account A to another account B looks like Figure 2-1. The transaction first reads the balance from Account A. Then, if Account A has no less than 100 USD, the transaction reads the balance of Account B and performs the transfer.

Modern databases process transactions in parallel. Transactions provide an easy and natural programming paradigm for concurrent data processing [60]. The major benefits of this paradigm are the ACID properties guaranteed by database management systems. They refer to the following four properties [121]:

- **Atomicity**. Either all or no operations of a transaction complete in the database (i.e., *all or nothing*).

```
1 balance_A ← Read(A);
2 if balance ≥ 100 then
3 |   balance_B ← Read(B);
4 |   Write(A, balance_A − 100);
5 |   Write(B, balance_B + 100);
```

Figure 2-1: Example Transfer Transaction.

If the transaction only executes partially, the data state may become something that was not intended by the programmer. For the example in Figure 2-1, if execution stops after finishing Line 4, Account B will not get the money transferred, and therefore, the bank loses 100 USD permanently, producing an inconsistent state.

- **Consistency**. Any given database transaction must obey semantic invariants, including constraints, cascades, and triggers. A transaction cannot leave the database in an invalid state.

  In our bank account example, one possible consistency definition would be "the sum of all the accounts remains the same after every transfer transaction."

- **Isolation**. An isolation level defines when a transaction's effects can be observed by another concurrent transaction. For example, *serializability*, the gold standard isolation level, requires concurrent execution of transactions to produce the same results as *some* sequential execution. Verifying isolation is more complicated than the other three properties since it involves coordinating multiple transactions instead of a single one.

  Violating the isolation level can result in significant consequences. In the bank account example, consider two transfer transactions running concurrently. Both transactions transfer 100 USD from Account A to Account B. Initially, A has 200 USD and B has 500 USD. Consider the two interleavings in Figure 2-2. Interleaving 1 first runs $T1$ and then $T2$, resulting A's balance being 0 and B's being 700 USD. Interleaving 2, however, runs the first half of $T1$, then the first

```
   // A:200, B:500
1  Read(A) → 200 // T1
2  Read(B) → 500 // T1
3  Write(A, 100) // T1
4  Write(B, 600) // T1
5  Read(A) → 100 // T2
6  Read(B) → 600 // T2
7  Write(A, 0) // T2
8  Write(B, 700) // T2
   // A:0, B:700
```

(a) Interleaving 1

```
   // A:200, B:500
1  Read(A) → 200 // T1
2  Read(B) → 500 // T1
3  Read(A) → 200 // T2
4  Write(A, 100) // T1
5  Write(B, 600) // T1
6  Read(B) → 600 // T2
7  Write(A, 100) // T2
8  Write(B, 700) // T2
   // A:100, B:700
```

(b) Interleaving 2

Figure 2-2: Transfer Transaction Interleavings.

line of $T2$, followed by the second half of $T1$, and finally the rest of $T2$. The resulting database state is A's balance being 100 USD and B's being 700 USD. This result is apparently wrong, since Account A is charged only 100 USD, while account B is credited with 200 USD.

Interleaving 2 violates serializability because there is no serial order of $T1$ and $T2$ that results in the state of A being 100 and B being 700. An attacker can exploit the serializability compromise (e.g., by forcing the server to run Interleaving 2) to "double-spend" the money of Account A. Vulnerabilities similar to this example have resulted in the bankruptcy of cryptocurrency exchanges [137, 162]. Warszawski et al. comprehensively studied this type of attacks and named them *ACIDRain* attacks [162].

Besides serializability, there are other isolation levels, including repeatable read, read committed, and read uncommitted. We will discuss alternative isolation levels in Chapter 3 and Chapter 11.

- **Durability**. Effects of committed transactions will survive permanently, even if the system crashes. Durability requires each committed transaction to either update or log to persistent storage.

  In the bank account system example, if durability is compromised, a hacker can potentially first commit a transfer to a car dealer, drive the car away, and then

crash the server before the effects become permanent. The transfer transaction is then "rolled back," and the hacker effectively paid nothing.

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 [25, 117]. We choose to utilize cryptography to force the DBMS to provide a proof of honest behavior. 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. In Chapter 11, we will revisit discussions on consistency and durability.

2.2 Challenges

There are many challenges in designing such a verifiable DBMS:

1. **It is not clear how to provide short proofs of inter-transaction properties like serializability without giving up parallelism.** There are exponentially many valid transaction interleavings among exponentially many possible transaction interleavings. Theoretically, sending the logic of the whole multi-core 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 they are multi-threaded. Moreover, cryptography has special requirements on the input logic, e.g., arithmetic circuits on finite fields. We will explain cryptographic circuits in detail in Section 2.4. Lastly, the cryptographic formalization of inter-transaction is not straightforward.

2. **Existing general cryptographic tools are (concretely) computationally heavyweight, posing a practicality challenge.** Empirically, general-purpose cryptographic constructions incur concretely large computation overhead. For example, zero-knowledge proving systems sometimes costs more than 1,000× compared to native computation [48].

```
   // A:200, B:500
1  Read(A) → 200 // T1
2  Read(B) → 500 // T1
3  Write(A, 100) // T1
4  Write(B, 600) // T1
5  Read(A) → 100 // T2
6  Read(B) → 600 // T2
7  Write(A, 0) // T2
8  Write(B, 700) // T2
   // A:0, B:700
```

(a) Interleaving 1

```
   // A:200, B:500
1  Read(A) → 200 // T1
2  Read(B) → 500 // T1
3  Write(A, 100) // T1
4  Read(A) → 100 // T2
5  Write(B, 600) // T1
6  Read(B) → 600 // T2
7  Write(A, 0) // T2
8  Write(B, 700) // T2
   // A:0, B:700
```

(b) Interleaving 2

Figure 2-3: Transfer Transaction Interleavings.

3. To justify the motivation of database delegation, **the client is assumed to be computationally lightweight with limited memory.** If the client machine is powerful enough to handle the maximum possible data processing demand, setting up a local DBMS becomes a better choice in terms of security.

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. For example, Figure 2-3 shows two serializable interleavings. Both interleavings result in the same state (Account A being 0 and Account B being 700), but technically they differ by two steps (Step 4 and Step 5), highlighted in gray. To prove serializability, it suffices to show that the resulting state can be produced from running $T1$ followed by $T2$. In this case, we consider the minor difference in Step 4 and Step 5 as a runtime detail that can be decoupled from the verification.

Following this idea, 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, *deterministic reservation* (c.f. Section 8.1), which identifies a subset of non-conflicting transactions. Witnesses of correctly executing non-conflicting trans-

actions 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 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 efficiently verify succinct proofs.

2.3 Notations

In this section, we list the denotations we use in this thesis.

- $\mathbb{F}$ is a finite field. $GF(q)$ is the Galois Field of order q. $\mathbb{G}$ is a group. Specifically, $\mathbb{G}_?$ is a group with an unknown order. $\mathbb{N}$ is the set of natural numbers. $\mathbb{P}$ is the set of all the *odd* prime numbers. $\mathbb{P}_i$ is i-th prime category, a subset of $\mathbb{P}$.

- $\mathbb{Z}^+$ is the set of positive integers. $\mathbb{Z}_N^*$ is the multiplicative group of integers modulo N. $\mathbb{Z}_N^*$ contains all the integers in $[0, N-1]$ co-prime with N.

- $\mathcal{X}$ is the alphabet set, denoting the universe of all possible elements for an authenticated data structure.

- $[n]$ denotes the integer set from 1 to n: $[n] = [1, n] \cap \mathbb{N}$. We use $i, j, k, l, \ldots$ to represent integers.

- Unless stated explicitly, we use capitalized letters, e.g., S, to represent sets or mappings (a.k.a., dictionaries). Non-capitalized letters, e.g., x, represent values or functions. Fully capitalized words are gate types, e.g., ADD and MUL.

- We use yes and no to represent Boolean values yes and no.

- We use $deg(f)$ to represent the degree of a polynomial f.

- $\mathsf{poly}(\lambda)$ and $\mathsf{negl}(\lambda)$ represent values polynomial to the security parameter λ and negligible to λ, respectively. Namely, $\mathsf{poly}(\lambda) = \Theta(\lambda^k)$ for some integer k, and

$\mathsf{negl}(\lambda) = O(2^{-\lambda})$. Sometimes, we implicitly omit the security parameter λ and only use poly and negl.

- Bold letters, e.g., $\boldsymbol{x}$, mean vectors, and $\boldsymbol{x}^{(i)}$ stands for the i-th dimension value of $\boldsymbol{x}$. Unless explicitly stated otherwise, the indices start from 1. We use $\boldsymbol{f}(\cdot)$ to represent vector functions, namely, functions that return vectors. We use bold capitalized letters, e.g., $\mathbf{S}$, to represent a set of vectors. An exception is Chapter 13, where multi-linear extension points are technically vectors over the finite field, but we use italic symbols, e.g., x, to represent a multi-linear extension point. We also use italic symbols, e.g., f, to represent a function that maps to a multi-linear extension point. This is to distinguish a single point from a vector of multi-linear extension points, which we represent by bold letters, e.g., $\boldsymbol{x}$.

- Operators $\wedge, \vee$, and $\neg$ represent Boolean logic and, or, and negation operations, respectively. We use $\barwedge$ to represent Boolean NAND operation.

- We use $=$ to represent equations, $\leftarrow$ to represent assignments, and $:=$ to represent definitions. We use $A \stackrel{?}{=} B$ to represent the expression evaluated to be true if $A = B$ and false otherwise. We use $x \leftarrow_{\$} S$ to represent that x is uniformly randomly chosen from the set S.

- We use $\mathsf{MSG\_TYPE}$ to represent a message type in distributed communications. Examples include $\mathsf{MSG\_WRTXN}$ and $\mathsf{MSG\_PKEY}$.

- We use $\boldsymbol{u} \| \boldsymbol{v}$ to represent concatenating two vectors or strings a and b. If s is a string, we use $s[i \dots j]$ to represent the substring from i-th position to j-th position. If i is omitted, the substring starts from the first character (included). If j is omitted, the substring ends at the last character (included).

- We use $\tilde{f}$ to represent the *multi-linear extension* (c.f. Section 13.3.3) of the function f.

- The function $\phi(\cdot)$ is the Euler totient function, namely $\phi(N)$ is the number of positive integers in $[N]$ and co-prime with N.

2.4 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 track the changes on the data verifiably. We defer the formal definition of weakly-binding authenticated dictionaries to Section 5.1. Finally, we specifically design the concurrency control algorithm to process transactions in batches [148], which enables the aggregation of cryptographic computations and witnesses. We discuss how these building blocks work together in Chapter 4.

Cryptographic Circuits. The cryptographic circuit is a model to describe computation at a low level. People widely use circuits in cryptography, including in secure Multi-Party Computation (MPC), Fully Homomorphic Encryption (FHE), Zero-Knowledge Proofs (ZKP), and Verifiable Computation (VC).

Verifiable computation schemes require the input computation to be described as arithmetic cryptographic circuits over a finite field $\mathbb{F}$. Specially, Boolean circuits can be seen as special cases over the Galois field $GF(2)$. There are many definitions of cryptographic circuits, and some of them are equivalent to each other. Here, we present a simple but general one. Formally, we define a gate to be a tuple $(d_i, d_o, f : \mathbb{F}^{d_i+d_o} \to \{\mathsf{yes}, \mathsf{no}\})$, where d_i is the in-degree (i.e., the number of input wires), d_o is the out-degree (i.e., the number of output wires), and f is a function representing the semantic constraint of the gate. For example, an AND gate on Boolean values will have an in-degree 2, an out-degree 1, and a function $f(i_1, i_2, o_1)$ that outputs yes if and only if $o_1 = i_1 \wedge i_2$, where i_1 and i_2 are values of the input wires, and o_1 is the value of the output wire. Figure 2-4 shows an example set of gates for arithmetic circuits.

Gate	d_i	d_o	Constraints (f)
ADD	2	1	$f(i_1, i_2; o_1) := (o_1 \stackrel{?}{=} i_1 + i_2)$
MUL	2	1	$f(i_1, i_2; o_1) := (o_1 \stackrel{?}{=} i_1 \cdot i_2)$

Figure 2-4: **Example Set of Gates** — This example set provides addition gates and multiplication gates for arithmetic circuits.

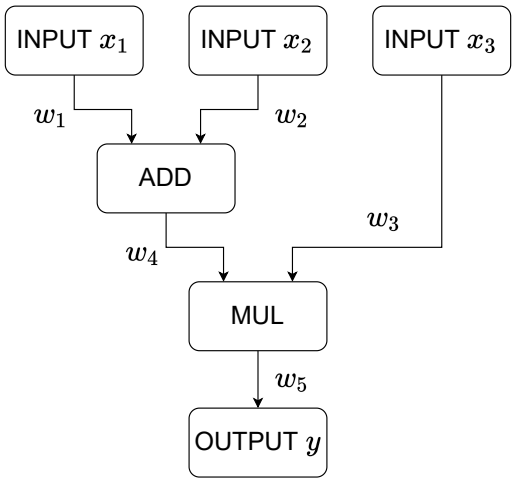


Figure 2-5: **Example Arithmetic Circuit** — This circuit expresses the computation $y = (x_1 + x_2) \cdot x_3$. It contains six gates, namely, 3 input gates, an addition gate, a multiplication gate, and an output gate.

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 $f = \mathsf{yes}$. 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 $C(\boldsymbol{x}) = \boldsymbol{y}$ to indicate that when the INPUT gates emit values in $\boldsymbol{x}$, the OUTPUT gates will report the values $\boldsymbol{y}$ as the output. In this thesis, we assume that the circuits are polynomially-sized and deterministic, unless explicitly stated otherwise.

Figure 2-5 shows an example arithmetic circuit. The circuit expresses the com-

putation $y = (x_1 + x_2) \cdot x_3$. The gate set follows the example in Figure 2-4, namely, $\mathcal{G} = \{\text{ADD, MUL, INPUT, OUTPUT}\}$. There are three input gates, emitting x_1, x_2, and x_3, respectively. An addition gate takes in the outward wires of x_1 and x_2 as inputs and emits them into a multiplication gate. The multiplication gate also takes in the outward wires of x_3, and emits into the output gate of y. To evaluate the circuit, we assign values $w_1, w_2, \ldots, w_5$ to the wires. Excluding the input gates and the output gates, there are two gates with the following constraints:

$$f_{ADD} = \mathsf{yes} \iff w_4 = w_1 + w_2 \tag{2.1}$$

$$f_{MUL} = \mathsf{yes} \iff w_5 = w_3 \cdot w_4 \tag{2.2}$$

In other words, $C(x_1, x_2, x_3) = y$ holds if and only if $w_1 = x_1, w_2 = x_2, w_3 = x_3, w_4 = w_1 + w_2, w_5 = w_3 \cdot w_4, y = w_5$ all hold, which implies $y = (x_1 + x_2) \cdot x_3$.

Verifiable Computation (VC): Verifiable computation is a cryptographic protocol that enables a (usually computationally limited) client to delegate expensive computation to an untrusted server, while being able to verify the computation results. It consists of three algorithms ($\mathsf{VC.Gen}, \mathsf{VC.Prove}, \mathsf{VC.Verify}$). Figure 2-6 shows a typical walkthrough of an instance of verifiable computation. The protocol involves two parties, the prover Prv and the verifier Vrf. The computation description F is known to both the prover and the verifier. Before the protocol starts, a trusted third party produces a pair of keys by calling $\mathsf{VC.Gen}$, the proving key prk and the verifying key vrk. At the beginning of the protocol, Vrf sends the input values x to Prv. Then, Prv uses prk and x to compute the result $y \leftarrow F(x)$ and a cryptographic proof π. Finally, Prv sends y together with π to Vrf, and Vrf verifies y and π with vrk. Vrf accepts the result y if the verification succeeds. In practice, the prover $\mathsf{VC.Prove}$ also takes in auxiliary information (e.g., the values of intermediate wires $\{w_i\}$) to help accelerate the proving process.

A verifiable computation is *correct* if and only if, when the claimed output $\boldsymbol{y}$ equals $C(\boldsymbol{x})$, the proof verification always passes. It is *sound* if and only if, when the

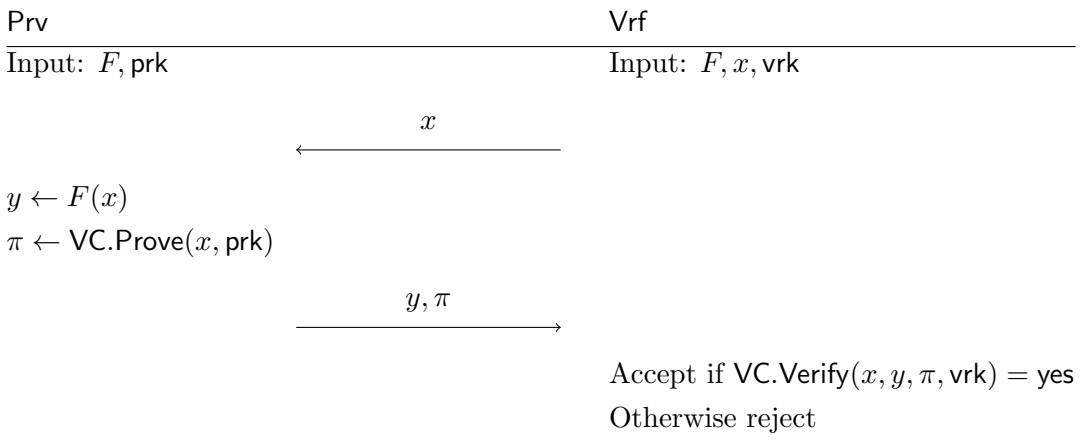


Figure 2-6: Verifiable Computation Walkthrough.

verification passes, there is only an exponentially small chance that the server could cheat by not computing correctly.

Definition 1 (Verifiable Computation Correctness)**.** *A verifiable computation scheme is* correct *if and only if, for any* $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{F}^{\mathsf{poly}}$*, and deterministic polynomially-sized circuit* C*, such that* $C(\boldsymbol{x}) = \boldsymbol{y}$*, the following holds:*

$$\Pr\left[\mathsf{VC.Verify}(\boldsymbol{x}, \boldsymbol{y}, \pi, \mathsf{vrk}) = \mathsf{yes} : \begin{matrix}(\mathsf{prk}, \mathsf{vrk}) \leftarrow \mathsf{VC.Gen}(1^\lambda, C), \\ \pi \leftarrow \mathsf{VC.Prove}(x, \mathsf{prk})\end{matrix}\right] = 1.$$

Definition 2 (Verifiable Computation Soundness)**.** *A verifiable computation scheme is* sound *if and only if, for any probabilistic polynomial time (p.p.t.) adversary* A *and any deterministic polynomially-sized circuit* C*, the following holds:*

$$\Pr\left[\begin{matrix}C(\boldsymbol{x}) \neq \boldsymbol{y}, \\ \mathsf{VC.Verify}(\boldsymbol{x}, \boldsymbol{y}, \pi, \mathsf{vrk}) = \mathsf{yes}\end{matrix} : \begin{matrix}(\mathsf{prk}, \mathsf{vrk}) \leftarrow \mathsf{VC.Gen}(1^\lambda, C), \\ (\pi, \boldsymbol{x}, \boldsymbol{y}) \leftarrow A(1^\lambda, C, \mathsf{prk})\end{matrix}\right] = \mathsf{negl}(\lambda).$$

We call a verifiable computation scheme a *Universal Verifiable Computation* scheme if the proving key and verifying key do not depend on the circuit. Later, in Chapter 11, we will discuss how using universal VCs will change the workflow. There are also cryptographic primitives (e.g., STARKs [16]) that imply a *transparent* verifiable computation scheme. In other words, a trusted setup becomes unnecessary.

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 cryptographic circuits using compilers like Frigate [110], extracts interleaving hints from the concurrency control algorithms, and applies a verifiable computation scheme on the circuit and the interleaving hints. The verifiable computation scheme guarantees that if the function returns true and the proofs pass verification, the client can rest assured that the transactions were executed correctly, and the semantic properties are preserved.

Informally, we present a small example to show how a verifiable computation scheme proves the simple circuit in Figure 2-5 is evaluated correctly. To begin with, we convert the circuit into a Rank-1 Constraint System (R1CS). Given a circuit C, a R1CS is a group of quadratic equation constraints on the input values $\boldsymbol{x}$, output values $\boldsymbol{y}$, and the intermediate values $\boldsymbol{w}$. It has feasible solutions if and only if $C(\boldsymbol{x}) = \boldsymbol{y}$ holds.

For each gate G, we write out its semantic constraint $f_G(\boldsymbol{i}; \boldsymbol{o}) = \mathsf{yes}$, where $\boldsymbol{i}$ is the vector of input wire values, and $\boldsymbol{o}$ is the vector of output wire values. If this equation is already quadratic, namely, it only contains terms with degrees no more than two, we add this equation to the R1CS group and proceed to the next gate. Otherwise, we introduce intermediate variables and split the constraint into multiple quadratic constraints. For example, $x^3 = y$ can be rewritten into two constraints $x^2 = w$ and $w \cdot x = y$ by introducing a variable w. Concretely, for the circuit example in Figure 2-5, we convert the circuit into the following R1CS.

$$\begin{cases} x_1 - w_1 & = 0, \\ x_2 - w_2 & = 0, \\ x_3 - w_3 & = 0, \\ w_1 + w_2 - w_4 & = 0, \\ w_3 \cdot w_4 - y & = 0 \end{cases}$$

Substituting w_1, w_2, and w_3 with x_1, x_2, and x_3 results in the following simplified R1CS.

$$\begin{cases} x_1 + x_2 - w_4 & = 0, \\ x_3 \cdot w_4 - y & = 0 \end{cases}$$

This R1CS contains five variables, namely, x_1, x_2, x_3, w_4, and y. It is straightforward to see it implies $y = (x_1 + x_2) \cdot x_3$. For convenience, we concatenate the variables and the constant 1 into a single vector $\boldsymbol{v} = [x_1, x_2, x_3, w_4, y, 1]^T$ so that every constraint can be written in the form of $(\boldsymbol{a} \cdot \boldsymbol{v})(\boldsymbol{b} \cdot \boldsymbol{v}) - \boldsymbol{c} \cdot \boldsymbol{v} = 0$, where $\boldsymbol{a}$, $\boldsymbol{b}$, and $\boldsymbol{c}$ are coefficient vectors.

Many verifiable computation schemes take in R1CS formats as input directly. Take Groth16 [76] as an example. Given evaluation points provided by the verifier $\delta_1, \delta_2, \ldots, \delta_n$, the scheme encodes the coefficients of the terms into polynomials and converts the constraint system into the following form (also known as QAP),

$$A(\delta) \cdot B(\delta) - C(\delta) = 0, \tag{2.3}$$

such that it holds for all $\delta = \delta_i$, where $A(\delta) = \boldsymbol{\alpha}(\delta)^T \cdot \boldsymbol{v}$, $B(\delta) = \boldsymbol{\beta}(\delta)^T \cdot \boldsymbol{v}$, and $C(\delta) = \boldsymbol{\gamma}(\delta)^T \cdot \boldsymbol{v}$. The vector polynomial $\boldsymbol{\alpha}(\delta_i)$ is the coefficient of the corresponding term $\boldsymbol{a}$ in the i-th constraint for all $i \in [n]$. The definitions of $\boldsymbol{\beta}(\delta)$ and $\boldsymbol{\gamma}(\delta)$ are similar.

Equation 2.3 holds for all $\delta = \delta_i$ if and only if there exists a polynomial H such that

$$A(\delta) \cdot B(\delta) - C(\delta) = H(\delta) \cdot Z(\delta),$$

where the target polynomial $Z(\delta) = \prod_{i \in [n]} (\delta - \delta_i)$, by the fundamental theorem of algebra.

As for the example of Figure 2-5, we have two constraints: $(x_1 + x_2) \cdot 1 - w_4 = 0$ and $x_3 \cdot w_4 - y = 0$. We let $\boldsymbol{\alpha}(\delta_1) = [1, 1, 0, 0, 0, 0]^T$, $\boldsymbol{\beta}(\delta_1) = [0, 0, 0, 0, 0, 1]^T$, and $\boldsymbol{\gamma}(\delta_1) = [0, 0, 0, 1, 0, 0]^T$ to encode $(x_1 + x_2) \cdot 1 - w_4 = 0$. And similarly, $\boldsymbol{\alpha}(\delta_2) = [0, 0, 1, 0, 0, 0]^T$, $\boldsymbol{\beta}(\delta_2) = [0, 0, 0, 1, 0, 0]^T$, and $\boldsymbol{\gamma}(\delta_2) = [0, 0, 0, 0, 1, 0]^T$ encode $x_3 \cdot w_4 - y = 0$.

Given the input values $x_1 = 1, x_2 = 3, x_3 = 3$, the output value $y = 12$. The

prover first derives $w_4 = 4$. Assume $\delta_1 = 1$ and $\delta_2 = 2$, we have $A(1) = 4, B(1) = 1, C(1) = 4, A(2) = 3, B(2) = 4, C(2) = 12$. Interpolating these polynomials yields $A(\delta) = 5 - \delta, B(\delta) = 3\delta - 2$, and $C(\delta) = 8\delta - 4$. Then, we have

$$A(\delta) \cdot B(\delta) - C(\delta) \tag{2.4}$$

$$=(5 - \delta) \cdot (3\delta - 2) - (8\delta - 4) \tag{2.5}$$

$$= - 3\delta^2 + 9\delta - 6 \tag{2.6}$$

$$= - 3(\delta - 1)(\delta - 2). \tag{2.7}$$

Namely,

$$H(\delta) := \frac{A(\delta) \cdot B(\delta) - C(\delta)}{Z(\delta)} = \frac{-3(\delta - 1)(\delta - 2)}{(\delta - 1)(\delta - 2)} = -3.$$

It suffices to convince the verifier that H exists. A naive way is to send H directly to the verifier and let the verifier perform polynomial operations to check it. However, the number of constraints easily exceeds tens of millions in the real world. The polynomial H could be too large. The proof is not succinct anymore.

Fortunately, a number of tools can help the prover at this stage. For example, Groth16 [76] uses bilinear mapping and polynomial commitments to convince the verifier of correct polynomial evaluation. STARK [14] relies on collision resistance of hash functions and proximity proofs. Plonk [64] utilizes a permutation argument. We defer discussions of verifiable computation schemes to Section 2.5.

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 by the client. A weakly-binding authenticated dictionary guarantees the correctness and soundness properties (c.f. Section 5.2) if the digest committer and updater are trusted to behave honestly. A closely related terminology is *strongly-binding authenticated dictionaries*, where the correctness and soundness

properties hold even when a malicious adversary performs the digest generating and updating. Again, we defer discussions on existing authenticated dictionaries to Section 2.5. This dissertation proposes two constructions of authenticated dictionaries. One is weakly-binding, and the other is strongly-binding. Further, we propose a new cryptographic primitive, *wildcard accumulator*, as a generalization of authenticated dictionaries in Section 5.3.

In Litmus, we let the client as well as the delegated computation maintain an authenticated dictionary to track the database state. The client and the server agree on the initial digest. And, the delegated computation, which performs digest updates, is guaranteed correct by the verifiable computation framework. Therefore, a weakly-binding authenticated dictionary is sufficient. In Chapter 5, we discuss authenticated dictionaries in detail.

Deterministic Reservation: This is a concurrency control algorithm that processes transactions by batches [148, 27], which we call *processing batches* to distinguish it from the *verification batch* (the number of transactions submitted by the client). It identifies a maximal non-conflicting subset of transactions, as described in Chapter 8.

Here, we briefly explain the algorithm. It proceeds in two phases:

- In the first phase, the transactions run in parallel. Whenever any transaction modifies a row in the table, instead of applying the changes directly, it leaves a reservation with its predefined priority. Reservations with higher priorities overwrite the ones with lower priorities.

- In the second phase, the DBMS checks the transactions' read and write sets in parallel. If any row accessed by a transaction T is reserved by another transaction with a higher priority, the DBMS moves T to the next batch and re-tries it later. Lastly, it returns those transactions that survived as the non-conflicting subset and applies their changes to the database.

In our design, deterministic reservation helps the authenticated dictionary scheme

"merge" the non-conflicting transactions and provide aggregated proofs of data integrity. Merging transactions reduces the workload of the verifiable computation framework as well. Further, given the transaction set and the processing batch size, deterministic reservation produces a fixed transaction scheduling. This determinism greatly simplifies the client's work of validating circuits, as it can produce the same scheduling on its own. Then, circuit validation becomes simple pattern matching.

2.5 Related Work

Cryptographically-Protected Cloud DBMSs. Many existing works on secure cloud databases focus on protecting the data content [63, 174, 120]. They either use searchable encryption schemes or order-preserving/order-revealing encryption schemes to hide the plaintext of data and let the server operate on the ciphertext. Recently, researchers proposed oblivious databases to obfuscate the access patterns further to hide metadata [59, 96]. When the server cannot see the data, it could be argued that it has no motivation to tamper with the data or falsify the computation. However, we argue that this implication does not always hold. In the real world, external incentives jump in. For example, a malicious hacker can manipulate the data of a publicly-traded company to benefit from shorting the stock prices, regardless of whatever the data content is. We believe that guaranteeing data integrity and computation integrity is critical, and it is orthogonal to data privacy protection. An ideal secure database system needs both integrity and privacy.

General-purpose verifiable computation. 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.

Existing works of verifiable computation mainly fall into four categories. The first category includes attempts based on *linear probabilistically checkable proofs* (Linear PCPs). Ishai et al. [81] proposed a technique to compile an exponentially long PCP

into an efficient argument system. Combined with the Hadamard-code-based PCP construction due to Arora et al. [7], this leads to an argument system for NP with a constant number of communications between the prover and the verifier. Gennaro et al. presented a method known as the GGPR protocol to encode programs into quadratic span programs [65]. The GGPR protocol constructs a non-interactive zero-knowledge argument in the CRS model for Circuit-SAT problems using a constant number of group elements. Pinocchio [118], proposed by Parno et al., further improves the GGPR protocol and builds an end-to-end system that converts (a subset of) C code into formats suitable for the protocol. Bitansky et al. proposed a general method for the construction of preprocessing succinct non-interactive arguments (SNARGS) with concrete efficiency improvements [26]. This type of methods is further developed and implemented in many follow-up works [130, 129, 128, 35, 29, 158, 160, 18, 21, 50].

The second category is based on Interactive Proof (`IP`) systems. Originated from the works of Lund et al. [103] and Shamir [131] that `IP=PSPACE`, Goldwasser et al. [72] proposed the concept of delegating computation based on interactive proofs, together with several constructions, known as the GKR protocol. Cormode et al. derived a new protocol [53] known as the CMT protocol that makes GKR more practical. Allspice [154] refined CMT by improving the circuit components and generalizes the CMT protocol to support a broader class of computation with a failover mechanism. Zebra [155] is an implementation specifically for Application-Specific Integrated Circuit (ASIC) verification based on the Allspice interactive protocol. Giraffe [156] is a similar system based on a new interactive protocol by Thaler et al. [142], which improves the CMT protocol for circuits with special wiring patterns.

The third category uses *short PCPs*. With collision-resistant hashing functions (CRHFs) and PCPs, Kilian et al. showed an interactive argument [91] consisting of only four messages for NP, where the verification time is only poly-logarithmic in the computation itself. This line of PCP schemes was further improved through [23, 24, 17, 20]. Ben-Sasson et al. proposed SCI, a scalable PCP construction [13] based on core PCP techniques like proof composition and PCP of Proximity. The SCI system provides verifiable computation integrity, whose definition is similar to verifiable

computation.

The fourth category utilizes garbled circuits [173] and MPC-in-the-head techniques [82]. Jawurek, Kerschbaum, and Orlandi proposed an actively secure zero-knowledge protocol [84] based on Yao's garbled circuits. ZKBoo [69] further developed this idea and made the non-interactive proof for Boolean circuits more practical.

There are also other lines of work related to verifiable computation constructions. For example, Bulletproofs [38] is a practical argument system proposed by Bünz et al. It is based on the zero-knowledge construction for arithmetic circuit satisfiability by Bootle et al. [34], optimizing the Groth protocol [75].

Existing verifiable computation frameworks are not designed for parallel environments or practical scenarios. Besides, most of them are not general enough to directly support external memories, while the primary job of a DBMS system is to read and write external memories. Therefore, it is challenging for us to build capable verifiable computation frameworks suitable for a modern database system with massive parallelism.

By using general-purpose verifiable computation schemes [26, 130, 129, 128, 35, 29, 158, 160, 18, 21, 50], we can, in theory, construct a verifiable database system that satisfies the cryptographic properties of Section 3.1 by compiling the whole DBMS into a giant circuit. Even though recent constructions show promising asymptotic results [123], this would still incur an impractical computational overhead due to significant constant factors. Moreover, existing verifiable computation frameworks are not designed for parallel environments and cannot practically support large database-style workloads. Litmus only verifies essential parts of the database and parallelizes the provers to achieve a practical throughput.

Authenticated Data Structures: Cryptographic accumulators [61], multiset hashes [51], vector commitment [44], and authenticated dictionaries [150, 151] are widely used in verifiable data storage in various settings. For example, Jain et al. [83] uses Merkle trees to keep track of the data on the server. vChain [171, 161] 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 [170] 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: Directly applying verifiable computation techniques to the database context is impractical because it requires the server to compile the entire database logic into a program expressed by Boolean/arithmetic circuits. To circumvent the need to compile the whole DBMS server into a circuit, vSQL [181] and IntegriDB [182] construct VC schemes that handle the processing of a non-trivial subset of SQL, one query at a time, while Litmus focuses on concurrent transactions with low-level accesses, namely, *read* and *write* operations.

Figure 2-7 gives an overview of vSQL's approach. To begin, the client sends its data to the database server. Then, both the server and the client compute a digest based on the data. For each SQL query, the server and client run an interactive verifiable-computation protocol [53] that takes more than $2,000$ seconds per query, convincing the client that the server executed the query correctly. At the end of this protocol, the client holds an updated database digest, and the parties can repeat this process for subsequent queries. The system executes each query in strict sequential order, which guarantees atomicity and isolation. However, vSQL has *no parallelism*: if a client issues 50 queries at once (as is common in many database applications), the client and server must execute the proving protocol in serial order. So, while this and other recent works make important progress towards practical verifiable databases, there is still much work to be done in bringing the full power of multi-core processors and concurrency to the realm of database verification.

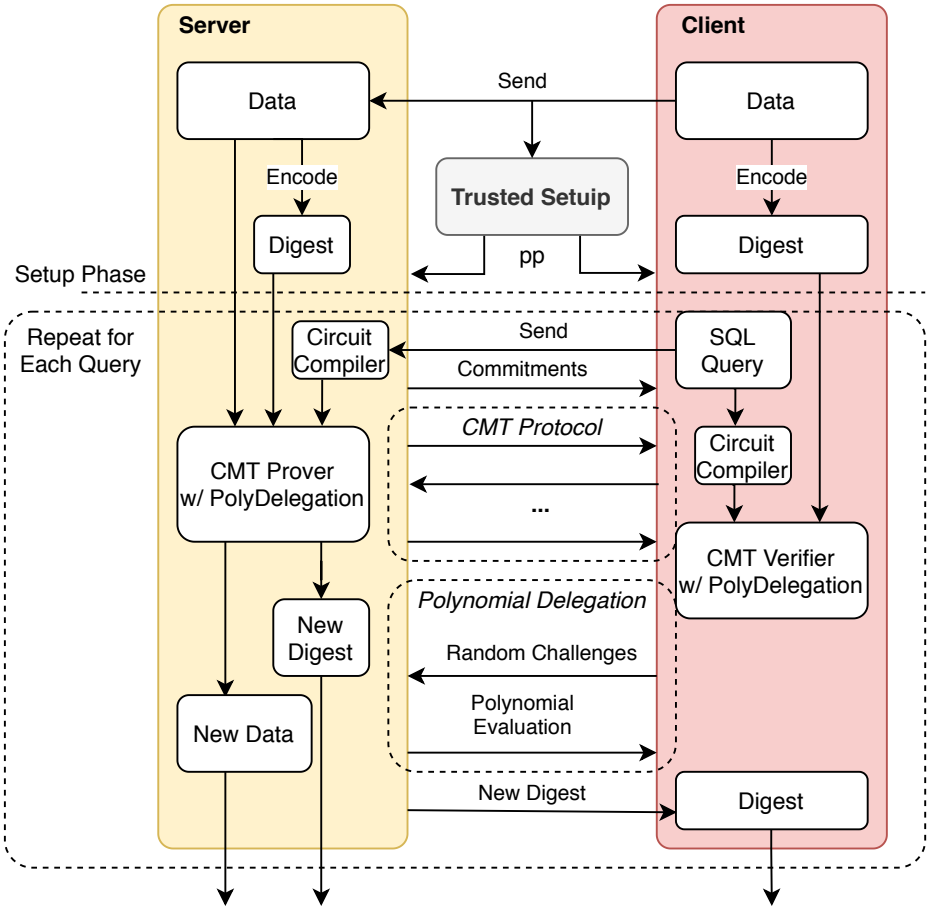


Figure 2-7: **Related Existing Work: vSQL** — The vSQL system [181] achieves a verifiable DBMS by sequentially invoking the CMT [53] protocol per query.

Verification of concurrent systems: Recent works consider the task of verifying general-purpose computations in a concurrent setting. Orochi [139] is a system for verifying PHP web applications. Spice [127] 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. However, we may be able to leverage the techniques developed in these works in the future.

Checking serializability: Recent works have proposed using external programs to verify the serializability of transactions in a DBMS [136, 172, 93, 1, 141, 140, 141, 37, 78, 112, 176]. These programs take the traces from a transactional database and verify

whether the execution is serializable. These techniques either require the DBMS to 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 [141]. Works like Elle [92] require the database to make data accesses into list operations to keep track of the history of the tuples; we evaluate Elle in Section 10.3. Other approaches that provide verifiable serializability include Haeberlen et al. [77] and Jain et al. [83]. The work by Haeberlen et al. [77] applies to general distributed systems settings, where the nodes are uniform, and at least one node is honest [77]. We believe this technique will be helpful in verifiable database replications. Compared to the classic Merkle-tree approach, Jain et al. [83] is novel in decoupling the data owner and the clients and introducing postponed verification; this enables the server to process transactions in parallel. However, the DBMS has to send the entire history of Merkle tree roots to the data owner for later verification, which is, in fact, an equivalent of a detailed trace of sequential ordering. Alternatively, the client might choose to immediately verify the transaction before issuing the next one to avoid such overhead, but then the DBMS cannot exploit parallelism. Moreover, the single MBTree root is affected by updates on any tuple, so the performance is limited by the global contention of the MBTree. In contrast, Litmus can reach roughly three orders of magnitude higher throughput on a database with 10,000x more tuples. Besides, Litmus incurs no initialization overhead if the database is in a default state (a state that both the server and the client previously agreed on), while a Merkle-tree approach has to incur at least a linear overhead. However, interactive approaches like Jain et al. [83] have lower latency.

Chapter 3

Cryptographic Formalizations

3.1 Cryptographic Formalization

We now present a sketch of the cryptographic framework to formalize ACID properties. We focus on verifying serializability, the highest level of isolation, and atomicity. However, 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 Chapter 11.

Formally, we model the database state as a bitstring $D \in \{0,1\}^*$ (as some encoding of a dictionary). We model a transaction $T\colon \{0,1\}^* \to \{0,1\}^* \times \{0,1\}^*$ 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:

- $\mathsf{Digest}(D) \to \delta$. Compute a succinct cryptographic digest δ of the database D. Ideally, δ is constant-sized assuming the security parameter is fixed.

- $\mathsf{Execute}(D, \mathcal{T}) \to (D', \mathcal{V}, \pi)$. 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.

- $\mathsf{Verify}(\mathcal{T}, \delta, \delta', \mathcal{V}, \pi, \mathsf{aux}) \to \{0,1\}$. 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 $\mathcal{V}$, a claimed proof of correctness π, and auxiliary information aux, check the proof and output "1" if and only if the proof is valid. For generality, we include the auxiliary information in the definition in case the verification function needs extra information. For example, Litmus with a non-deterministic concurrency control algorithm in Section 6 needs the server to send the circuit C to the client. In this case, $\mathsf{aux} = \{C\}$. In contrast, Litmus with a deterministic concurrency control algorithm does not need auxiliary information to verify the properties.

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

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

Namely, the correctness property says that the verification always passes if the digests δ and δ' correspond to the beginning state and the ending state while the database management system is executing transactions honestly.

In the real world, database management systems allow transactions to abort themselves. For example, a transfer transaction aborts if the sender account does not have enough balance. For theoretical simplicity, we view self-aborted transactions as committed transactions. These transactions return special values to distinguish them from normally committed transactions.

The soundness, however, is a bit more complicated as it depends on the isolation

level. For now, we present the soundness definition with serializability. In Section 3.2, we will talk about other isolation levels.

Informally, a verifiable database scheme is sound for serializability if, for all lists of transactions $\mathcal{T} = \{T_1, \ldots, T_n\}$ and initial database state D_0, whenever an adversary produces an ending digests δ', a list of outputs $\mathcal{V} = \langle v_1, \ldots, v_n \rangle$, a proof π, and some auxiliary information that the verifier accepts, this adversary "must know" the corresponding database D_n and a permutation σ on $\{1, \ldots, n\}$ 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 $i = 1, \ \ldots, \ n$: $\quad (D_i, \ v_i) \leftarrow T_{\sigma(i)}(D_{i-1})$, and

(c) $\delta' = \mathsf{Digest}(D_n)$.

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

Definition 4 (Soundness – Serializability)**.** *A verifiable database scheme* (Digest, $\mathsf{Execute}, \mathsf{Verify}$) *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 $\mathcal{T} = \{T_1, \ldots, T_n\}$ and an initial database D_0 that*

$$\Pr[\mathsf{Verify}(\mathcal{T}, \delta_0, \mathcal{A}(D_0, \mathcal{T})) = \mathsf{yes}]$$

is non-negligible (where $\delta_0 = \mathsf{Digest}(D_0)$), the extractor $\mathcal{E}^{(\mathcal{A})}$ outputs databases $\{D_i\}_{i \in \{1,\ldots,n\}}$ and a permutation σ on $\{1, \ldots, n\}$ such that the following quantity is negligibly *close to 1 in the (implicit) security parameter:*

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

Note that this definition also implies atomicity because no transactions are partially executed.

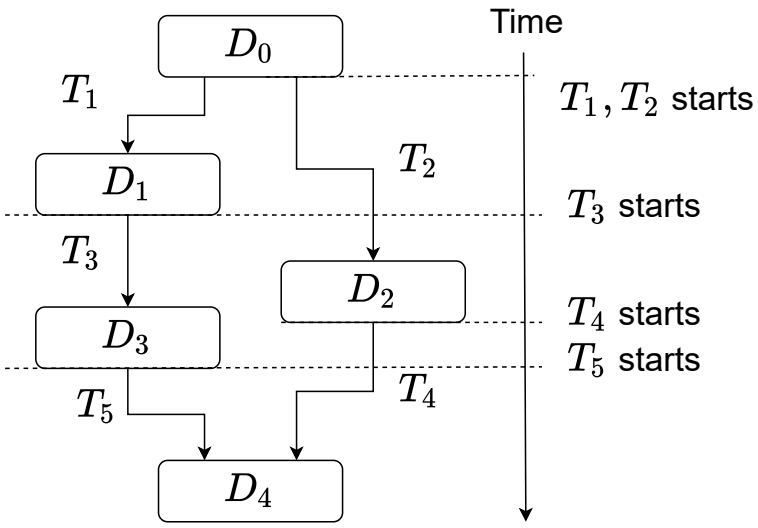


Figure 3-1: Example Interleaving of Repeatable Read Isolation.

3.2 Alternative Isolation

Besides serializability, there are other isolation levels, such as Read Uncommitted, Read Committed, and Repeatable Read [135].

In this section, we take *repeatable read* as an example to show how to extend the definition in Section 3.1 to other isolation levels.

The Repeatable Read isolation level provides two guarantees:

- Within a single transaction, read operations of the same data location always return the same result, unless the transaction itself has updated it.
- Every read value is written by committed transactions, or from the initial values when no transaction has updated it. In other words, no transactions can see intermediate values from uncommitted transactions.

Intuitively, when every transaction starts, it "freezes" a snapshot and always reads and writes to this snapshot. When the transaction commits, it applies the changes to the database.

Figure 3-1 shows an example interleaving that follows the repeatable read isolation level. Initially, the database state is D_0. A transaction $T1$ starts and changes the state from D_0 to D_1. At the same time, another transaction $T2$ starts and changes

the state from D_0 to D_2. Then, a transaction $T3$ continues on the path of $T1$, and it cannot see the changes of $T2$ because $T2$ is still running. Before $T3$ finishes and results in state D_3, a transaction $T4$ starts from state D_2. Shortly after $T3$ applies the changes to the database, $T5$ starts. After both $T4$ and $T5$ commit (in arbitrary order), the database's final state becomes D_4. Note that this transaction interleaving is not necessarily *serializable*.

	Initially, X=0, Y=0, Z=0
$T1$	If X is not 20, X=10
$T2$	If X=0, write X=20 and Y=10
$T3$	If X=10, write Y=20
$T4$	If Y=10, write Y=100
$T5$	If Y=20. write Z=10

Figure 3-2: Example of Transactions that make the Interleaving in Figure 3-1 Non-Serializable.

State	X	Y	Z
D_0	0	0	0
D_1	10	0	0
D_2	20	10	0
D_3	20	20	0
D_4	20	100	10

Figure 3-3: Data States of the Example in Figure 3-2.

To see why this interleaving does not imply serializability, consider the transactions in Figure 3-2. Running them results in data states shown in Figure 3-3. Now we show that there is no serial order of the transactions such that running them one by one transforms D_0 to D_4.

Assume there is a serial order of $T1$, $T2$, $T3$, $T4$, and $T5$ that leads to the state D_4. Since the resulting state contains $Z = 10$, we know this must come from $T5$. For $T5$ to write $Z = 10$, $T3$ must have written $Y = 20$. For $T3$ to write $Y = 20$,

$T1$ must have written $X = 10$. Similarly, $Y = 100$ comes from $T4$. The effect of $T4$ implies $T2$ has written $Y = 10$. Now, we have two chains of partial order constraints, $T1 \to T3 \to T5$, and $T2 \to T4$. Consider $T1$ and $T2$. If $T1$ runs before $T2$ in the serial order, the if-condition in $T1$ will hold since only $T2$ writes $X = 20$. Then, the if-condition in $T2$ will never hold because the value of X is changed, and no one else changes X back to 0. Therefore, $T2$ runs before $T1$. However, this leads to $T1$ not writing $X = 10$, which stops $T3$ from writing $Y = 20$, and therefore, $T5$ will not write $Z = 10$. This conflicts with the final state. Such serial order does not exist, and the interleaving is not serializable.

Now, we discuss the cryptographic formalization of a verifiable repeatable read database. The correctness definition remains the same. Different from the *serializability soundness*, the *repeatable read soundness* relaxes the sequential order of transactions into a Directed Acyclic Graph (DAG). In other words, the permutation σ in Definition 4 now becomes a graph of states.

Before we present the formal definition, there is a caveat we need to address. In Figure 3-1, both $T4$ and $T5$ lead to the same state. If the write operations of $T4$ and $T5$ do not conflict with each other, D_4 simply means the state of applying both $T4$ and $T5$. However, if the write operations do conflict, there is no guarantee that the conflicting parts will reflect which transaction's effects.

Consider the resulting state D_4, the values can only come from three places — (i) D_3, the most recent state by the physical time order, (ii) changes by $T4$ executing from D_2, and (iii) changes by $T5$ executing from D_3.

Denote the write set of a transaction T when reading from a data state D as $\mathsf{WS}(T, D) := \{(k_i, v_i)\}$, where (k_i, v_i) means T wrote to address k_i with value v_i. To quantify this observation, we define the *repeatable read* constraint $\mathsf{RR}(T_1, D_1, T_2, D_2, \ldots, T_n, D_n; D_{n+1}) = \mathsf{yes}$ if and only if there exists an $i_0 \in [n]$, representing the index of the most recent state, such that

- For any k s.t. $D_{n+1}(k) \neq D_{i_0}(k)$, there exists $i \neq i_0$ s.t. $(k, D_{n+1}(k)) \in \mathsf{WS}(T_i, D_i)$. Namely, every change in D_{n+1} comes from one of the transactions.

- For all k s.t. there exists v s.t. $(k, v) \in \{\cup_{i\in[n]}\mathsf{WS}(T_i, D_i)\}$, $(k, D_{n+1}(k)) \in \{\cup_{i\in[n]}\mathsf{WS}(T_i, D_i)\}$. Namely, for all the places the transactions intend to modify, at least one transaction's write operation goes through.

To simplify the denotation, we define $\mathsf{InNode}(D)$ as the list of incoming edge transactions and their source nodes. For example, $\mathsf{InNode}(D_4)$ in Figure 3-1 returns $(T4, D_2, T5, D_3)$. Now, we can present the soundness definition for repeatable read with an extractor that produces a DAG of states and transactions.

Definition 5 (Soundness – Repeatable Read)**.** *A verifiable database scheme* (Digest, $\mathsf{Execute}, \mathsf{Verify}$) *is* sound for repeatable read *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 $\mathcal{T} = \{T_1, \ldots, T_n\}$ and initial database D_0 that*

$$\Pr[\mathsf{Verify}(\mathcal{T}, \delta_0, \mathcal{A}(D_0, \mathcal{T})) = \mathsf{yes}]$$

is non-negligible, the extractor $\mathcal{E}^{(\mathcal{A})}$ outputs databases states $\mathcal{D} := (D_0, D_1, \ldots, D_m)$ and a DAG $\mathcal{G}$, where the vertices are the states and the edges are the transactions, such that the following quantity is negligibly *close to 1 in the (implicit) security parameter:*

$$\Pr\left[\begin{array}{r} \mathsf{Digest}(D_0) = \delta, \\ \text{for all vertex } D \neq D_0\text{:} \\ \mathsf{RR}(\mathsf{InNode}(D); D) = \mathsf{yes}. \\ \text{for all edge } T_i \in \mathcal{T} : D_j \to D_k, \\ (D_k, v_i) = T(D_j). \\ \mathsf{Digest}(D_m) = \delta' \end{array} : \begin{array}{r} (\delta, \delta', \mathcal{V}, \pi) \leftarrow \mathcal{A}(D_0, \mathcal{T}), \\ (\mathcal{G} := (\mathcal{D}, \mathcal{T}), \sigma) \leftarrow \mathcal{E}^{(\mathcal{A})}(D_0, \mathcal{T}) \end{array}\right].$$

We remark that the repeatable read soundness also implies atomicity, similar to the serializability soundness.

3.3 Consistency

Consistency is a special property that is application-dependent and relies on the programmer to be implemented correctly. Namely, it specifies an invariant that a transaction should convert a *consistent* database state to another *consistent* state, where the programmer defines whether or not a state is *consistent*. For example, in a bank account system, transfer transactions should preserve the sum of account balances — the total sum of money of all the accounts remains the same since no money is created or lost during the transfer.

Define the *consistency check* as a function CC that takes in the before-state D_0 and after-state D_1 and returns whether or not the check is successful. For example, in the bank transfer system, we can define CC as:

$$\mathsf{CC}(D_0, D_1) = \Big(\sum_{(k,v)\in D_0} v \stackrel{?}{=} \sum_{(k,v)\in D_1}\Big).$$

Now, we present the soundness definition for consistency assuming the isolation level is serializability.

Definition 6 (Soundness – Consistency)**.** *A verifiable database scheme* ($\mathsf{Digest}, \mathsf{Execute}, \mathsf{Verify}$) *is* sound for consistency *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 $\mathcal{T} = \{T_1, \ldots, T_n\}$ and initial database D_0 that*

$$\Pr[\mathsf{Verify}(\mathcal{T}, \delta, \mathcal{A}(D_0, \mathcal{T})) = 1]$$

is non-negligible, the extractor $\mathcal{E}^{(\mathcal{A})}$ outputs databases (D_0, D_n) and a permutation σ on $\{1, \ldots, n\}$ such that the following quantity is negligibly *close to 1 in the (implicit)*

security parameter:

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

The definition is almost the same as serializability soundness, except that we perform a consistency check on the states before and after every transaction.

Chapter 4

System Overview

This chapter presents an overview of the verifiable database system. We assume that a single client interacts with a single database server for simplicity. 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 for them. However, the single-client setting does not prevent our design from being performant. Chapter 8 introduces concurrency, where the client submits a batch of transactions.

Supporting multiple clients is non-trivial. It is challenging to distinguish between a malicious server tampering with the data or just another client modifying the data. The design must incorporate Public Key Infrastructure (PKI) into the digest updates and verification. We will discuss multi-client support in Section 11.4.

In this chapter, we first present the overview by each component in Section 4.1, and then, we show the system in the time order in Section 4.2.

4.1 Overview By Components

This section focuses on verifying *Atomicity* and *Isolation* (namely, *Serializability*) of the database. We will discuss *Consistency* and *Durability* in Section 11.2 and Section 12.2, respectively.

For a DBMS, one of the most important properties is data integrity — correct computation comes from correct data. We use the **memory integrity** component

to keep data integrity. It uses succinct authenticated data structures [33, 151] to help the client to track the data state. The component consists of two parts, the integrity provider and the integrity checker. The provider runs natively on the server, and it provides proofs for read values. The checker is plugged into the circuits and checks the proofs. A malicious server cannot tamper with the data without the client noticing it unless with extremely small probability. We defer detailed discussion of memory integrity to Section 5.2.

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 transactions** to help the verifiable computation scheme handle the interleavings. A wrapped transaction represents a set of transactions "glued" together, with the logic of the memory integrity checker (c.f. Section 5.2) plugged into each transaction. The function takes the initial digest and all the read values together with their memory integrity proofs as input. Specifically, before every transaction starts to run its logic, it first runs the checker to see whether the read values and their proofs are consistent with its local digest. If the check passes, the transaction logic continues. Otherwise, it aborts by directly returning 0. After the transaction is done, it applies the written values to the digest. The circuit of the wrapped transaction is the cryptographic description of the exact same logic, compiled into a network of Boolean/arithmetic operation gates.

Figure 4-1 shows an example of a wrapped transaction containing two transactions, $T1$ and $T2$. $T1$ reads DB[A] and assigns the value to variable X. Then, it computes the value of X + 1 and assigns it to Y. Finally, it writes the value of Y to DB[A] back. Similarly, $T2$ reads DB[A] and DB[B] and assigns the values to X and Y, respectively. It computes X + Y and assigns it to a new variable Z. Lastly, $T2$ writes Z to DB[A]. The wrapped transaction takes the initial digest d_0, the read values, and their corresponding memory integrity proofs. The function body runs $T1$ and $T2$ in the sequential order. Before running $T1$, it performs an integrity check on the read values DB[A] for X with the digest d_0. If the check fails, it will abort and return zero. Then, it runs the logic of $T1$ and records all the write operations, namely,

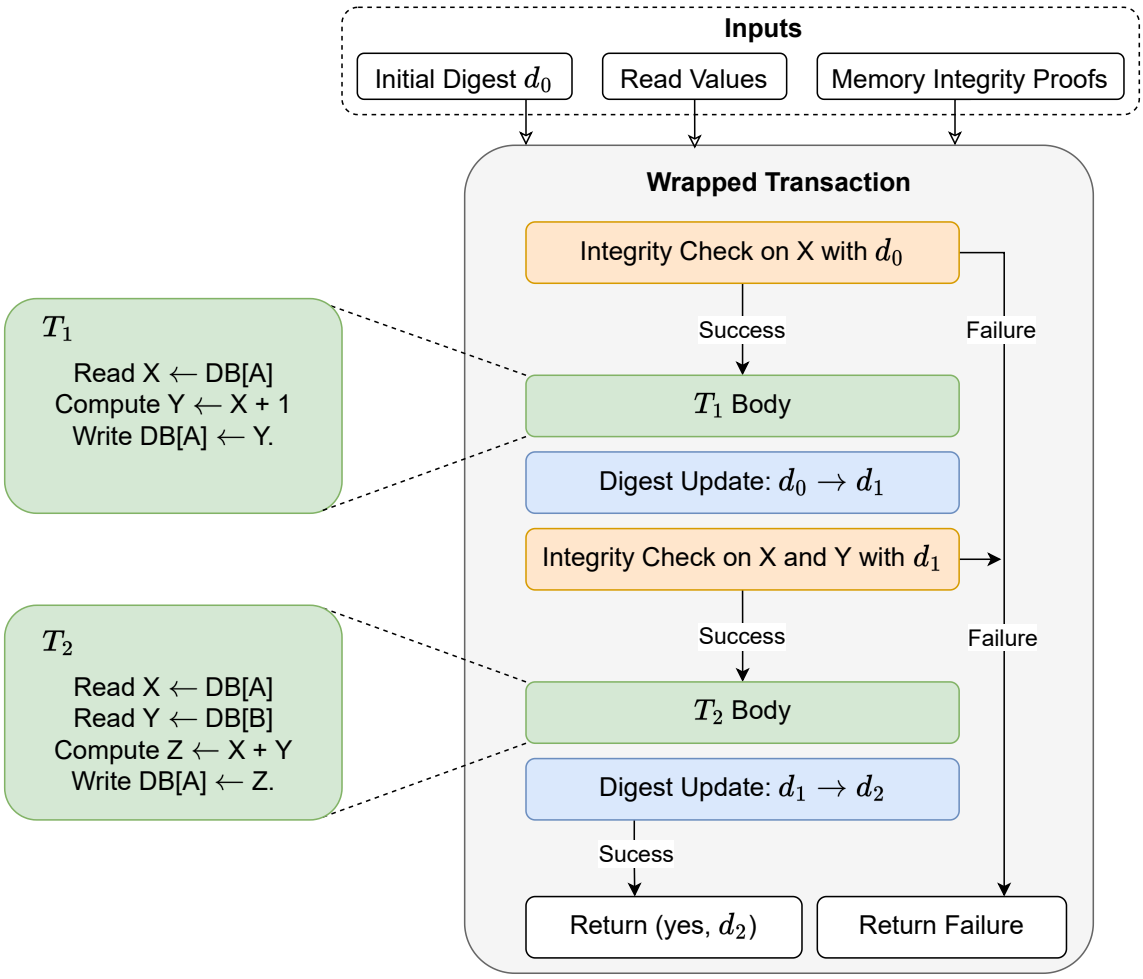


Figure 4-1: **Example Wrapped Transaction** — This wrapped transaction chains two transactions $T1$ and $T2$ in the sequential order.

DB[A] ← Y. After the transaction finishes, it performs the digest update through the authenticated dictionary interface and produces a new digest d_1. The dictionary corresponding to d_1 has a key-value pair DB[A] ← Y. Next, the wrapped transaction performs the integrity check on the read values of $T2$ (namely, X ← DB[A] and Y ← DB[B]) with the new digest d_1. Note that $T2$ should read the new value of DB[A] written by $T1$. The logic of $T2$ follows the check. Finally, the wrapped transaction performs another digest update and returns a tuple (yes, d_2), where yes indicates that all the integrity checks have passed, and d_2 is the latest digest.

Figure 4-2 shows a detailed view of $T1$ in Figure 4-1. The input contains the previous check status, the old digest d_0, the address A, the value DB[A], and its proof

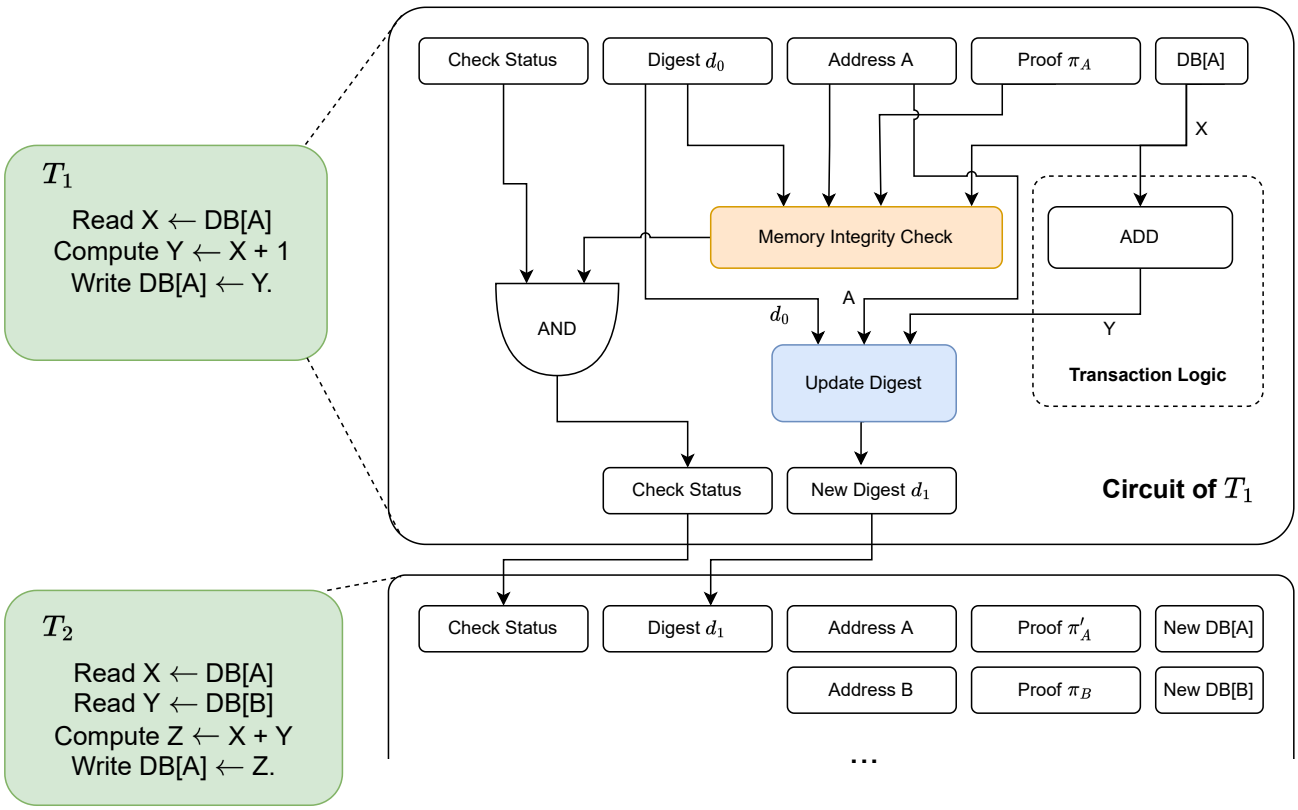


Figure 4-2: **Example Transaction Circuit** — Zoom-in View of $T1$ in Figure 4-1.

π_A. Excluding the gates for checking the read values and updating the digest, the computation logic of $T1$ contains only a single ADD gate. The orange box represents the circuit representation of the memory integrity checker. It takes in d_0, A, π_A, and the value X =DB[A]. It outputs a result bit indicating whether the check succeeds. The result bit goes into the AND gate with the previous check status. The new check status is true if and only if both the result bit and the previous check status are true. Similarly, the blue box represents the updating circuit. Given the old digest d_0, the address A, and the value Y, it produces a new digest d_1. Both the check status and the new digest go into the inputs of the next transaction circuit, namely, the circuit of $T2$.

We defer additional details of wrapped transaction to Chapter 6.

As shown in Figure 4-3, the system contains three key modules: (1) The **server** hosts the **normal DBMS**, receives transactions from the clients, and creates a

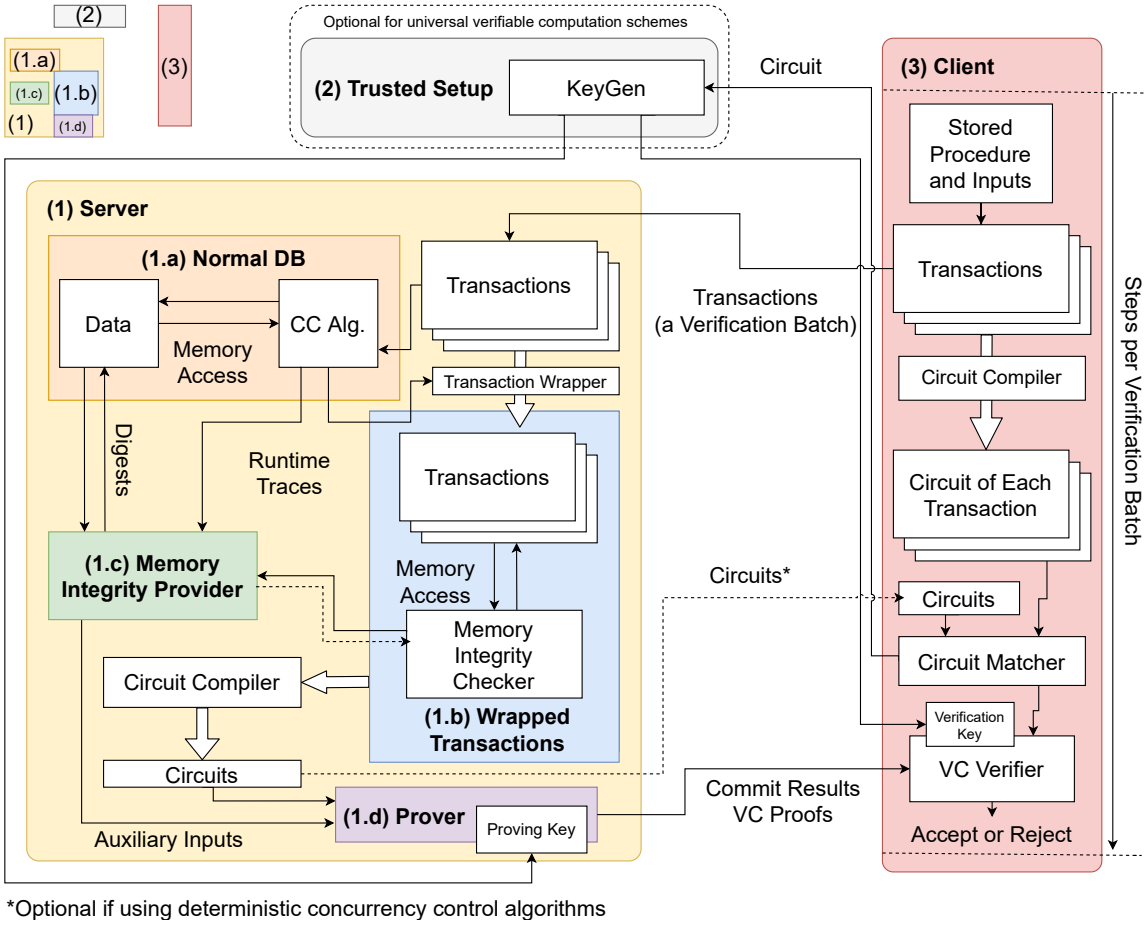


Figure 4-3: **Overview of the Proposed Verifiable Database** — The system contains three modules as shown in the top left corner: (1) the verifiable DBMS with (1.a) a normal DBMS system, (1.b) the wrapped transactions, (1.c) the memory integrity provider, and (1.d) the verifiable computation prover; (2) the (optional) trusted third-party setup; and (3) the client.

wrapped transaction; The server also contains the **memory integrity provider** and the **verifiable computation prover**. (2) The optional **key generator** can be the client itself or a trusted third-party, which can be implemented by another verifiable computation instance if the client wants the server to carry the heavy computation (as the key generation logic is fixed, there is no recursive circuit dependency on the key generation). Alternatively, we can use a *universal verifiable computation scheme* (where the keys do not depend on the circuits, making key generation a one-time cost) or a *transparent verifiable computation scheme*, which does not require trusted setups; (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. If this is the first time of running the protocol, we also assume that the client and the server agree on an initial database state, and the client needs to compute the initial digest. 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 transaction. Upon receiving the circuit of the wrapped transaction (c.f. (1b)) from the server, the client performs the circuit well-formedness check. It tries to match the local circuits and the circuit of the wrapped transaction sent from the server. If the check passes, the client sends the circuit to the key generator. The client also contains the verifier part of the verifiable computation framework, which takes in the verification key from the key generator and the proof generated from the server. The verifier outputs a single bit indicating whether the proof is accepted or rejected. Later, we explain the steps in the chronological order in Section 4.2.

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 and verification keys are sent to the server and the client, respectively. Here, we assume that a trusted third-party does this process. However, in practice, depending on the application, this process can also be done by (i) the server together with the client (through proof systems or secure two-party computation), or (ii) the client itself. We cannot solely rely on the server to perform the key generation because a malicious server is able to sample the key pair from particular distributions to break the security.

We now describe the four components of the server.

(1a) Normal DBMS: This is a full-fledged database system. For the sake of generality of our design, we place minimal constraints on the normal database component.

Theoretically, it can run any valid concurrency control algorithm like Two-Phase Locking (2PL) or Optimistic Concurrency Control (OCC). However, we choose to use the *deterministic reservations* concurrency control algorithm in Litmus (see Section 8.1) 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 concurrency control algorithm is deterministic, the client by itself is able to infer the transaction interleaving and produce the whole circuit. As a result, 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. The runtime traces closely resemble logging records (e.g., ARIES [109]). It exposes the runtime traces to the *memory integrity provider* to prepare memory digests and integrity proofs, and to the *transaction wrapper* to coordinate the interleaving of transactions in the circuit. Related to the two types of database logging schemes, namely, 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 chain the transactions one by one into a single wrapped transaction, as shown in Figure 4-1. It takes in a group of transactions as well as the specification of the memory integrity checker, and it outputs a circuit representing the wrapped transaction. Plugging in the checker logic 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 a logic circuit 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. Figure 4-2 showed an example of transaction circuit. The key generator then uses the circuit to generate the proving

key and the verification key, and the prover uses the circuit 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 of 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 verifiable computation prover takes in the proving key, the circuit generated by the transaction wrapper, and the inputs supplied by the memory integrity provider. Then, it outputs a proof indicating the transaction's output is correct for 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 verifiable computation constructions. In Litmus, we use Groth16 [76] to instantiate the verifiable computation scheme and parallelize the prover for better performance. However, our design is compatible with verifiable computation schemes in general. For example, we can plug Hydra, our new verifiable computation scheme, into Litmus. We will explain Hydra and how to use it in Litmus in Chapter 13 later.

4.2 Chronological Flow

Figure 4-4 shows an overview of the steps in the time order. As the readers may recall, the system consists of three participants — the server, the client, and the key generator. In the beginning, the client sends the transactions to the server. Upon receiving the transactions, the server runs them with the *normal database* and compiles them into circuits representing *wrapped transactions*. Then, the server sends the circuits to the client for a well-formedness check. Note that this step (in the orange shade) is optional if the system uses deterministic concurrency control algorithms

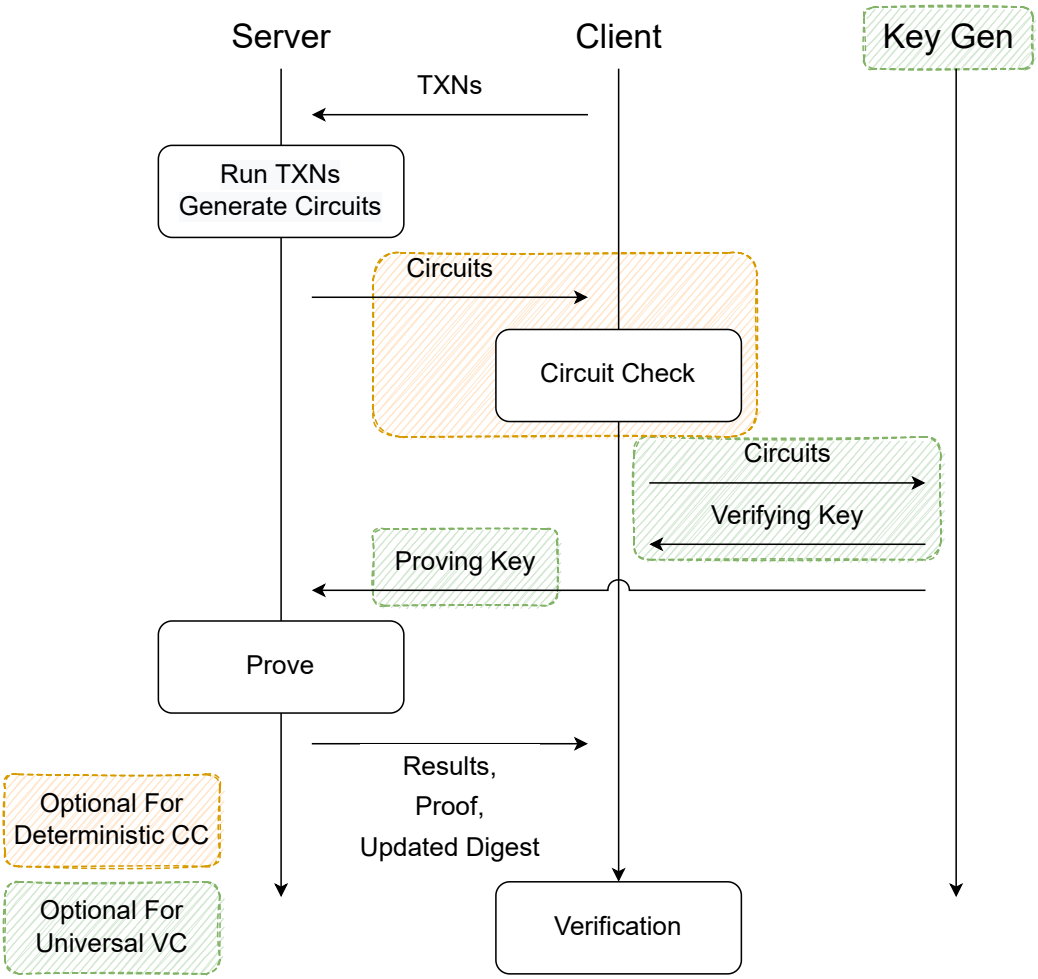


Figure 4-4: **Overview of Chronological Steps** — There are three entities, the server, the client, and the key generator. Steps in the green shade and the key generator are optional if using universal verifiable computation schemes. Steps in the orange shade are optional if using deterministic concurrency control algorithms.

because the client is able to locally produce exactly the same circuits as those on the server. Next, the client forwards the circuits to the key generator, which returns a verification key to the client and a proving key to the server. The server starts proving the circuit with the verifiable computation prover upon receiving the proving key. When the prover finishes, the server sends the result, proof, and the updated digest to the client. Finally, the client performs the verification. The client can rest assured that the transactions are correctly executed if the verification passes. Note that if the system uses a universal verifiable computation scheme, the key pair does not depend on the circuit. Therefore, the steps in the green shade become optional.

Chapter 5

New Authenticated Data Structures and Memory Integrity

In Chapter 4, we walked through the system by component. Among the components, the memory integrity provider and checker are critical for proving and verifying transaction properties. This chapter presents new authenticated data structures. They are building blocks for verifying memory integrity. We first describe a weakly-binding authenticated dictionary construction in Section 5.1. Then, we show how to build the memory integrity provider and checker with the authenticated dictionary in Section 5.2. Lastly, we generalize the authenticated dictionary to a new cryptographic primitive, wildcard accumulators in Section 5.3. When we talk about dictionaries, we mean single-valued dictionaries. Namely, the dictionary contains at most one value for each key.

5.1 Weakly-Binding Authenticated Dictionaries

Before diving into the details of the system, we discuss a useful cryptographic primitive to guarantee data integrity in Litmus. We model the data storage as a large key-value store.

We propose a new weakly-binding authenticated dictionary scheme that only needs a constant length of storage, a constant length for a lookup proof, and a constant

number of arithmetic operations for each verification. The authenticated dictionary scheme is based on an RSA accumulator [31]. One might argue that it is trivial to build a weakly binding authenticated dictionary 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. Some recent RSA-based authenticated dictionaries [151] support key-nonexistence proofs, but they also, at the cost of extra computational complexity, come with more properties (e.g., incremental proof aggregation) we do not need in Litmus. 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 authenticated dictionary can track only the "accessed" key-value pairs. When the transaction requests a value not accessed before, the server can prove that the requested key was not previously accessed, and provide an initial value, say 0, previously agreed with the client. Moreover, the naive approach will be problematic if we extend the system to multiple clients. If the digest always covers the entire memory space, then *any client* updating *any memory slot* has to notify *all the clients* holding a digest of the memory. Otherwise, the local digests of the clients will become invalid and inconsistent with the current state. This authenticated dictionary scheme is the core building block of the memory integrity model. In total, it results in a linear number of extra logic gates in the circuit in terms of the number of batches of non-conflicting transactions.

Compared to our authenticated dictionary construction, using Merkle Tree in Litmus has the following disadvantages: (1) the proofs are $O(\log n)$-sized; (2) checking the proofs incurs $O(\log n)$ computation in the circuit; (3) Merkle Trees incur an initialization overhead depending on the data size. When the default data is uniform, the initialization takes $O(\log n)$ computation. However, when the default data is procedurally generated, e.g., $DB[x] = x^2$, the Merkle Tree initialization cost degenerates to $O(n)$ time.

5.1.1 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 verify such proofs with a constant number of operations. These properties make our authenticated dictionary scheme circuit-friendly.

Categorization of Prime Numbers:

A categorization of prime numbers is a set of *disjoint* subsets $\mathsf{cat} = (\mathbb{P}_1, \mathbb{P}_2, \ldots, \mathbb{P}_l)$, 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 $(\mathsf{Sample}, \mathsf{Verify})$ that satisfy the following:

- $\mathsf{Sample}(\lambda, i, nonce)$ 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.
- $\mathsf{Verify}(p, i)$ 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 time-bounded by poly, and Sample outputs a unique prime number for a nonce with an overwhelming probability.

A categorization scheme satisfies the following properties.

Definition 7 (Correctness). *For any bit-length λ, a categorization index $i \in [l]$, and any nonce, we have*

$$\Pr[\mathsf{Verify}(p, i) = \mathsf{yes} : p \leftarrow \mathsf{Sample}(\lambda, i, nonce)] \geq 1 - \mathsf{negl}.$$

Definition 8 (**Soundness**). *For any adversary A, any bit-length λ and a categorization index $i \in [l]$, we have*

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

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 $13 \in \mathbb{P}_3$.

Therefore, we can simply "plug" a rejection sampling after an existing collision-resistant hash function $\mathsf{phash}(\cdot)$ from the domain space to prime numbers. Figure 5-1 shows an example of a hash function that maps to a prime number with the target category. This construction is collision-resistant if the underlying function phash is collision-resistant. However, this construction needs extra assumptions on phash, because k could exhaust the space before it finds a prime number in that category. We avoid diving into details on this problem since it happens only with negligible probability if the mapping is *uniform* enough, qualitatively speaking.

On input *nonce* and target category index i, initialize $k = 0$, and repeat the following steps:

- Let $p = \mathsf{phash}(k \cdot |\mathcal{X}| + nonce)$.
- If $p \equiv i \pmod l$, return p. Otherwise, increase k and continue the loop.

Figure 5-1: Example instantiation of a hash function to a prime number with the target category.

5.1.2 Assumptions and Interfaces

The authenticated dictionary scheme relies on the *strong RSA assumption* [10].

An RSA group is defined as the multiplicative group of integers modulo N, denoted as $\mathbb{Z}_N^*$, where N is the product of two large prime numbers $N = pq$ and therefore hard to factorize.

Definition 9 (Strong RSA Assumption)**.** *An RSA group generator* RSAGen *satisfies the strong RSA assumption if and only if for all p.p.t. adversary* $\mathcal{A}$*,*

$$\Pr\left[\begin{array}{ccc} u^x \equiv a \pmod N, & (p,q) \leftarrow \mathsf{RSAGen}(\lambda), N = pq, \\ x \geq 3, & : & a \leftarrow_{\$} \mathbb{Z}_N^*, \\ x \in \mathbb{P}. & & (u,x) \leftarrow \mathcal{A}(a,N) \end{array}\right] \leq \mathsf{negl}.$$

Namely, given a random element a in an RSA group, it is hard to find a base u and a non-trivial exponent x that u^x leads to a.

Before we dive deeper, we provide the definition of authenticated dictionaries.

Notation: We use $k \in D$ to indicate key k is in the dictionary with some value $v \neq \perp$. We also use $(k, v) \in D$ to indicate key k has value $v \neq \perp$ in the dictionary. We assume a well-formed dictionary has at most one value for a key. We often denote a subset of a dictionary as a pair (K, V) where $V(k)$ stores the value of each key $k \in K$.

Definitions: An AD scheme consists of the following APIs.

- $\mathsf{Setup}(1^\lambda) \to (\mathsf{prk}, \mathsf{vrk})$. Returns the *proving* and *verification keys*.
- $\mathsf{Commit}(\mathsf{prk}, D) \to d$. Returns a digest d of the dictionary D.
- $\mathsf{Update}(\mathsf{prk}, d, K, V, V') \to d'$. Updates the digest d by setting the value of the key $k \in K$ from $V(k)$ to $V'(k)$. It returns the new digest d'.

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

- $\mathsf{VerLookup}(\mathsf{vrk}, d, K, V, \pi) \to b \in \{\mathsf{yes}, \mathsf{no}\}$. Verifies the proof that each $k \in K$ has value $V(k)$ in the dictionary with digest d.

- $\mathsf{ProveNoKey}(\mathsf{prk}, d, D, K) \to \phi$. Returns a *non-membership proof* ϕ that there does not exist any key value pair (k, v) with $k \in K$ in the dictionary with digest d.

- $\mathsf{VerNoKey}(\mathsf{vrk}, d, K, \phi)$. Verifies the proof ϕ 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 10 (**Correctness**). *An authenticated dictionary scheme is correct if,* $\forall$ *public parameters* $(\mathsf{prk}, \mathsf{vrk}) \leftarrow \mathsf{Setup}(1^\lambda)$, $\forall$ *dictionaries* D *with digest* $d \leftarrow \mathsf{Commit}(\mathsf{prk}, D)$, *the following hold:*

- Lookup correctness: $\forall$ *sets of keys* K, *if* $\pi \leftarrow \mathsf{ProveLookup}(\mathsf{prk},\ D,\ K,\ V)$ *and* $V(k) = D(k),\ \forall k \in K$, *then* $\mathsf{VerLookup}(\mathsf{vrk},\ d,\ K,\ V,\ \pi) = 1$.

- Key Non-membership correctness: $\forall\ K$, *if* $\pi \leftarrow \mathsf{ProveNoKey}(\mathsf{prk},\ d,\ D,\ K)$ *and* $\forall k \in K,\ k$ *is not in* D, *then* $\mathsf{VerNoKey}(\mathsf{vrk},\ d,\ K,\ \pi) = 1$.

Definition 11 (**Weak Key Binding**). $\forall$ *p.p.t. adversaries* $\mathcal{A}$, *the following inequalities hold:*

Lookup Soundness:

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

Key Non-membership Soundness:

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

5.1.3 Extending RSA Accumulators to Authenticated Dictionaries

A naive way to extend RSA accumulators to an authenticated dictionary is to map the key-value pairs to distinct prime numbers and use RSA accumulators to accumulate them. This naturally results in an authenticated dictionary that supports membership/non-membership proofs of key-value pairs. However, it is hard to add provable key-nonexistence support because the structural information of keys gets lost when mapping a key-value pair to a prime number — you cannot tell whether or not there is a key-value pair with a specific key by testing divisibility since even two pairs with the same key are mapped to irrelevant prime numbers. Therefore, we map a key-value pair to a product of multiple prime numbers in different categories.

Now, we are ready to extend an existing dynamic universal RSA accumulator scheme (e.g. [33]) 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, and the dot symbol $(\cdot)$ means multiplication.

- $\mathsf{Setup}(1^\lambda) \to (\mathsf{prk}, \mathsf{vrk})$. Sample an RSA group $\mathbb{G}_?$ with generator g. Set $\mathsf{prk} = g$ and $\mathsf{vrk} = g$. The order of $\mathbb{G}_?$ remains secret.

- $\mathsf{Commit}(\mathsf{prk}, D) \to d$. 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

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

- $\mathsf{ProveLookup}(\mathsf{prk}, D, K) \rightarrow \pi$. 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

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

- $\mathsf{VerLookup}(\mathsf{vrk}, d, K, V, \pi) \rightarrow \{\mathsf{yes}, \mathsf{no}\}$. Checks whether

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

- $\mathsf{Update}(\mathsf{prk}, D, d, K, V) \rightarrow d'$. We first compute $\pi \leftarrow \mathsf{ProveLookup}(\mathsf{prk}, D, K)$, then build the new digest d' based on π:

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

- $\mathsf{ProveNoKey}(\mathsf{prk}, d, D, K) \rightarrow \{A, B\}$. We compute $(A, B) = \mathsf{Bézout}(S, \prod_{k\in K} \mathsf{Sample}(\lambda, 0, k))$, where $\mathsf{Bézout}(x, y)$ returns the Bezout coefficients A, B s.t. $Ax + By = 1$ for x, y with $gcd(x, y) = 1$.

- $\mathsf{VerNoKey}(\mathsf{vrk}, d, K, A, B) \rightarrow \{\mathsf{yes}, \mathsf{no}\}$. 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 Chapter 8 is *aggregatability*. As shown in the $\mathsf{ProveLookup}$ and $\mathsf{VerLookup}$ interfaces, we can aggregate several lookups into a set of keys and provide a single proof for all of them. The same holds for key non-existence proofs and digest updates. 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 [33]. Our authenticated dictionary scheme inherits this property.

Theorem 1. *The RSA-based authenticated dictionary scheme is correct and weakly key binding.*

We include the proof in Appendix A.2.

As an example, suppose we want to compute a digest d of dictionary D where $D[1] = 2,\ D[3] = 4$. Then, $d \leftarrow g^{H(1,2)\cdot H(3,4)}$. To prove $D[1] = 2$, the server needs to compute $\pi \leftarrow g^{H(3,4)}$ as the proof. To verify, the client can check whether $\pi^{H(1,2)}$ 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 [119, 115]). Figure 5-1 shows an example process of sampling by rejection. Since the process is deterministic, the prover does not have influence over the prime number choices. *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 (okay to be costly), and "boost" it up by providing (r, a) s.t. $r < p_0,\ \gcd(a^r - 1,\ rp_0 + 1) = 1,\ a^{rp_0} \equiv 1 \pmod{rp_0 + 1}$. Then, $N := rp_0 + 1$ 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 the certificates a_i's and r_i's 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.

This construction is not the first authenticated dictionary scheme to offer key-nonexistence proofs. My colleagues and I proposed a construction [151] based on the CF13 vector commitment [46]. It has a dedicated RSA accumulator for all the keys. Therefore, proving the non-membership of a key in this accumulator is sufficient to argue key-nonexistence in the authenticated dictionary. It also offers other properties (e.g., incremental proof aggregation) we do not use here. Merkle-Patricia Tries provide key-nonexistence proofs by having *terminating leaves* in a Merkle Trie [149]. Another trie-based construction [150] also has *frontier nodes* similar to terminating leaves. Proving the existence of a terminating leaf on the path of a key suffices to argue the key does not exist in the dictionary. Compared to the existing constructions, the construction presented in this section is the most lightweight in terms of asymptotic storage and proof sizes.

5.2 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, corresponding to the checker, as small as possible. Existing works do not perfectly suit our purpose because they either need a variable length digest (e.g., Merkle Tree) or require significant time to verify.

We follow the approach in Pantry [35] to enable a memory access interface for circuits, except that we use our new authenticated dictionary scheme. As shown in Algorithm 1 and Algorithm 3, the memory integrity model consists of two parts — the *provider* and the *checker*. The provider runs natively on the server and gener-

ates 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 a *consistent* initial digest, namely $g_0 \leftarrow \mathsf{Commit}(\mathsf{prk}, D_0)$. The initial digest does not have to cover all the possible memory addresses. For example, D_0 could be empty.

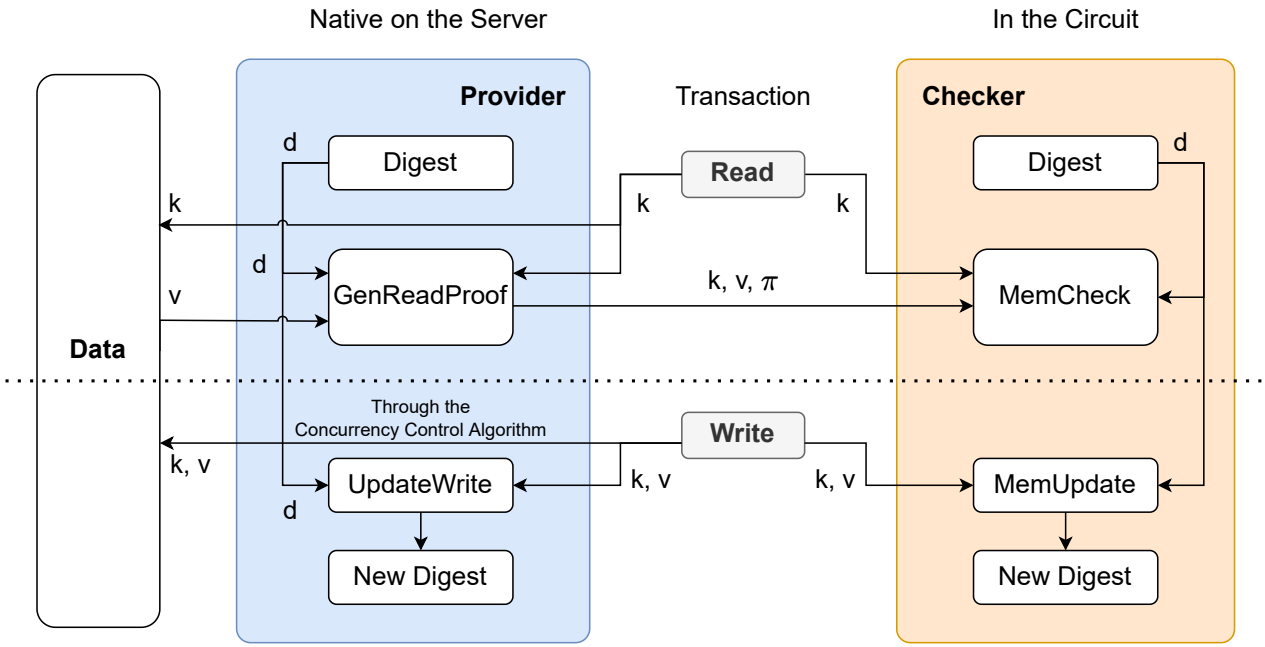


Figure 5-2: **Memory Integrity** — This figure shows the interaction between the provider and checker for read and write operations.

Figure 5-2 shows how the provider and the checker work together to protect data integrity. For every read operation on address k, the server reads the value v and calls $\mathsf{GenReadProof}$ with k, v. The function generates a proof π. The server stores the tuple (k, v, π) and inputs it to the circuit. Inside the wrapped transaction circuit, $\mathsf{MemCheck}$ takes the tuple and verifies the proof. Similarly, the server calls $\mathsf{UpdateWrite}$ with the new value pair (k, v) for a write operation and applies the written values to the data according to the concurrency control algorithm. $\mathsf{UpdateWrite}$ updates the digest. Similarly, the circuit also updates its local digest with $\mathsf{MemUpdate}$.

Memory Integrity Provider

The memory integrity provider maintains two global variables, S and acc. The former (S) tracks the product of the elements, and acc stores the latest authenticated dictionary digest. The provider also maintains a dictionary to keep track of the memory changes. In the initialization function InitProvider, 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 authenticated dictionary 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 (Algorithm 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 $\pi \leftarrow g^{S/H(k,v)}$. 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 a default value.

The UpdateWrite operation is simpler (Algorithm 1). It reads the old value at address k from the dictionary D and computes the lookup proof $\pi \leftarrow g^{S/H(k,v')}$. 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.

Algorithm 2 shows the memory integrity provider but with abstracted authenticated dictionary interfaces.

Memory Integrity Checker

The memory integrity checker consists of three interfaces, MemInit, MemCheck, and MemUpdate. Before a transaction starts, the circuit first invokes MemCheck (Algorithm 3) to determine whether the read values are correct according to its local digest, namely, $\pi^{H(k,v')} =$ acc. If this fails, it checks whether the proof indicates that the address has not been accessed previously so the value should be a default value. If both fail, the function returns 0, indicating that integrity is compromised.

After the transaction finishes execution, the circuit runs MemUpdate (Algorithm 3).

Algorithm 1: Memory Integrity Provider (on the Server Side) with Initial Database State Agreed as (g_0, D_0).

Input: $\mathsf{AgreedInitState} := \{g_0, D_0\}$

Func InitProvider ():

  $S \leftarrow \prod_{(k,v)\in D_0} H(k, v)$;

  $\mathsf{acc} \leftarrow g_0$;

  $D \leftarrow D_0$; /* initialization */

Func GenReadProof (k, v):

  if $k \leftarrow D$ **then**

    return $\pi \leftarrow g^{S/H(k,v)}$; /* generate the lookup proof */

  else

    /* non-existence proof */

    return $(A, B) \leftarrow \mathsf{Bézout}(S, \mathsf{Sample}(\lambda, 0, k))$

  end

Func UpdateWrite (k, v):

  $v' \leftarrow D[k]$; /* old value */

  $\pi \leftarrow \mathsf{GenReadProof}(k, v')$; /* must be a lookup proof */

  $\mathsf{acc} \leftarrow \pi^{H(k,v)}$; /* update the digest */

  $S \leftarrow S/H(k, v') \cdot H(k, v)$; /* update the product */

  $D[k] \leftarrow v$; /* update the dictionary */

  return

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 (namely, transactions write to an address only after reading it), since π is already verified in the read operation. 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 performance concern here is computing large exponentiation. We can address this by letting the server provide the result directly with a Proof-of-Exponentiation [33] protocol. 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[1], meaning the

[1]The checker performs exactly three exponentiations, two multiplications, three comparisons, and two Boolean operations per request. Using the Proof-of-Exponentiation protocol causes extra three exponentiations and three multiplications. This logic produces a constant number of gates.

Algorithm 2: Memory Integrity Provider (with Authenticated Dictionary Abstraction).

Input: $\mathsf{AgreedInitState} := \{d_0, D_0\}$ and the authenticated dictionary proving key prk.

Func InitProvider ():
  $d \leftarrow d_0$;
  $D \leftarrow D_0$; /* initialization */
Func GenReadProof (k, v):
  if $k \in D$ **then**
    /* generate the lookup proof */
    return $\pi \leftarrow \mathsf{ProveLookup}(\mathsf{prk}, d, D, \{v\}, \{k\})$
  else
    /* non-existence proof */
    return $\phi \leftarrow \mathsf{ProveNoKey}(\mathsf{prk}, d, D, \{k\})$
  end
Func UpdateWrite (k, v'):
  $d \leftarrow \mathsf{Update}(\mathsf{prk}, d, \{k\}, \{D[k]\}, \{v'\})$;
  $D[k] \leftarrow v$; /* update the dictionary */
  return

total number of gates of the circuit is linear with the number of transactions.

Algorithm 4 shows the memory integrity checker but with abstracted authenticated dictionary interfaces.

5.3 Wildcard Accumulators

In this section, we generalize the authenticated dictionaries to a new cryptographic primitive, *wildcard accumulators*. We show that our construction in Section 5.1 fits the more general definition. Wildcard accumulators can also be used in the memory integrity provider and checker to provide more functionalities. For example, the database can track multi-column tables and perform authenticated queries on rows with conditions on a subset of columns, instead of tracking key-value pairs only. Namely, the transactions can perform SQL-like queries on multiple columns, e.g., "find students who majors in Mathematics and lives in Massachusetts," or, `SELECT * FROM students WHERE major=math AND state=MA`.

Algorithm 3: 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 $\leftarrow g_0$; /* initialize the local digest */
Func MemCheck (k, v, π, A, B):
  if $\pi^{H(k,v)} =$ acc *or* $(\text{acc}^A \cdot g^{B \cdot H(k,v)} = g$ *and* $v = 0)$ **then**
    return 1 ; /* verification passes */
  end
  return 0
Func MemUpdate (k, v', v, π):
  if $\pi^{H(k,v')} \neq$ acc **then**
    return 0 ; /* verification failure */
  end
  acc $\leftarrow \pi^{H(k,v)}$; /* update the local digest */
  return 1

From	To
Guy Rothblum, Ron Rothblum, Shafi Goldwasser	(apple, orange), (banana, orange), (watermelon, pineapple)

Figure 5-3: Insecure Mapping of Cryptographers' Names.

5.3.1 Introduction

Conventional authenticated data structures often operate on uniform data entries like elements of a set, dimensions of a vector, or key-value pairs of a dictionary [33, 151, 108, 46]. For the sake of security, these schemes intentionally treat each data item as a whole and map it to an element in the working group. This prevents learning the pattern of the commitments. For example, if we work on a commitment scheme for cryptographers' names, we probably do not want the attacker to find out which one is the commitment of Guy Rothblum from the knowledge of that of Ron Rothblum. Figure 5-3 shows one such insecure scheme. It is easy to infer that the surname "Rothblum" is mapped to "orange".

However, for some applications, we do want to perform a partial information lookup over the commitment digest. Examples include blurry search on relational databases, access control in secure systems, and software version control in supply

Algorithm 4: Memory Integrity Checker (with Authenticated Dictionary Abstraction).

Input: $\mathsf{AgreedInitState} = (d_0)$ and the authenticated dictionary verification key vrk.

```
1  Global variable d maintained by a dedicated wire;
2  Func MemInit ():
3  |  d ← d_0 ;                          /* initialize the local digest */
4  Func MemCheck (k, v, π, φ):
5  |  if VerLookup(vrk, d, {k}, {v}, π) or (VerNoKey(vrk, d, {k}, φ) and v = 0)
   |   then
6  |  |  return 1 ;                              /* verification passes */
7  |  end
8  |  return 0
9  Func MemUpdate (k, v', v, π):
10 |  d ← Update(prk, d, {k}, {v}, {v'});
11 |  return 1
```

chain security. Consider a voter registration table in Figure 5-4. It contains three columns — Name, County, and Voted. Assume a secure voting scheme now involves a cryptographic digest of the table to track the voter registration information. Existing cryptographic accumulators would hash each row into a single element in the space and accumulate them naively. They do not provide a handy interface if we want to make queries like "Is there a voter from Middlesex that voted yes?" with cryptographically verifiable proofs.

Name	County	Voted
John	Middlesex	Yes
Alex	Middlesex	No
Bryan	Suffolk	Yes

Figure 5-4: Example Voter Registration Table.

We propose a new cryptographic primitive called *wildcard accumulators*, where, instead of accumulating single elements as a whole, the accumulator is aware of the structure of the data. Namely, it accumulates element vectors, and each dimension is distinguishable from the others. Wildcard accumulators are a generalization of authenticated dictionaries, whereas authenticated dictionaries accumulate key-value pairs, in other words, two-column vectors. Wild accumulators generalize two-column

vectors in authenticated dictionaries to l-column vectors for an arbitrary integer l. With structure preserved, wildcard accumulators support membership proofs and non-membership proofs on a subset of columns.

Wildcard accumulators fit perfectly for the voter registration example in Figure 5-4. We treat each row as a vector of three columns and accumulate them. As a result, we can prove statements like "there is at least one voter from Middlesex who voted yes" and "there is no voter from Suffolk who voted no" while enjoying similar security properties as conventional cryptographic accumulators.

We present two constructions of wildcard accumulators (c.f. Section 5.3.5 and 5.3.6). The first one is based on the RSA accumulators and prime number categorization. It is efficient and constant-sized but requires $O(2^k)$ prime categories. The second one originates from the Tomescu-Xia-Newman authenticated dictionaries (TXN20) [151], which is further based on the Catalano-Fiore vector commitment (CF13) [46, 43]. Its size is linear in the number of columns, and it has succinct proofs and fast verification.

Section 5.3.2 provides discussion on background and related works. Section 5.3.4 formally defines wildcard accumulators. Section 5.3.5 presents a construction of a wildcard accumulator based on the RSA accumulators. This construction is efficient when the number of columns is small. Section 5.3.6 presents another construction based on the TXN20 authenticated dictionaries.

5.3.2 Background

This section provides the definitions of (dynamic) accumulators and discusses related work. For simplicity, we omit the setup interfaces.

Accumulators

A *cryptographic accumulator* commits a set of elements $S = \{x_i\} \subseteq \mathcal{X}$ into a short digest acc_i. Here, $\mathcal{X}$ is the *working domain* or the universe of possible alphabets. For every element x_i, one can efficiently compute a membership proof if $x_i \in S$, or a non-

membership proof if $x_i \notin S$. A *dynamic* cryptographic accumulator supports adding or removing elements from the corresponding set S dynamically. More formally,

Definition 12 (Dynamic Accumulator [42, 61])**.** *A dynamic cryptographic accumulator is a tuple of the following algorithms:*

$\mathsf{Acc.Accumulate}(1^\lambda, S \subseteq \mathcal{X})$*: Given the security parameter* λ *and a subset* S *of the working domain* $\mathcal{X}$*, it returns an accumulator* acc *for the set* S*, as well as auxiliary information* aux*. Usually,* aux *contains the set* S *and useful information for computing/verifying witnesses.*

$\mathsf{Acc.GenMemWit}(x, \mathsf{acc}, \mathsf{aux})$*: Returns a membership witness* π *for an element* x *in the corresponding set* S*. It takes in the element* x*, the accumulator* acc*, and the auxiliary information* aux*.*

$\mathsf{Acc.GenNonMemWit}(x, \mathsf{acc}, \mathsf{aux})$*: Returns a non-membership witness* ϕ *for an element* $x \notin S$*.*

$\mathsf{Acc.VerifyMem}(x, \pi, \mathsf{acc})$*: Verifies a membership witness* π *of an element* x*. It returns* yes *if the verification succeeds, and it returns* no *otherwise.*

$\mathsf{Acc.VerifyNonMem}(x, \phi, \mathsf{acc})$*: Similarly, verifies a non-membership witness* ϕ*.*

$\mathsf{Acc.Add}(\mathsf{acc}, x, \mathsf{aux})$*: Takes in an accumulator* acc *and an element* x*. It returns a new accumulator* acc' *where the corresponding set* $S' = S \cup \{x\}$*, as well as the updated auxiliary information* aux'*.*

$\mathsf{Acc.Del}(\mathsf{acc}, x, \mathsf{aux})$*: Returns a new accumulator by removing* x *from the corresponding set* S*, as well as the updated auxiliary information* aux'*.*

Some schemes require a trusted setup. We omit the technical details of setups in our description to keep the abstraction concise.

Definition 13 (Correctness)**.** *An accumulator scheme* Acc *is* correct *if and only if for any* $S \subseteq \mathcal{X}$*,* $x \in S$*, and* $x' \notin S$*, we have*

$$\Pr\left[\begin{array}{c} (\mathsf{acc}, \mathsf{aux}) \leftarrow \mathsf{Acc.Accumulate}(1^\lambda, S) \\ \pi \leftarrow \mathsf{Acc.GenMemProof}(x, \mathsf{acc}, \mathsf{aux}) \\ \phi \leftarrow \mathsf{Acc.GenNonMemProof}(x', \mathsf{acc}, \mathsf{aux}) \\ \mathsf{Acc.VerifyMemProof}(x, \pi, \mathsf{acc}) = \mathsf{Acc.VerifyNonMemProof}(x, \phi, \mathsf{acc}) = 1 \end{array}\right] = 1.$$

Definition 14 (Soundness)**.** *An accumulator scheme* Acc *is* sound *if and only if, for any p.p.t. adversary A, there exists a negligible function such that the following holds:*

$$\Pr\left[\begin{array}{c}(\mathsf{acc}, x, \pi, \phi) \leftarrow A(1^\lambda) \\ \mathsf{Acc.VerifyMemProof}(x, \pi, \mathsf{acc}) = 1 \\ \mathsf{Acc.VerifyNonMemProof}(x, \phi, \mathsf{acc}) = 1\end{array}\right] = negl(\lambda).$$

For the definition to be non-trivial, the accumulator acc is succinct, namely, the size of acc is sublinear in the size of the set $|\mathsf{acc}| = o(|S|)$. Ideally, an accumulator is constant-sized.

Some schemes support *proof aggregation*. Namely, a number of proofs of different elements regarding *the same* accumulator can be merged into a single proof. For these schemes, we allow Acc.GenMemProof and Acc.GenNonMemProof to take in a set of elements, instead of a single x, to represent the interface of producing aggregated proofs. If the scheme allows aggregation across different accumulators, we call it *cross-aggregatable*.

Further, if the membership witness and non-membership witness are updatable when the corresponding set changes, we call the accumulator scheme an *updatable dynamic accumulator*.

5.3.3 Related Work

Authenticated Data Structures (ADS). Authenticated Data Structures provide a secure way to track a large chunk of data with a short digest. The data owner can delegate the data to an untrusted data holder and keep the digest local. The data holder can compute proofs regarding a particular data item (e.g., whether or not an element exists in the set, or whether or not a position of the data contains the element). With the digest, the data owner can verify proofs. It is difficult for a computationally-bounded adversary to forge a false proof. Examples include cryptographic accumulators, vector commitments, and authenticated dictionaries.

Accumulators. Cryptographic accumulators assume the data is in the form of a set. They provide interfaces for proving the membership of an element regarding a set. We call it a *universal accumulator* [101] if the scheme also supports non-membership proofs. If the scheme allows adding and removing elements, we call it a *dynamic accumulator* [42]. Well-known constructions of cryptographic accumulators include RSA-based accumulators [33], bilinear mapping accumulators [41], and LWE-based accumulators [175].

Vector Commitments. Vector commitments assume the data is a vector of elements. Unlike an accumulator, a vector commitment is a succinct cryptographic digest of a mapping from indices to elements. It provides lookup proofs on the positions. Merkle Tree [108] is one of the most popular vector commitment constructions. Catalano and Fiore [46, 43] proposed an incrementally aggregatable vector commitment scheme.

Authenticated Dictionaries (AD). Authenticated dictionaries further generalize vector commitments by relaxing the indices to arbitrary keys. They track a dictionary, namely, a mapping from a set of keys to elements. Tomescu et al. proposed an append-only authenticated dictionary scheme in [150]. TXN20 [151] constructs an authenticated dictionary scheme that supports incrementally aggregatable proofs. In this work, we propose a new authenticated dictionary scheme that is lightweight and computationally friendly to initialization of large memory. Specifically, the scheme is constant-sized, and the verification takes a constant amount of time assuming the security parameter is fixed. As the scheme supports key non-existence proofs, we can use it to track the changed values only. Therefore, the initial digest can be an empty dictionary regardless of the database size.

Cryptographic Abstractions of Databases. People mainly use existing general constructions like Merkle Trees [83] for data integrity in databases. However, a few special constructions exist. For example, IntegriDB [182] replies on a special authenticated data structure to support expressive SQL-like queries. This chapter presents

wildcard accumulators as an effort toward tailored cryptographic abstractions of relational tables in databases.

5.3.4 Wildcard Accumulators

In this section, we present the formal definition of wildcard accumulators. To extend a universal accumulator to a wildcard accumulator, we make further assumptions on the format of the element domain $\mathcal{X}$. We assume that every element $x \in \mathcal{X}$ can be uniquely decomposed into a vector of l elements $\boldsymbol{x} := (\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(l)})$ where $\boldsymbol{x}^{(i)} \in \mathcal{X}_i$, and $\mathcal{X}_i$ is the alphabet set of the i-th dimension,

$$\mathcal{X} = \mathcal{X}_1 \times \mathcal{X}_2 \times \cdots \times \mathcal{X}_l.$$

From now on, we use $\boldsymbol{x}$ to represent such a vector. We define the *filter space* as $\mathcal{F} := \mathcal{F}_1 \times \mathcal{F}_2 \times \cdots \times \mathcal{F}_l$, where $\mathcal{F}_i := \mathcal{X}_i \cup \{\star\}$. The symbol $(\star)$ in the filter pattern indicates a *wildcard* that matches anything. It is similar to the question mark in wildcard matching, but we use $\star$ instead to avoid confusion with the wildcard symbols (? and $*$) in Section 6.5. A *filter* is a vector $\boldsymbol{f} := (\boldsymbol{f}^{(1)}, \boldsymbol{f}^{(2)}, \ldots, \boldsymbol{f}^{(l)}), \boldsymbol{f}^{(i)} \in \mathcal{F}_i$ from the *filter space*. We say a filter $\boldsymbol{f}$ *matches* a decomposed element $\boldsymbol{x} = (\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(l)})$ if, and only if,

$$\forall i \in [l], \boldsymbol{x}^{(i)} = \boldsymbol{f}^{(i)} \vee \boldsymbol{f}^{(i)} = \star$$

For convenience, we define the match function $\mathsf{match}(\boldsymbol{f}, \boldsymbol{x})$ that maps a filter and an element to either yes or no:

$$\mathsf{match}(\boldsymbol{f}, \boldsymbol{x}) = \begin{cases} \mathsf{yes} & \text{if } \boldsymbol{f} \text{ matches } \boldsymbol{x}, \\ \mathsf{no} & \text{Otherwise.} \end{cases}$$

Similarly, we also define the match function $\mathsf{match}(\boldsymbol{f}, S)$ that maps a filter and a set to either yes or no:

$\mathsf{match}(\boldsymbol{f}, S) = \mathsf{yes}$ if and only if $\exists \boldsymbol{x}_0 \in S$ s.t. $\mathsf{match}(\boldsymbol{f}, \boldsymbol{x}_0) = \mathsf{yes}$.

One might argue that a matching element $x \in S$ with the conventional membership witness π suffices as the wildcard membership witness, since anyone can check x with the filter. We note here that, in some cases, the prover does not want to disclose the exact element nor the number of such elements. A wildcard membership proof does not necessarily reveal the whole element.

Additional Wildcard APIs. We add the following interfaces to extend a dynamic accumulator to a wildcard accumulator:

- $\mathsf{Acc.GenWildcardMemWit}(\boldsymbol{f}, \mathsf{acc}, \mathsf{aux})$: Given a filter f, an accumulator acc, and the auxiliary information, this function computes a wildcard membership witness $\tilde{\pi}$ of $\boldsymbol{f}$, indicating that $\mathsf{match}(\boldsymbol{f}, S) = \mathsf{yes}$.

- $\mathsf{Acc.GenWildcardNonMemWit}(\boldsymbol{f}, \mathsf{acc}, \mathsf{aux})$: It returns a wildcard non-membership witness $\tilde{\phi}$ of the filter $\boldsymbol{f}$, indicating that $\mathsf{match}(\boldsymbol{f}, S) = \mathsf{no}$.

- $\mathsf{Acc.VerifyWildcardMem}(\boldsymbol{f}, \tilde{\pi}, \mathsf{acc})$: This interface takes a filter $\boldsymbol{f}$, a wildcard membership witness $\tilde{\pi}$, and an accumulator acc. It verifies if the witness is valid about $\boldsymbol{f}$ and acc. Similar to $\mathsf{Acc.VerifyMem}$, it returns yes or no.

- $\mathsf{Acc.VerifyWildcardNonMem}(\boldsymbol{f}, \tilde{\phi}, \mathsf{acc})$: Similar to $\mathsf{Acc.VerifyWildcardMem}$, it verifies a wildcard non-membership witness $\tilde{\phi}$.

For simplicity, we unify the two witness generation interfaces, $\mathsf{Acc.GenWildcardMemWit}$ and $\mathsf{Acc.GenWildcardNonMemWit}$ into a single interface $\mathsf{Acc.GenWildcardWit}(\boldsymbol{f}, \mathsf{flag}, \mathsf{acc}, \mathsf{aux})$, where $\mathsf{flag} \in \{\mathsf{yes}, \mathsf{no}\}$. If $\mathsf{flag} = \mathsf{yes}$ (indicating a membership witness), $\mathsf{Acc.GenWildcardWit}(\boldsymbol{f}, \mathsf{yes}, \mathsf{acc}, \mathsf{aux}) := \mathsf{Acc.GenWildcardMemWit}(\boldsymbol{f}, \mathsf{acc}, \mathsf{aux})$. Otherwise, $\mathsf{Acc.GenWildcardWit}(\boldsymbol{f}, \mathsf{no}, \mathsf{acc}, \mathsf{aux}) := \mathsf{Acc.GenWildcardNonMemWit}(\boldsymbol{f}, \mathsf{acc}, \mathsf{aux})$ generates a non-membership witness. We use a new symbol $\tilde{\rho}$ to represent either a wildcard membership witness $\tilde{\pi}$ or a wildcard non-membership witness $\tilde{\phi}$.

Similarly, we unify the two verification interfaces, $\mathsf{Acc.VerifyWildcardMem}$ and $\mathsf{Acc.VerifyWildcardNonMem}$ into a single interface $\mathsf{Acc.VerifyWildcard}(\boldsymbol{f}, \mathsf{flag}, \tilde{\rho}, \mathsf{acc})$. It returns yes if and only if $\tilde{\rho}$ is valid regarding flag.

Definition 15 (Wildcard correctness)**.** *A wildcard accumulator scheme is correct if and only if the following holds for any set $S \subseteq \mathcal{X}$ and any $\boldsymbol{f} \in \mathcal{F}$:*

$$\Pr\left[\mathsf{Acc.VerifyWildcard}(\boldsymbol{f}, \mathsf{flag}, \tilde{\rho}, \mathsf{acc}) = \mathsf{yes} : \begin{array}{c} \mathsf{flag} \leftarrow \mathsf{match}(\boldsymbol{f}, S) \\ \mathsf{acc}, \mathsf{aux} \leftarrow \mathsf{Acc.Accumulate}(1^\lambda, S), \\ \tilde{\rho} \leftarrow \mathsf{Acc.GenWildcardWit}(\boldsymbol{f}, \mathsf{flag}, \mathsf{acc}, \mathsf{aux}) \end{array}\right]$$

$\geq 1 - \mathsf{negl}$.

Definition 16 (Weak Wildcard Soundness)**.** *A wildcard accumulator scheme is weakly wildcard sound if and only if for any adversary A, the following holds:*

$$\Pr\left[\mathsf{Acc.VerifyWildcard}(\boldsymbol{f}, \neg\mathsf{flag}, \tilde{\rho}, \mathsf{acc}) = \mathsf{yes}. : \begin{array}{c} \tilde{\rho}, S, \boldsymbol{f} \leftarrow A(1^\lambda), \\ \mathsf{flag} \leftarrow \mathsf{match}(\boldsymbol{f}, S), \\ \mathsf{acc}, \mathsf{aux} \leftarrow \mathsf{Acc.Accumulate}(1^\lambda, S) \end{array}\right] \leq \mathsf{negl},$$

where the symbol $\neg$ means Boolean negation.

To formalize a stronger notion of soundness, we first introduce the notion of *Consistency* of a set of filter-flag pairs $C = \{(\boldsymbol{f}_i, \mathsf{flag}_i)\}$. We say C is *consistent* if there exists a set S that:

$$\forall i \in |C|, \mathsf{match}(S, \boldsymbol{f}_i) = \mathsf{flag}_i.$$

For example, assume the domain $\mathcal{X} = \{0,1\}^3$. A filter-flag pair set $C = \{((0,1,\star), \mathsf{yes}), ((1,\star,\star), \mathsf{no})\}$ is consistent because a single-element set $S = \{(0,1,0)\}$ satisfies that $match((0,1,\star), S) = \mathsf{yes}$ and $match((1,\star,\star), S) = \mathsf{no}$. However, consider another filter-flag pair set $C' = \{((0,1,\star), \mathsf{yes}), ((0,\star,\star), \mathsf{no})\}$. C' is inconsistent because there does not exist such a set S that satisfies $match((0,1,\star), S) = \mathsf{yes}$ and $match((0,\star,\star), S) = \mathsf{no}$ because any element that matches the filter $(0,1,\star)$ always matches $(0,\star,\star)$.

More formally,

Definition 17 (Consistency)**.** *A set of filter-flag pairs $C = \{(\boldsymbol{f}_i, \mathsf{flag}_i)\} \subseteq \mathcal{F} \times$*

$\{\mathsf{yes}, \mathsf{no}\}$ *is consistent if, and only if, there exists a set* $S \subseteq \mathcal{X}$ *such that* $\forall i \in |C|, \mathsf{match}(S, \boldsymbol{f}_i) = \mathsf{flag}_i$.

If such a set C is *not* consistent, we say that it is *inconsistent*.

Now, we are ready to present the definition of strong wildcard soundness.

Definition 18 (Strong Wildcard Soundness)**.** *An accumulator scheme is strongly wildcard sound if and only if for any adversary A, the following probability is negligible:*

$$\Pr\left[\begin{array}{c} \textit{for each } (\boldsymbol{f}_i, \mathsf{flag}_i) \in C, \\ \mathsf{Acc.VerifyWildcard}(\boldsymbol{f}_i, \mathsf{flag}_i, \tilde{\rho}_i, \mathsf{acc}) = \mathsf{yes}, : (\mathsf{acc}, C = \{(\boldsymbol{f}_i, \mathsf{flag}_i)\}, \tilde{\rho}_i) \leftarrow A(1^\lambda) \\ C \textit{ is inconsistent.} \end{array}\right].$$

Further, we define the privacy of wildcard witnesses. Namely, no information about the actual element(s) that matches the filter pattern or other elements, other than the filter itself and a bit whether the filter matches the set, is revealed by the wildcard witnesses. Before we formally define wildcard witness privacy, we describe two experiments, involving a challenger, an adversary A, and a simulator sim, in Figure 5-5. Then, we say the wildcard accumulator scheme is private when an adversary cannot distinguish between the two games. We adapt the definition from Ghosh et al. [68]. However, we omit the details of the setup and only consider static usages without dynamic updates. Compared to Ghosh et al. [68], we note that a wildcard accumulator is not necessarily zero-knowledge. The two constructions we will present in Section 5.3.5 and Section 5.3.6 are deterministic. Note that our privacy definition does not allow the adversary to choose the set arbitrarily because the adversary can produce the (deterministic) accumulator from the set and compare it with the one received from the challenger or the simulator. A zero-knowledge accumulator has to be randomized. We defer a zero-knowledge wildcard accumulator to future work.

Definition 19 (Wildcard Witness Privacy)**.** *An accumulator scheme is wildcard private if and only if for any adversary A, there exists a simulator* sim *with a trapdoor*

τ, we have

$$\left| \Pr[\mathsf{Real}_A(1^\lambda) = 1] - \Pr[\mathsf{Ideal}_{A,\mathsf{sim}}(1^\lambda) = 1] \right| \leq \mathsf{negl}.$$

$\mathsf{Real}_A(1^\lambda)$:
The challenger performs the setup and uniformly randomly samples a set $S \subseteq \mathcal{X}$ with $|S| = \mathsf{poly}(\lambda)$, and computes $(\mathsf{acc}, \mathsf{aux}) \leftarrow \mathsf{Acc.Accumulate}(1^\lambda, S)$.
The challenger sends acc to A. The following is repeated polynomially many times:

- A sends a filter $\boldsymbol{f}$ to the challenger.
- Let $\mathsf{flag} \leftarrow \mathsf{match}(\boldsymbol{f}, S)$.
- The challenger computes $\tilde{\rho} \leftarrow \mathsf{Acc.GenWildcardWit}(\boldsymbol{f}, \mathsf{flag}, \mathsf{acc}, \mathsf{aux})$ and sends $(\mathsf{flag}, \tilde{\rho})$ to A.

Finally, A outputs a bit b.

$\mathsf{Ideal}_{A,\mathsf{sim}}(1^\lambda)$:
The challenger performs the setup and sends the trapdoor to the simulator. Then, it uniformly randomly samples a set $S \subseteq \mathcal{X}$ with $|S| = \mathsf{poly}(\lambda)$ and initializes an oracle D. The simulator has access to the oracle $D(\boldsymbol{f})$, which returns $\mathsf{match}(\boldsymbol{f}, S)$, where $\boldsymbol{f}$ must have appeared in A's messages. The simulator (without seeing the set S) computes and sends an accumulator acc to A. The following is repeated polynomially many times:

- A sends a filter $\boldsymbol{f}$ to the simulator.
- Let $\mathsf{flag} \leftarrow D(\boldsymbol{f})$.
- The simulator computes $\tilde{\rho}'$, and sends $(\mathsf{flag}, \tilde{\rho}')$ to A.

Finally, A outputs a bit b.

Figure 5-5: Experiments $\mathsf{Real}_A(1^\lambda)$ and $\mathsf{Ideal}_{A,\mathsf{sim}}(1^\lambda)$.

Intuitively, the simulator does not know the set nor the element that matches the filter. If the adversary cannot distinguish between the ideal world and the real world, the wildcard accumulator scheme does not leak extra information about the actual element that matches with the filter except that "an element matches the filter". Similarly, the non-membership witnesses do not leak information of the set except

that "no elements match the filter".

5.3.5 RSA Weak Wildcard Accumulator

This section presents the first construction of a wildcard accumulator based on the RSA assumption (c.f. Definition 9).

Before we dive into the details of our RSA weak wildcard accumulator construction, we recap RSA accumulators previously discussed in Section 5.1.

Given an RSA instance (g, N) and a hash function that maps elements to deterministic and unique prime numbers, an RSA accumulator works as follows.

- $\mathsf{Acc.Accumulate}(1^\lambda, S \subseteq \mathcal{X}) \to := (g^{\prod_{x \in S} \mathsf{hash}(x)}, S) := (\mathsf{acc}, \mathsf{aux})$.
- $\mathsf{Acc.GenMemWit}(x, \mathsf{acc}, \mathsf{aux}) \to g^{\prod_{x' \in \mathsf{aux}, x' \neq x} \mathsf{hash}(x')} := \pi$.
- $\mathsf{Acc.GenNonMemWit}(x, \mathsf{acc}, \mathsf{aux}) \to \mathsf{Bézout}(\mathsf{hash}(x), \prod_{x' \in \mathsf{aux}} \mathsf{hash}(x')) := (A, B) := \phi$, where $\mathsf{Bézout}(x, y)$ computes A, B s.t. $Ax + By = \gcd(x, y)$.
- $\mathsf{Acc.VerifyMem}(x, \pi, \mathsf{acc})$: Check if $\pi^{\mathsf{hash}(x)}$ equals acc.
- $\mathsf{Acc.VerifyNonMem}(x, \phi, \mathsf{acc})$: Checks if $g^{A \cdot \mathsf{hash}(x)} \cdot \mathsf{acc}^B$ equals g.
- $\mathsf{Acc.Add}(\mathsf{acc}, x, \mathsf{aux}) \to (\mathsf{acc}^{\mathsf{hash}(x)}, \mathsf{aux} \cup \{x\})$.
- $\mathsf{Acc.Del}(\mathsf{acc}, x, \mathsf{aux}) \to (g^{\prod_{x' \in \mathsf{aux}, x' \neq x} \mathsf{hash}(x)}, \mathsf{aux} \backslash \{x\})$.

Extend RSA Accumulators to Weakly-Binding Wildcard Accumulators

We extend RSA accumulators to a weak wildcard accumulator based on the construction in [33] and the prime categorization technique in Section 5.1.1.

We call the set of indices $\{i | \boldsymbol{f}^{(i)} \neq \star\}$ as the *supporting indices*, denoted as $\mathsf{SPI}(\boldsymbol{f})$. We define a hash function m that maps a filter $\boldsymbol{f} = (\boldsymbol{f}^{(1)}, \boldsymbol{f}^{(2)}, \ldots, \boldsymbol{f}^{(l)})$ to a product of primes from 2^l categories, namely,

$$\mathsf{m}(\boldsymbol{f}) := \prod_{I \subseteq \mathsf{SPI}(\boldsymbol{f})} \mathsf{Sample}(\lambda, \sum_{i \in I} 2^{i-1}, \boldsymbol{f}^{(I)}),$$

where $\boldsymbol{f}^{(I)}$ denotes the number $\sum_{i\in I} |\mathcal{X}|^i \cdot \boldsymbol{f}^{(i)}$.

Notice that an element $\boldsymbol{x} \in \mathcal{X}$ is also an element in the filter space $\mathcal{F} = \mathcal{F}_1 \times \mathcal{F}_2 \times \cdots \times \mathcal{F}_l$, so m can operate on $\boldsymbol{x}$ as well.

Given an RSA instance (g, N),

- $\mathsf{Acc.Accumulate}(1^\lambda, S \subseteq \mathcal{X}) \to (g^{\prod_{\boldsymbol{x}\in S} \mathsf{m}(\boldsymbol{x})}, S) := (\mathsf{acc}, \mathsf{aux})$.
- $\mathsf{Acc.GenMemWit}(\boldsymbol{x}, \mathsf{acc}, \mathsf{aux}) \to g^{\prod_{\boldsymbol{x}'\in\mathsf{aux}, \boldsymbol{x}'\neq\boldsymbol{x}} \mathsf{m}(\boldsymbol{x}')} := \pi$.
- $\mathsf{Acc.GenNonMemWit}(\boldsymbol{x}, \mathsf{acc}, \mathsf{aux}) \to \mathsf{Bézout}(\mathsf{m}(\boldsymbol{x}), \prod_{\boldsymbol{x}'\in\mathsf{aux}} \mathsf{m}(\boldsymbol{x}')) := (A, B) := \phi$, where $\mathsf{Bézout}(x, y)$ computes A, B s.t. $Ax + By = \gcd(x, y)$.
- $\mathsf{Acc.VerifyMem}(\boldsymbol{x}, \pi, \mathsf{acc})$: Check if $\pi^{\mathsf{m}(\boldsymbol{x})}$ equals acc.
- $\mathsf{Acc.VerifyNonMem}(\boldsymbol{x}, \phi, \mathsf{acc})$: Checks if $g^{A\mathsf{m}(\boldsymbol{x})} \cdot \mathsf{acc}^B$ equals g.
- $\mathsf{Acc.Add}(\mathsf{acc}, \boldsymbol{x}, \mathsf{aux}) \to (\mathsf{acc}^{\mathsf{m}(\boldsymbol{x})}, \mathsf{aux} \cup \{\boldsymbol{x}\})$.
- $\mathsf{Acc.Del}(\mathsf{acc}, \boldsymbol{x}, \mathsf{aux}) \to (g^{\prod_{\boldsymbol{x}'\in\mathsf{aux}, \boldsymbol{x}'\neq x} \mathsf{m}(\boldsymbol{x})}, \mathsf{aux}\backslash\{\boldsymbol{x}\})$.
- $\mathsf{Acc.GenWildcardMemWit}(\boldsymbol{f}, \mathsf{acc}, \mathsf{aux}) \to g^{\prod_{\boldsymbol{x}'\in\mathsf{aux}} \mathsf{m}(\boldsymbol{x}')/\mathsf{m}(\boldsymbol{f})} := \tilde{\pi}$.
- $\mathsf{Acc.GenWildcardNonMemWit}(\boldsymbol{f}, \mathsf{acc}, \mathsf{aux}) \to \mathsf{Bézout}(\mathsf{m}(\boldsymbol{f}), \prod_{\boldsymbol{x}'\in\mathsf{aux}} \mathsf{m}(\boldsymbol{x}')) := (\tilde{A}, \tilde{B}) := \tilde{\phi}$.
- $\mathsf{Acc.VerifyWildcardMem}(\boldsymbol{f}, \tilde{\pi}, \mathsf{acc})$: Check if $\tilde{\pi}^{\mathsf{m}(\boldsymbol{f})}$ equals acc.
- $\mathsf{Acc.VerifyWildcardNonMem}(\boldsymbol{f}, \tilde{\phi}, \mathsf{acc})$: Checks if $g^{\tilde{A}\mathsf{m}(\boldsymbol{f})} \cdot \mathsf{acc}^{\tilde{B}}$ equals g.

Theorem 2. *The RSA-based wildcard accumulator is correct.*

Theorem 3. *The RSA-based wildcard accumulator is weakly binding.*

Theorem 4. *The RSA-based wildcard accumulator is witness private.*

We include the proofs of these three theorems in Appendix A.2.

5.3.6 TXN20-based Wildcard Accumulator

In this section, we build a wildcard accumulator based on an authenticated dictionary scheme, Tomescu-Xia-Newman (TXN20) [151], which is further based on the Catalano-Fiore (CF13) vector commitment scheme [46].

As an overview, we use l instances of authenticated dictionaries $d_1, d_2, \ldots, d_l$. The i-th instance d_i is a digest of the mapping from the hash of a tuple $\boldsymbol{x}$ to the i-th element of the tuple $\boldsymbol{x}^{(i)}$. We denote the i-th mapping as D_i. To provide a wildcard membership witness of a pattern $\boldsymbol{f}$, we find a matching element $\boldsymbol{x}_0$ in the set, and collect the lookup proofs of $\mathsf{hash}(\boldsymbol{x}_0) \to \boldsymbol{x}_\mathbf{0}^{(k)}$ from D_i for every i such that $\boldsymbol{f}^{(i)} = \boldsymbol{x}_\mathbf{0}^{(i)} \neq \star$.

To provide a wildcard non-membership witness, we sample a random linear combination of the authenticated dictionaries corresponding to the columns in which the filter does not have a wildcard symbol, $d = \prod_{f^{(i)} \neq \star} d_i^{r_i}$, where r_i's are random numbers from the verifier challenge or random oracles. Because the construction is linearly homomorphic with respect to values, d is a mapping from the hashes of tuples to the linearly combined values. We then use a new technique of providing non-membership witnesses on the value domain (c.f. Section 5.3.6) to claim that there does not exist a key k such that $D(k) = \sum_{f^{(i)}} D_i(k) \cdot r_i$. Because the random coefficients $\{r_i\}$ are sampled independently randomly, with a high probability, there does not exist a key k such that $D_i(k) = f^{(i)}$ holds for all i.

We first recap on how to build an authenticated dictionary scheme from TXN20.

Recap on the Tomescu-Xia-Newman Authenticated Dictionary Scheme

Public parameters. The public parameters remain $\mathsf{prk} = \mathsf{vrk} = (g, H)$, where g is a generator for the hidden-order group $\mathbb{G}_?$, and H is a CRHF that maps keys (not vector indices) to $(\ell+1)$-bit primes. Values in the dictionary must be ℓ-bit numbers.

Digest. Let D be a dictionary and K be the set of keys with a value $D(k) \neq \perp$ in the dictionary. We often use $e_K = \prod_{k \in K} e_k$ to denote the product of the prime representatives of all keys in K. Let $S = g^{e_K}$ and $S_k = S^{\frac{1}{e_k}}$. The digest is $d = (c, S)$,

where $c = \prod_{(k,v)\in D} (S_k)^v$. Note that the prime numbers e_i are larger than all the values v_i.

Lookup proofs. A proof π_k for k having value $v \neq \perp$ in the dictionary D with digest $d = (c, S)$ consists of two parts. The first part is a commitment Λ_k to D without (k, v) in it, where $\Lambda_k = \prod_{(k',v)\in D, k'\neq k} ((S_{k'})^v)^{\frac{1}{e_k}}$. The second part, is an RSA membership witness S_k for k w.r.t. the RSA accumulator S in the digest, where $S_k = S^{\frac{1}{e_k}}$. As before, to verify the proof, one checks if $S = (S_k)^{e_k}$ and if $c = (S_k)^v(\Lambda_k)^{e_k}$.

Partial Homomorphism. This authenticated dictionary construction is homomorphic for value addition, subtraction, and multiplication with a constant number. Namely, given two digests (c_1, S) and (c_2, S) of two dictionaries D_1 and D_2 with the same sets of keys,

$$c_1 \cdot c_2 = \prod_{(k,v_1)\in D_1} (S_k)_1^v \cdot \prod_{(k,v_2)\in D_2} (S_k)_2^v = \prod_{k:(k,v_1)\in D_1 \wedge (k,v_2)\in D_2} (S_k)^{(v_1+v_2)},$$

corresponding to a dictionary with the same keys but values $D_1(k) + D_2(k)$ for any key k. Similarly, for a digest (c, S) of a dictionary D,

$$c^k = \prod_{(k,v)\in D} (S_k)^{(v\cdot k)},$$

corresponds to a dictionary D' with the same keys but all the values multiplied with the constant k.

Now, we present an important trick to extend TXN20 with non-membership support.

Extend TXN20 with value non-membership proof

We call an authenticated dictionary a *unit authenticated dictionary* on a key set $\mathcal{K} = \{k_i\}$ if it maps from each key k_i to the unit element 1.

Given a dictionary D and a value v, we want to prove that there does not exist a key h s.t. $D(h) = v$. As the TXN20 dictionary constructions are linearly homomor-

phic, this is equivalent to proving that $D' = D/E^v$ does not have zeroes in the value domain, where E is the unit authenticated dictionary, and E^v is the authenticated dictionary with all the values equal to v. We denote the digest of the unit dictionary as (w, S).

Now, we present a trick to prove that a TXN20 dictionary does not contain zero values. We first patch the lookup proof of the element corresponding to e_i by adding an extra element $y := (w/S_i)^{1/e_i} = (\prod_{j \neq i} S_j)^{1/e_i}$ to the proof. Adding this element will not break the correctness or the soundness of the lookup proof.

Theorem 5. *Given an instance of TXN20-based authenticated dictionary* $D = (c, S) = (\prod_i \left(S^{1/e_i}\right)^{v_i}, g^{\prod e_i}) := (g^e, g^{\prod e_i})$. *We have* $\prod v_i \neq 0$ *if and only if*

$$gcd(e, \prod_i e_i) = 1.$$

Therefore, to provide a non-membership proof of a value v on the value domain of the authenticated dictionary $D = (c, S)$, we just need to compute the unit authenticated dictionary $E = (w, S)$, and provide the Bézout coefficients (A, B) such that $A \cdot e + B \cdot \prod_i e_i = 1$, or, equivalently, $(c/w^v)^A \cdot S^B = g$, where $w = \prod_i S^{1/e_i}$ is part of the unit authenticated dictionary. Formally, we let a non-membership proof consist of the following elements:

$$\phi := \left(A, S^B\right) := (A, b).$$

To verify the proof ϕ, we just need to check if $(c/w^v)^A \cdot S^B = g$, and $y^v = (w/S_i)^{1/e_i}$.

Theorem 6 (Strongly Binding)**.** *The non-membership proof of value domain is strongly binding. Namely, for any efficient adversary* $\mathcal{A}$*, it cannot, with non-negligible probability, produce an authenticated dictionary* $D = (S, c, w)$*, a value* v*, a lookup proof* $\pi = (S_i, \Lambda_i, y)$ *on the key pair* (k_i, v)*, and a non-membership proof* $\phi = (A, B)$ *of the value* v*, such that both* π *and* ϕ *are valid, i.e.,*

$$S_i^{e_i} = S, S_i^v \cdot \Lambda^{e_i} = c, (c/w^v)^A \cdot S^B = g.$$

We defer the proofs of Theorem 5 and Theorem 6 to Appendix A.3.

TXN20 Goes Wild(card)

Now, we are ready to present our construction of a wildcard accumulator based on TXN20 [151]. We denote a collision-resistant hash function mapping from the element vectors to prime numbers as cfhash.

Setup The construction requires a general group with unknown order $\mathbb{G}_?$.

Public Information Same as the authenticated dictionary scheme presented in Section 5.3.6, the public information remains $\mathsf{prk} = \mathsf{vrk} = (g, H)$.

Accumulating To accumulate a set of vectors $\mathbf{S} = \{\boldsymbol{x}_i\}$, $\mathsf{Acc.Accumulate}$ takes in $\mathbf{S}$ and some randomness. It will output acc as an ordered list of authenticated dictionaries. The i-th digest to a mapping from $\mathsf{cfhash}(\boldsymbol{x})$ to $\boldsymbol{x}^{(i)}$, namely, $\mathsf{Acc.Accumulate}(1^\lambda, \mathbf{S}) = (\mathsf{acc}, \mathsf{aux})$. The accumulator $\mathsf{acc} = (d_1, \ldots, d_l; S)$, where $S = g^{\prod_{\boldsymbol{x}_j \in \mathbf{S}} \mathsf{cfhash}(\boldsymbol{x}_j)}$ and $d_i = \prod_{\boldsymbol{x}_j \in \mathbf{S}} (S^{1/\mathsf{cfhash}(\boldsymbol{x}_j)})^{\boldsymbol{x}_j^{(i)}}$. The auxiliary information aux remains the set itself $\mathbf{S}$.

Membership Witness To provide membership witness of a vector $\boldsymbol{x}$, we generate lookup proofs of the key value pairs $\mathsf{cfhash}(\boldsymbol{x})$ to each dimension $\boldsymbol{x}^{(i)}$.

$\mathsf{Acc.GenMemWit}(\boldsymbol{x}, \mathsf{acc}, \mathsf{aux}) = (\Lambda_1, \Lambda_2, \ldots, \Lambda_l; S_{\boldsymbol{x}})$, where $S_{\boldsymbol{x}} = S^{1/\mathsf{cfhash}(\boldsymbol{x})}$ and

$$\Lambda_i = \prod_{\boldsymbol{x}' \neq \boldsymbol{x}} (S_{\boldsymbol{x}}^{1/\mathsf{cfhash}(\boldsymbol{x}')})^{\boldsymbol{x}'^{(i)}}. \tag{5.1}$$

For a constant sized proof, we aggregate $\Lambda_1, \Lambda_2, \ldots, \Lambda_l$ into a single group element by a random linear combination $\Lambda = \prod \Lambda_i^{r_i}$, where r_i's are random numbers.

Note that, we can use a Proof of Knowledge of Co-prime Roots (PoKCR) [33] to aggregate the proofs together. We discuss this technique in Appendix B.

Lastly, we include an extra element $y = (w/S_{\boldsymbol{x}})^{1/\mathsf{cfhash}(\boldsymbol{x})}$ in the membership witness, where w is the unit dictionary digest.

Verifying Membership Witness Intuitively, verifying the membership proofs consists of verifying the individual lookups. One checks whether $S = (S_{\boldsymbol{x}})^{\mathsf{cfhash}(\boldsymbol{x})}$

and $d_i = (S_{\boldsymbol{x}})^{\boldsymbol{x}^{(i)}}(\Lambda_i)^{\mathsf{cfhash}(\boldsymbol{x})}$ for all $i \in [l]$. For the aggregated witness proof, the verifier just needs to check whether or not $\prod d_i^{r_i} = (S_{\boldsymbol{x}})^{\sum r_i \boldsymbol{x}^{(i)}} \cdot \Lambda^{\mathsf{cfhash}(\boldsymbol{x})}$.

Non-membership Witness For the non-membership witness, we follow a similar idea as the membership witness. Namely, we perform a random linear combination on the l digests and repeat the trick described in Section 5.3.6. Specifically, we compute $d = \prod d_i^{r_i}$ and the target value $v = \sum r_i \cdot \boldsymbol{x}^{(i)}$. The non-membership witness is the Bézout coefficient (A, B) of e and $\prod e_i$, where e is the exponent of (d/w^v) and w is the unit authenticated dictionary. It is straightforward to see that $(d/w^v)^A \cdot S^B = g$.

Verifying Non-membership Witness Given a non-membership witness (A, B), we check whether $(c/w^v)^A \cdot S^B = g$, where $v = \sum r_i \cdot \boldsymbol{x}^{(i)}$ and w is the unit authenticated dictionary.

Wildcard Membership Witness Similar to a membership witness, a wildcard membership witness consists of a number of lookup proofs. Different from the former, the latter only contains the dimensions where the filter is not the wildcard symbol. Denote the element in the set that matches the filter as $\boldsymbol{x}$, we have $\mathsf{Acc.GenWildcardMemWit}(\boldsymbol{f}, \mathsf{acc}, \mathsf{aux}) = (\Lambda_i | \boldsymbol{f}^{(i)} \neq \star; S_{\boldsymbol{x}})$, where

$$\Lambda_i = \prod_{\boldsymbol{x}' \neq \boldsymbol{x}} (S_{\boldsymbol{x}}^{1/\mathsf{cfhash}(\boldsymbol{x}')})^{\boldsymbol{x}'^{(i)}}. \tag{5.2}$$

Likewise, we can perform a linear combination to aggregate $\{\Lambda_i\}$ into a single group element: $\Lambda = \prod_{\boldsymbol{f}^{(i)} \neq \star} \Lambda_i^{r_i}$. We also include an extra element $y = (w/S_{\boldsymbol{x}})^{1/\mathsf{cfhash}(\boldsymbol{x})}$ in the membership witness, where w is the unit dictionary digest.

Verifying Wildcard Membership Witness The verification process is exactly the same as verifying the ordinary membership witness, except that the verifier also checks whether or not the witness contains all the Λ_i where $\boldsymbol{f}^{(i)} \neq \star$.

Wildcard Non-Membership Witness Similar to a non-membership witness, a wildcard non-membership witness consists of the Bézout coefficients (A, B) s.t. $(d/w^v)^A \cdot S^B = g$, where $d = \prod_{\boldsymbol{f}^{(i)} \neq \star} d_i^{r_i}$ and $v = \sum_{\boldsymbol{f}^{(i)} \neq \star} r_i \boldsymbol{x}^{(i)}$.

Verifying Wildcard Non-Membership Witness To verify the wildcard non-membership witness, we check whether $(d/w^v)^A \cdot S^B = g$.

Adding a Row This construction inherits the dynamic properties from the TXN20 authenticated dictionary [151]. To add an element $\boldsymbol{x}$ to an existing wildcard accumulator $(d_1, d_2, \ldots, d_l; S)$, we compute $S' = S^{\mathsf{cfhash}(\boldsymbol{x})}$ and $d_i' = d_i^{\mathsf{cfhash}(\boldsymbol{x})} \cdot S^{\boldsymbol{x}^{(i)}}$ for all $i \in [l]$.

It turns out adding and removing columns are extremely straightforward. To add a column, one simply adds a new empty authenticated dictionary instance. To remove a column, one removes the corresponding authenticated dictionary instance.

As a final remark, our construction also inherits the incremental aggregatability property of [151], and it can extend to support append-only proofs as in [151]. The randomness used in linear combinations makes the scheme interactive. Fiat-Shamir heuristics can replace the randomness with oracle queries and produce a non-interactive variant of the scheme.

Theorem 7. *The TXN20-based wildcard accumulator is correct.*

Theorem 8. *The TXN20-based wildcard accumulator is strongly binding.*

Theorem 9. *The TXN20-based wildcard accumulator is witness private.*

We defer the proofs of these three theorems to Appendix A.3.

5.4 Multi-Column Memory Integrity

This section shows how the wildcard accumulators bring extra features to memory integrity checking.

Without loss of generality, we assume the database only has one relational table. We can always concatenate the columns of different tables to merge them into a single one. Merging the columns will not increase the sum of digest sizes since both constructions presented in Section 5.3.5 and Section 5.3.6 are not super-linear regarding the number of columns. Alternatively, we can use a dedicated digest for each table.

Denote l as the number of columns in the table. Intuitively, we follow the design in Section 5.2, but replace the authenticated dictionary with an l-column wildcard accumulator. The provider and the checker take a filter as the input instead of an element. This interface change enables queries that specify only a subset of columns.

Before we present the details of the multi-column memory integrity provider and checker, we define a useful property.

For the RSA-based wildcard accumulator, an observation is that the membership witness of an element $\boldsymbol{x}$ is a wildcard accumulator of all the rows except $\boldsymbol{x}$. For the TXN20-based construction, a similar observation holds.

To capture this property, we present the definition of *membership witness reflexiveness*.

Definition 20 (Membership-Witness Reflexiveness)**.** *We call an accumulator scheme membership-witness reflexive, if and only if for all set S,* $\mathsf{Acc.GenMemWit}(x, \mathsf{Acc.Accumulate}(S))$ *contains* $\mathsf{Acc.Accumulate}(S \backslash \{x\})$*. Similarly, we call a wildcard accumulator scheme membership-witness reflexive, if and only if for all set S,* $\mathsf{Acc.GenWildcardMemWit}(x, \mathsf{Acc.Accumulate}(S))$ *contains* $\mathsf{Acc.Accumulate}(S \backslash \{x\})$*.*

5.4.1 Multi-Column Memory Integrity Provider

Algorithm 5 shows the multi-column provider using a wildcard accumulator.

Since a filter might match a number of rows in the table, there are two possible use cases of reading values. The first one is to query *any* one element matching the filter. The second one is to query *all* the elements matching the filter. Different from the GenReadProof function in Algorithm 1, we provide two separate interfaces for these two cases, GenReadProofAny and GenReadProofAll. Both accept a filter when

generating the read proof. Namely, the transaction can get a verifiable query on a subset of columns.

The logic of GenReadProofAny is basically the same as GenReadProof in Algorithm 1. However, GenReadProofAll is more complicated. We not only need to show that the returned rows are in the table (*inclusiveness*), but also have to prove that no other matching rows exist besides the ones in the result (*completeness*). Therefore, GenReadProofAll first computes an aggregated membership witness π to show the inclusiveness. Then, it computes a non-membership proof $\tilde{\phi}$ of $\boldsymbol{f}$regarding the digest with all the matching rows removed to show the completeness. If we assume membership-witness reflexiveness, this digest is simply π. Finally, it returns a tuple $(\pi, \tilde{\phi})$ as the proof.

When updating the digest in UpdateWrite, the pseudocode uses Acc.Del and Acc.Add. In practice, there are better ways to handle the update.

For example, the RSA-based wildcard accumulator is membership-witness reflexive. We can perform the update similar to UpdateWrite in Algorithm 1. The new digest is the membership witness to the power of $\mathsf{m}(\boldsymbol{x}')$. Intuitively, we are adding $\boldsymbol{x}'$ to the rows corresponding to the membership witness, namely, the rows with the previous row $\boldsymbol{x}$ removed.

Since the CF13 construction is also membership-witness reflexive, we perform Acc.Add on the witness. Specifically, the new digest $(\{d_i'\}; S')$ satisfies

$$S' = S_{\boldsymbol{x}}^{\mathsf{cfhash}(\boldsymbol{x}')},$$

$$d_i' = \Lambda_{\boldsymbol{x}}^{\mathsf{cfhash}(\boldsymbol{x}')} \cdot S^{\boldsymbol{x}'(i)}, \forall i \in [l].$$

If the identifier is the same, namely, $\mathsf{cfhash}(\boldsymbol{x}) = \mathsf{cfhash}(\boldsymbol{x}')$, the update is extremely simple — we compute a vector $\delta_{\boldsymbol{x}} = \boldsymbol{x}' - \boldsymbol{x}$ and update the digest through the homomorphism.

5.4.2 Multi-Column Memory Integrity Checker

As shown in Algorithm 6, the multi-column checker is divided into two interfaces, MemCheckAny and MemCheckAll. Both MemCheckAny and MemCheckAll take in a filter instead of a key to keep the interface consistent with the multi-column provider.

MemCheckAny is similar to the MemCheck function in Algorithm 3, except that it also checks whether the claimed value $\boldsymbol{x}$ actually matches the filter $\boldsymbol{f}$. In contrast, MemCheckAll does several checks. First, it checks whether there are actually no rows matching the filter or the default value matches the filter. If this is the case, the checker verifies the wildcard non-membership proof and returns the verification result. Otherwise, the set $\{\boldsymbol{x}\}$ cannot be empty. It iterates through all the claimed values to make sure all $\boldsymbol{x}$ actually match the filter $\boldsymbol{f}$. Next, the checker verifies the inclusiveness proof and the completeness proof.

The MemUpdate function follows the design of UpdateWrite to make sure that the local digest gets updated accordingly. In practice, MemUpdate is computationally lightweight. As described in Section 5.4.1, MemUpdate utilizes the read proof to replace the row.

Besides the multi-column support, using wildcard accumulators also enables convenient index operations like conditional scan queries, insertion, and deletion. We will discuss them in Chapter 6.4.

Algorithm 5: Multi-column Memory Integrity Provider with initial database state agreed as D_0.

Input: D_0

Func InitProvider ():
  acc $\leftarrow$ Acc.Accumulate($1^\lambda, D_0$) ;
  $D \leftarrow D_0$; /* initialization */
Func GenReadProofAny $(\boldsymbol{f}, \boldsymbol{x})$:
  if $\boldsymbol{f}$ *matches* D **then**
    /* generate the lookup proof */
    return Acc.GenWildcardMemWit($\boldsymbol{f}$, acc, aux)
  else
    /* non-existence proof */
    return $(A, B) \leftarrow$ Acc.GenWildcardNonMemWit($\boldsymbol{f}$, acc, aux)
  end
Func GenReadProofAll $(\boldsymbol{f}, \{\boldsymbol{x}\})$:
  if $\boldsymbol{f}$ *matches* D **then**
    /* generate the aggregated inclusiveness proof */
    $\pi \leftarrow$ Acc.GenMemWit($\{\boldsymbol{x}\}$, acc, aux);
    /* generate the completeness proof (assuming
       membership-witness reflexiveness) */
    $\tilde{\phi} \leftarrow$ Acc.GenNonMemWit($\boldsymbol{f}, \pi$, aux);
    return $(\pi, \tilde{\phi})$;
  else
    return $(A, B) \leftarrow$ Acc.GenWildcardNonMemWit($\boldsymbol{f}$, acc, aux);
     /* non-existence proof */
  end
Func UpdateWrite $(\boldsymbol{x}, \boldsymbol{x}')$:
  /* Depending on the construction, better approaches exist */
  acc $\leftarrow$ Acc.Add(Acc.Del($\boldsymbol{x}$), $\boldsymbol{x}'$);
  return

Algorithm 6: Multi-Column Memory Integrity Checker (inside the Wrapped Transaction) with initial database state acc_0.

Input: acc_0

Global variable acc maintained by a dedicated wire;
Func MemInit ():
  acc $\leftarrow \mathsf{acc}_0$; `/* initialize the local digest */`
Func MemCheckAny $(\boldsymbol{f}, \boldsymbol{x}, \tilde{\pi}, \tilde{\phi})$:
  if (Acc.VerifyWildcardMem$(\boldsymbol{f}, \tilde{\pi}, \mathsf{acc})$ *and* $\boldsymbol{f}$ *matches* $\boldsymbol{x}$)) *or* (Acc.VerifyWildcardNonMem$(\boldsymbol{f}, \tilde{\phi}, \mathsf{acc})$ *and* $v = 0$) **then**
    return 1; `/* verification passes */`
  end
  return 0
Func MemCheckAll $(\boldsymbol{f}, \{\boldsymbol{x}\}, \pi, \tilde{\phi})$:
  if (Acc.VerifyWildcardNonMem$(\boldsymbol{f}, \tilde{\phi}, \mathsf{acc})$ *and* $(\{\boldsymbol{x}\} = \emptyset)$ *or* $\boldsymbol{f}$*matches the default value*) **then**
    return 1; `/* verification passes */`
  end
  if $\{\boldsymbol{x}\} = \emptyset$ **then**
    `/* At this stage,` $\{\boldsymbol{x}\}$ `cannot be empty */`
    return 0;
  end
  for $\boldsymbol{x} \in \{\boldsymbol{x}\}$; `/* Check if the rows actually match the filter */`
  do
    if $\boldsymbol{x}$ *does not match* $\boldsymbol{f}$ **then**
      return 0
    end
  end
  `/* Check the inclusiveness and completeness */`
  if Acc.VerifyMem$(\{\boldsymbol{x}\}, \pi, \mathsf{acc})$ *and* Acc.VerifyWildcardNonMem$(\boldsymbol{f}, \tilde{\phi}, \pi)$ **then**
    return 1;
  end
  return 0
Func MemUpdate $(\boldsymbol{x}, \boldsymbol{x}')$:
  acc = Acc.Add (Acc.Del ($\boldsymbol{x}$), $\boldsymbol{x}$');
  return 1

Chapter 6

A Single-Threaded Verifiable DBMS

Now, we are ready to show the design of Litmus, a verifiable database management system. We first present a single-threaded baseline for pedagogical purposes in this chapter. It consists of a server and a single client. We first describe the server in Section 6.1 and the client in Section 6.2. Next, Section 6.3 discusses why the design provides atomicity and isolation verification. After that, Section 6.4 talks about how to support index operations, namely, insertion, deletion, and range scans. Finally, we discuss wildcard text search in Section 6.5.

The baseline system provides verifiability of atomicity and isolation properties. Although Litmus's server can work with any concurrency control algorithm if it can access transactions' interleaving information, we use single-threaded Two-Phase Locking (2PL) to simplify our discussion.

Later, we will extend it to support multi-core machines for better performance in Chapter 8. We also present an interactive design in Section 9.1. Then, we show how to integrate both the batched and interactive designs to build a hybrid verification database management system in Section 9.2. Finally, we present the evaluation results in Chapter 10.

6.1 Server

This section describes the server of the system. We explained in Section 4.1 that the server consists of four components:

1. The memory integrity provider;
2. The transaction wrapper;
3. The circuit compiler and circuit matcher;
4. The normal database.

Throughout this section, we present these components in a technical and detailed way. First of all, we have shown the memory integrity in Section 5.2. We describe how the transaction wrapper works in Section 6.1.1. Then, Section 6.1.2 talks about the circuit compiler. Lastly, Section 6.1.3 connects the components together.

6.1.1 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 (explained in Section 4-1) from $\{T_i\}$ by chaining them sequentially and inserting memory integrity checking code before each transaction starts. Section 4.1 explained a wrapped transaction with a simple example. Now, we formally describe the transaction wrapper in Algorithm 7. First, it constructs a graph representing transactions and their partial orders decided by the concurrency control 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. For each transaction T_i in the list, it checks whether the corresponding read values provided by the input are correct using the memory integrity proofs. It runs the transaction with the read values. While running the transactions, it collects the written values and updates

Algorithm 7: The Serial Transaction Wrapper

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

the local memory digest accordingly. If any of the memory integrity checks fails, the return value of the wrapped transaction will start with a 0 bit if evaluated correctly by the server.

The wrapped transaction does not necessarily have to represent a sequential chain of transactions. As we will discuss in Chapter 8, forming a wide network of transactions is better for performance. The shape of the returned wrapped transaction is critical for exploiting parallelism.

6.1.2 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 LLVM-like representations) into a Rank-1 Constraint System (discussed in Section 2.4). A carefully designed circuit compiler can optimize the structure of the circuit without changing the underlying logic. This process is similar to what modern compilers do to optimize programs.

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. The circuit matcher does two things.

- First, it checks whether the logic of the transactions is consistent with that in the wrapped transaction.
- Second, the circuit matcher checks if the memory integrity checker in the circuit is correctly plugged in.

Generally, deciding if a description of a Turing machine satisfies a certain property can be reduced to the famous *halting problem*, which is undecidable. Fortunately, in our system, we have the freedom to force the server to apply a deterministic circuit compiler. One can reduce these two tasks to pattern matching problems because the same transaction logic would be compiled into the same circuit. A known memory integrity checker also results in a known circuit description. Later, in Section 8.1, we will show how to extend Litmus by using a deterministic concurrency control 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 write-sets do not depend on the read set). Instantiations of such deterministic circuit compilers exist and can be re-purposed for our verifiable DBMS [39, 110].

6.1.3 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 with the interleaving). Furthermore, the dependency information can serve as hints to the verifiable computation 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 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. This partial order 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 finish, 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 whether 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 cir-

cuit 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.

Note that the RuntimeTraces is an analogy of database logging. To see why this is true, both RuntimeTraces and the logging stream contain information to "redo" the transactions in an order consistent with the ground truth. In Section 11.5, we will revisit this observation and show how to migrate techniques in logging to efficiently collecting runtime traces.

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 whether the proof π is valid with respect to y (i.e., the circuit output contains a 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 Section 6.1.1 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 server evaluates the wrapped transaction 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 by a committed transaction. 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 with the Two-Phase Locking concurrency control algorithm. By the correctness of the concurrency control algorithm, the transaction interleaving is serializable. We include a proof in Appendix A.4.

6.4 Supporting Index Operations

Although our discussion has focused on *read* and *write* operations, Litmus can also support *insertion, deletion*, and *range scan* operations.

Both *insertion* and *deletion* are naturally supported by the dynamic properties of the authenticated data structures. Concretely, to add or delete an element x, both the server and the circuit update the digest by calling $\mathsf{Acc.Add}$ or $\mathsf{Acc.Del}$. For membership-witness reflexive constructions, the verifiable DBMS deletes an element by simply taking a valid witness proof of a row $\boldsymbol{x}$ and using the witness as the new digest.

For *range scans*, we extend our authenticated dictionary to support lookup proofs for range queries by using wildcard accumulators (Section 5.3). We assume the size of the address space is 2^l for simplicity. Instead of accumulating key pairs (k, v), we accumulate $(k_1, k_2, \ldots, k_l, v)$, where $k_i \in \{0, 1\}$, and k_i's are the binary decomposition of k. We let the server build an interval tree on $[0, 2^l - 1]$. Each node of the interval tree covers a range specified by a wildcard filter with a number of $\star$'s as its suffix. Specifically, nodes on the i-th level represent ranges $I_{i,j} := [j \cdot 2^i, (j+1) \cdot 2^i - 1]$ for $j \in [0, 2^{(l-i)} - 1]$. The range of the j-th node in the i-th level corresponds to a wildcard filter that looks like

$$\boldsymbol{f}_{i,j} = (b_1, b_2, \ldots, b_{l-i}, \overbrace{\star, \ldots, \star}^{\text{In total } i \text{ } \star\text{'s}}),$$

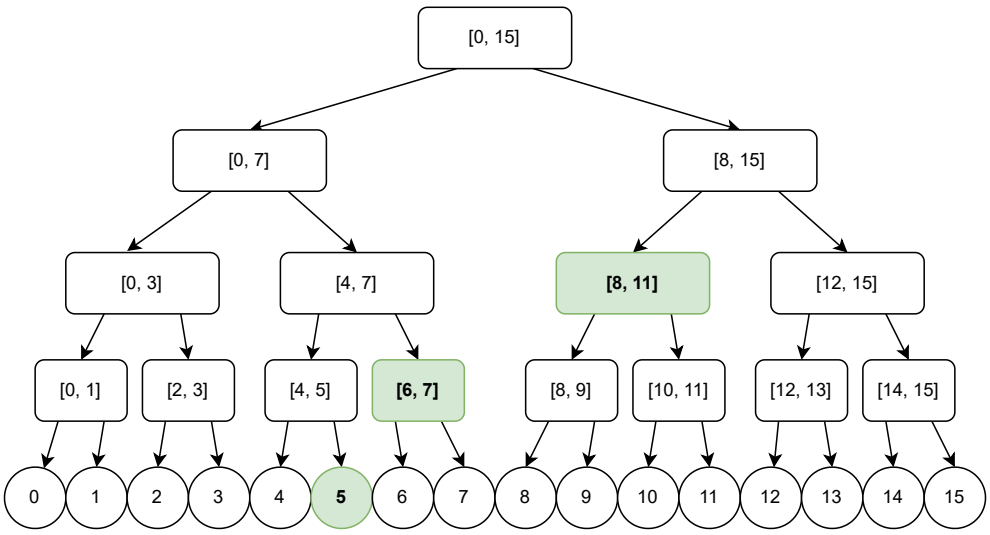


Figure 6-1: **Example Range Scan Decomposition** — A range $[5, 11]$ can be decomposed into three node ranges (colored in green).

where b_i's are the binary representation of j.

Whenever there is a range query on a consecutive range $[a, b] \subseteq [0, 2^l - 1]$, we decompose the interval into no more than $2l$ nodes of the interval tree. For example, $[5, 11]$ when $l = 4$ is equivalent to the union of three nodes $[5, 5], [6, 7], [8, 11]$, as shown in Figure 6-1.

For each node's filter $\boldsymbol{f}_{i,j}$, the server returns a list of rows $\{\boldsymbol{x}_t\}$ in the range $I_{i,j}$, together with a proof produced by GenReadProofAll in Algorithm 5. Concretely, the proof consists of two parts:

- For inclusiveness, an aggregated membership proof shows that the rows $\{\boldsymbol{x}_t\}$ exist in the table:

$$\pi_{i,j} \leftarrow \mathsf{Acc.GenWildcardMemWit}(\{\boldsymbol{x}_t\}, \mathsf{acc}, \mathsf{aux}).$$

- For completeness, a wildcard non-membership proof shows that the digest corresponding to the rest of the rows does not contain any rows that match the filter $\boldsymbol{f}_{i,j}$. Fortunately, both the RSA-based wildcard accumulator and the TXN20-based wildcard accumulator are membership-witness reflexive. All we need to do is to provide a non-membership proof of the filter $\boldsymbol{f}_{i,j}$ regarding $\pi_{i,j}$.

StudentID	Name	Grade
6	Alex	5.0
7	Bob	5.0
11	Cara	5.0

Figure 6-2: Example Multi-Column Table

Note that insertion, deletion, and range scans are sometimes called "index operations" because they are related to changing the indices in modern parallel DBMSs. For example, adding a row changes the index (likely a B-Tree), and therefore requires an exclusive lock on the index when using Two-Phase Locking. In our case, we don't need to involve the indices for the purpose of proving transactional properties because the statement we are proving does not model indices. In other words, indices are auxiliary structures to help modern databases improve their performance, but they are not fundamentally necessary for functionality.

Lastly, a similar technique applies for multi-column tables as well. We assume the range scan target is a numerical column. To use the technique described above, we add l new columns to store the binary representation $(k_1, k_2, \ldots, k_l)$. Figure 6-2 shows a table with columns **(StudentID, Name, Grade)**, where StudentID is four bits long. We use a wildcard accumulator with columns **(StudentID, StudentID_1, StudentID_2, ..., StudentID_6, Name, Grade)**. Note that we keep the original StudentID to handle exact matching queries like "`SELECT * FROM students WHERE StudentID=11`". Figure 6-3 shows the columns in the wildcard accumulator. For example, consider a range scan query "select * from **students** where $6 \leq$ **StudentID** ≤ 7". The interval $[6, 7]$ is decomposed into a single filter $(0, 1, 1, \star)$. The query is in fact converted into a wildcard query "`select * from students where StudentID_1=0 and StudentID_2=1 and StudentID_3=1`". The provider would return the two qualified rows, Alex and Bob, their membership proofs regarding the digest, and a non-membership proof of the filter $\boldsymbol{f} = (\star, \star, \star, 0, 1, 1, \star)$ regarding the rest of the rows.

StudentID	Name	Grade	StudentID_1	StudentID_2	StudentID_3	StudentID_4
6	Alex	5.0	0	1	1	0
7	Bob	5.0	0	1	1	1
11	Cara	5.0	1	0	1	1

Figure 6-3: Example Multi-Column Table After Adding Auxiliary Columns

6.5 Wildcard Search

We use a similar approach to the range scan to support wildcard searches on a text column. Assume the column has at most l characters. We add l auxiliary columns in the wildcard accumulator element format. Each contains a single character or an empty space. For example, the table in Figure 6-2 is padded into the table in Figure 6-4.

Conventionally, a wildcard search uses two types of wildcards. A question mark (?) replaces a single character, while an asterisk ($*$) replaces zero or several characters.

Given a pattern described with wildcards, we recursively decompose it into a number of wildcard accumulator filters. Algorithm 9 shows the decomposition algorithm. The entrance function **WildcardQuery** has a single input, the pattern p. It calls the recursive function **WildcardQueryRec** with p and the column width l. **WildcardQuery** will eventually return a set of wildcard filters.

The **WildcardQueryRec** function has two inputs, the pattern p and the remaining length r. In the beginning, it checks the simple case, namely, whether p does not contain wildcards. If this is true, it returns a set composed of p only if the remaining length is precisely the length of p. Otherwise, it returns an empty set because the length does not match.

Otherwise, let i be the index of the first wildcard. If $p[i]$ is a question mark (?), $p[i]$ represents exactly one character. We recursively call **WildcardQueryRec** with the rest part after $p[i]$ and the remaining length $r - i$ and denote the result set as F. For each filter $\boldsymbol{f}$ in F, we append the prefix of p till the first wildcard and a wildcard accumulator symbol ($\star$) to the left of $\boldsymbol{f}$. For example, **WildcardQueryRec**(C?ra, 4) will return $\{(C, \star, r, a)\}$. Finally, **WildcardQueryRec** returns the set of new filters.

If $p[i]$ is an asterisk ($*$), it might match any number of characters. Therefore,

StudentID	Name	Grade	Name_1	Name_2	Name_3	Name_4
6	Alex	5.0	A	l	e	x
7	Bob	5.0	B	o	b	
11	Cara	5.0	C	a	r	a

Figure 6-4: Example Multi-Column Table

we iterate on the number of characters this asterisk matches and use the variable j to represent this number. Note that j can be as small as 0, meaning this asterisk matches nothing, or as large as $r - i + 1$, filling all the remaining $r - i + 1$ positions. For each j, we recursively call **WildcardQueryRec**$(p[i + 1 \dots], r - i - j + 1)$ to get the filters of the suffix $p[i + 1 \dots]$ with the remaining length of $r - i - j + 1$ characters. Similarly, we append the prefix of p till the wildcard symbol and j's wildcard accumulator symbols ($\star$) to each filter. Finally, **WildcardQueryRec** returns the set of appended filters. For example, **WildcardQueryRec**($*$ra$*$, 4) will return $\{(\star, \star, r, a), (\star, r, a, \star), (r, a, \star, \star)\}$.

Note that it is okay for **WildcardQueryRec** to take an empty p. It will return a set with an empty filter when r is zero. This return value is *different* from an empty set.

After getting the decomposed filters, we query each of them via the memory integrity and merge the results together. For example, the pattern 'C?ra' is decomposed into a single filter $(C, \star, r, a)$. Applying this filter to the table in Figure 6-4 returns a single row with Cara. As another example, the pattern '$*$ra$*$' is decomposed into three filters $\{(\star, \star, r, a), (\star, r, a, \star), (r, a, \star, \star)\}$. Only $(\star, r, a, \star)$ matches a row, the row with Cara; The other two filters match nothing.

Finally, we remark that this algorithm runs in exponential time in the number of asterisks in the searching pattern. Fortunately, in practice, we only use a few asterisks in wildcard search. Both the techniques in Section 6.4 and Section 6.5 add a linear number of columns to the wildcard accumulator. However, the RSA-based wildcard accumulator incurs an exponential computation overhead in the number of columns. Therefore, we suggest using the TXN20-based construction with these techniques.

Algorithm 8: The Workflow of the Server

Input: A previously agreed initial statue of the database $\mathsf{AgreedInitState}$

Initialize $\mathsf{DB.Data}[*].\mathsf{LastReader} \leftarrow \mathsf{DB.Data}[*].\mathsf{LastWriter} \leftarrow \emptyset$;
Upon receiving message $(\mathsf{MSG\_TXN}, \{T_i\})$ from the client:
    Initialize the memory accumulator
     $\mathsf{acc} \leftarrow \mathsf{Acc.Accumulate}(\mathsf{AgreedInitState})$;
    Initialize the proof list $\mathsf{proofList} \leftarrow []$, the read values $\mathsf{readList} \leftarrow []$;
    $\mathsf{DB.Run}\ (\{T_i\})$;
        When T_i reads an item $\mathsf{DB.Data}[k]$:
            Acquire the shared read lock on $\mathsf{DB.Data}[k]$;
            Append $(\mathsf{DB.Data}[k].\mathsf{LastWriter} \rightarrow T_i)$ to $\mathsf{RuntimeTraces}$;
            Append T_i to $\mathsf{DB.Data}[k].\mathsf{LastReader}$;
            Append $(k, \mathsf{DB.Data}[k])$ to $T_i.\mathsf{read\_set}$;
            Append $\mathsf{GenReadProof}(\mathsf{acc}, k, \mathsf{DB.Data}[k])$ to $T_i.\mathsf{proofs}$;
        When T_i writes an item $\mathsf{DB.Data}[k]$ with a new value v:
            Acquire the exclusive write lock on $\mathsf{DB.Data}[k]$;
            Append $(\mathsf{DB.Data}[k].\mathsf{LastWriter} \rightarrow T_i)$ and
             $(\mathsf{DB.Data}[k].\mathsf{LastReader} \rightarrow T_i)$ to $\mathsf{RuntimeTraces}$;
            $\mathsf{DB.Data}[k].\mathsf{LastWriter} \leftarrow T_i$;
            $\mathsf{DB.Data}[k].\mathsf{LastReader} \leftarrow \emptyset$;
            Call $\mathsf{UpdateWrite}(\mathsf{acc}, k, v)$;
        When a transaction T_i commits:
            Append $T_i.\mathsf{proofs}$ to $\mathsf{proofList}$;
            Append $T_i.\mathsf{read\_set}$ to $\mathsf{readList}$;
        When the database finishes execution, get the runtime traces
         $\mathsf{RuntimeTraces} = \{(T_i \rightarrow T_j)\}_{i,j}$ and the final output y;
    $C = \mathsf{CircuitCompiler}(\mathsf{TransactionWrapper}(\{T_i\}, \mathsf{RuntimeTraces}))$;
    Send $(\mathsf{MSG\_WRTXN}, C)$ to the client;
Upon receiving message $(\mathsf{MSG\_PKEY}, \sigma)$ from the third-party:
    Initiate the prover $\mathsf{VC.Prove}$ on C and σ;
    Feed $\mathsf{readList}$ and $\mathsf{proofList}$ as circuit inputs to the prover;
    Feed $\mathsf{RuntimeTraces}$ as auxiliary information to the circuit to the prover;
    Get the proof π from the prover and send (π, y) to the client;

Algorithm 9: Wildcard Pattern Decomposition

```
1  Func WildcardQuery(p):
2      return WildcardQueryRec(p, l);
3  Func WildcardQueryRec(p, r):
4      if p does not contain '?' or '∗' then
5          if r < len(p) or r > len(p) then
6              return ∅;
7          end
8          return {p};
9      else
10         Let p[i] as the first wildcard;
11         if p[i] =? then
12             F ← WildcardQueryRec(p[i + 1 . . . ], r − i);
13             F′ ← ∅;
               /* We concatenate the first parts of p and '⋆' to all
                  the filters produced by the suffix                  */
14             for f ∈ F do
15                 Let f′ ← p[1 . . . i − 1] ∥ ⋆ ∥ f;
16                 F′ ← F′ ∪ {f′}
17             end
18             return F′;
19         else
               /* Otherwise, p[i] is '∗'.                             */
               /* We iterate on the number of letters this ∗
                  represents                                          */
20             F′ ← ∅;
21             for j ∈ [0, r − i + 1] do
22                 F ← WildcardQueryRec(p[i + 1 . . . ], r − i − j + 1);
23                 for f ∈ F do
24                     Let f′ = p[1 . . . i − 1] ∥ ⋆ (j's ⋆) ∥ f;
25                     F′ ← F′ ∪ {f′}
26                 end
27             end
28             return F′;
29         end
30     end
```

Chapter 7

Deterministic Reservation and its Application on STMs

This chapter presents LiTM, a lightweight software transaction memory. By supporting general transactions, LiTM extends a design paradigm of distributed graph algorithms by Blelloch et al. [27] to a full-fledged concurrency control algorithm. We call this concurrency control algorithm the *Deterministic Reservation* algorithm. We include evaluations on six graph applications as a micro-benchmark and comparisons with the state-of-the-art Software Transactional Memory (STM), Galois, to demonstrate the power of deterministic reservation. Chapter 8 will explain how deterministic reservation speeds up Litmus by massively batching non-conflicting transactions while simultaneously producing and proving the wrapped transaction circuit. Later, in Chapter 10, we will show that using deterministic reservation improves the throughput of Litmus by orders of magnitude.

7.1 Introduction

A *transaction* is a sequence of operations that must succeed or fail as a whole. Transactions offer a powerful abstraction for parallel programming as they enable programmers to wrap sequences of operations in regions that execute atomically, and not have to worry about concurrency issues. Software and hardware transactional memory

techniques are an active topic of research, where the goal is to provide programmability, while maintaining good performance and scalability. Transactions are also used in database management systems, where they bring useful and easy-to-understand ACID properties (Atomicity, Consistency, Isolation, and Durability).

One disadvantage of transactions, however, is the nature of non-determinism — even the strongest isolation level, serializability, allows transactions to be arbitrarily interleaved. This makes it challenging to reason about the correctness of the parallel program, as well as making debugging difficult.

Deterministic parallelism [30, 146, 147] is a promising technique that can largely resolve this issue. A parallel program is considered *deterministic* if each execution produces identical results, regardless of platform- or execution-specific features such as the number of processors being used or the scheduling of threads. Determinism makes programming significantly easier since a programmer only needs to consider a single execution order instead of all possible interleavings of parallel code. Debuggability is also improved since a bug can be reproduced by simply re-executing the program, effectively preventing the notorious Heisenbug [74] issue where a bug manifests itself differently in multiple executions of the same program.

Supporting determinism in transaction processing has been an active research area. Previous work has proposed to use determinism to improve debuggability of STM systems [153, 113, 122]. These previous approaches, however, either suffer from scalability bottlenecks [153, 122] or incur performance overhead in conflict resolution [113]. For example, in Galois, the state-of-the-art deterministic STM, performance is compromised by as much as 6× when determinism is turned on [113]. It is not surprising that a deterministic program performs worse than a non-deterministic one due to having a more constrained execution. However, the dramatic performance overhead can deter programmers from writing and using deterministic programs. Therefore, improving the performance of deterministic transaction processing is crucial for its widespread adoption.

In this chapter, we design LiTM, a deterministic STM system for multicores.[1]

[1]We have open-sourced the framework at `https://github.com/yuxiamit/LiTM`.

LiTM follows the *deterministic reservation* paradigm proposed by Blelloch et al. [28] but provides a more convenient and natural interface for general transactions to make it a full-fledged concurrency control algorithm. In contrast to [28], LiTM hides the algorithm completely from the programmer, who only needs to provide the content of the transactions instead of analyzing the algorithms and supplying the corresponding functions to handle the data conflicts manually. LiTM takes as input a set of transactions, automatically groups them into batches, analyzes their access patterns, detects conflicts, applies changes, and re-executes aborted transactions in a way that is transparent to the programmer.

Our evaluation on a 40-core machine demonstrates that LiTM can achieve up to 9.4× performance improvement over the sequential versions of the algorithms. Compared to the state-of-the-art deterministic STM framework Galois [113], LiTM achieves up to 5.8× speedup. Compared to hand-optimized parallel baselines [133], which require significantly more effort to program, LiTM incurs a modest overhead of at most 3×.

This chapter makes the following contributions:

- We design the LiTM deterministic STM system, which provides a simple yet powerful abstraction for programmers to write highly efficient parallel programs, making the paradigm by Blelloch et al. a full-fledged concurrency control algorithm.

- We implement six applications in LiTM: maximal independent set, maximal matching, spanning forest, PageRank, random permutation, and list contraction.

- We compare LiTM's performance against hand-optimized serial and parallel baselines, including Galois, a state-of-the-art deterministic parallel programming framework. We show that LiTM can outperform Galois by up to 5.8× and incurs modest overhead over the hand-optimized baselines.

The rest of the chapter is organized as follows. Section 7.2 discusses the background and motivation for LiTM. Section 7.3 presents the protocol of LiTM. In Sec-

tion 7.4, we evaluate LiTM on six applications and compare them against existing implementations.

7.2 Background

In this section, we discuss *deterministic reservation*, the technique that motivates LiTM. We also discuss the limitations of the protocol, which LiTM seeks to resolve.

7.2.1 Deterministic Reservation

Deterministic reservation [28] is a framework to write internally deterministic parallel programs comprised of multiple iterates. It guarantees that the program outcome is identical even when the iterates are executed in different orders. The key technique behind the framework is a way to deterministically resolve conflicts based on the predefined priorities of iterates, while allowing them to be executed in parallel. If there are any conflicts between iterates, the higher priority iterate takes precedence. This is achieved by having each iterate *reserve* its accessed data elements by writing its own priority into a reservation array *only when its priority is higher* than the current iterate reserving on the same data. Regardless of the order in which a data element is reserved, the iterate with the highest priority will reserve it in the end. Random numbers used in the programs must be deterministic across multiple runs [100].

For efficiency, the deterministic reservation framework orders the iterates by their priorities and processes them in multiple batches. In each batch, the framework first runs a *reserve*() function for each iterate in the batch in parallel. It reserves all of the data elements that the iterate accesses. Then, in parallel the framework runs a *commit*() function for each iterate to check whether it has successfully reserved all elements, and if so, applies the iterate's changes to shared memory, and otherwise, moves the iterate to the next batch and re-processes it later. Inside these two functions, data conflicts are handled manually by the programmer. The algorithm continues processing iterates in batches until no more iterates remain. It has been shown that deterministic reservation can achieve very competitive performance com-

Algorithm 10: MIS using LiTM.

```
1  enum {InMIS, NotInMIS} States;
2  DVector Flags = {NotInMIS, NotInMIS, ..., NotInMIS} # size n
3  Function T.runTxn()
4      for ngh in T.vertex.neighbors do
5          if Flags[ngh] == InMIS then
6              Flags[T.vertex] = NotInMIS
7              return
8      Flags[T.vertex] = InMIS
```

Algorithm 11: MIS using deterministic reservation.

```
1  enum {Undecided, InMIS, NotInMIS} States
2  char Flags[] = {Undecided, Undecided, ..., Undecided} # size n
3  int reservation[] = {∞, ∞, ..., ∞} # size n
4  Function reserve(i)
5      for ngh in vertices[i].neighbors do
6          if Flags[ngh] == InMIS then
7              Flags[i] = NotInMIS
8              return false # bypass the commit phase
9      # writeMin(&x, i) atomically sets x to min(x, i)
10     writeMin(&reservation[i], i)
11     return true
12 Function commit(i) # returns whether the iterate commits
13     for ngh in vertices[i].neighbors do
14         if (Flags[ngh] == Undecided and reservation[ngh] ¡ i)
             or Flags[ngh] == InMIS then
15             reservation[i] = ∞
16             return false
17     Flags[i] = InMIS
18     return true
```

pared to other approaches [28].

7.2.2 Example on Maximal Independent Set

We use maximal independent set (MIS) as an example application to demonstrate how deterministic reservation works and also to point out its limitations.

In a graph, an *independent set* is a set of vertices with no edge connecting any pair of them. A *maximal independent set* is an independent set such that if any other vertex joins the set, independence is violated.

Algorithm 10 shows how MIS can be implemented using transactions (this is also

the code to implement MIS in LiTM). Initially, all vertices are assigned to not be in the MIS using the *NotInMIS* flag (line 2). For each vertex, the *runTxn* function is executed. If any neighbor of the vertex is already in the MIS (lines 4-5), by definition, the current vertex cannot be selected in the set, and hence the transaction returns leaving the vertex out of the MIS (line 6–7). Otherwise, the current vertex is added to the MIS on line 8 with the *InMIS* flag.

To implement this application in the deterministic reservation framework, a programmer writes two functions, *reserve* and *commit*, as shown in Algorithm 11.[2] Both functions take the iterate number i as input. The iterate uses i as its priority, with a smaller value being higher priority. Initially, all vertices are given an undecided status (line 2), and the memory locations for performing reservations are initialized to ∞ (line 3). In the reserve function, if any neighbor of the vertex of interest has been selected, the current vertex is marked as not in the MIS (*NotInMIS*) and the function returns and tells the framework to bypass the *commit* phase (lines 5–8). Otherwise, the current vertex *might* be in the set and is therefore reserved using the *writeMin* function, which atomically updates *reservation*[i] with i if its previous value is greater than i (line 10).

In the commit function, if any neighbor of the vertex is in the MIS (*InMIS*), or undecided (*Undecided*) but reserved by a higher-priority transaction, the reservation is cleared and the current transaction has to abort (i.e., return false) in this batch (lines 13–16). Otherwise, the vertex is marked as *InMIS* and the transaction commits (lines 17–18).

LiTM inherits the same spirit of processing iterates in two phases. However, LiTM does not require the programmer to write the two functions manually. Instead, the programmer can provide the sequential logic of the code as a transaction (i.e., like in Algorithm 10), and LiTM automatically coordinates the transactions. The programmer does not need to deal with concurrency issues in LiTM. The details of the LiTM protocol will be discussed in Section 7.3.

[2]We note that there is a faster MIS implementation using an algorithm-specific optimization in [28]. We do not present it here because the optimization is not general across applications. Later we will use it as the hand-optimized baseline in Section 7.4.

7.3 The LiTM Protocol

The goal of LiTM is to implement the deterministic reservation framework without requiring a programmer to provide the *reserve* and *commit* functions. Instead, the programmer writes a transaction, and LiTM automatically extracts the codes for reserve and commit functions based on the provided transaction. We discuss the API of LiTM in Section 7.3.1, the data structures used in LiTM in Section 7.3.2, the detailed protocol in Sections 7.3.3 to 7.3.6, and design parameter selection in Section 7.3.7.

7.3.1 API of LiTM

In LiTM, the programmer writes the logic of transactions. The shared data structures are declared as special types, so that the system can capture read and write operations to them. Specifically, LiTM supports a special *DVector* type as a shared array. In Algorithm 10, for example, the *Flags* structure has the type of *DVector* with its base type as chars. Section 7.3.4 shows how LiTM handles reads and writes to a *DVector*.

7.3.2 Data Structures in LiTM

The following data structures are required in LiTM to perform the reserve and commit functions.

Lock Table (LT) is used to perform reservations. It is implemented as a hash table, where each entry in LT contains the priority (which is of integer type) of a transaction that reserves the data element that maps to the LT entry. The number of entries in LT does not have to equal the total number of data elements. If the LT is too small, hash collisions may occur, leading to more transaction aborts and performance degradation, but the execution is still functionally correct.

Read and Write Sets (RS and WS). RS (WS) of a transaction T keeps the pointers (pointers and values) of each data element that T reads (writes). These two sets are updated during the reserve phase and later used in the commit phase to check whether T has successfully reserved all elements it has touched so that it is

Algorithm 12: The overall logic of LiTM — The reserve, commit, and cleanup phases are highlighted in different colors. **foreach** loops are executed by all threads in parallel in any order.

```
1  # txns: list of transactions to be executed.
2  Function run(txns)
3      batch ← txns.pop(batch_size)
4      while not batch.empty() do
5          next_batch ← []
6          foreach T in batch do
7              T.runTxn()
8          foreach T in batch where T.WS ≠ ∅ do
9              can_commit ← run_commit_phase(T)
10             if not can_commit then
11                 next_batch.append(T)
12         batch ← next_batch + txns.pop(batch_size - next_batch.size())
```

able to commit. For a committed transaction, the values in the WS are copied to the *DVector*s.

7.3.3 Overall logic of LiTM

Algorithm 12 shows the overall logic of LiTM for executing a set of ordered transactions. The execution contains multiple batches, each broken down into three separate phases. The *reserve* and *commit* phases correspond to the two functions in the deterministic reservation algorithm. The *cleanup* phase tidies up relevant data structures between different batches.

LiTM begins by taking a prefix of *batch_size* transactions as a batch (line 3). If the number of remaining transactions is less than *batch_size*, the batch will contain all remaining transactions. The following three phases are performed as long as there are still transactions remaining (line 4).

During the *reserve phase*, the *runTxn* function is executed for each transaction in the batch. This function is the transaction logic defined by the programmer (see Algorithm 10 for an example). *runTxn* performs reservations and updates the RS and WS. More details will be in Section 7.3.4.

The *commit phase* begins after all transactions in the batch have finished the

Algorithm 13: The reserve phase — Read and write operation to the ith element in a *DVector*.

```
1  Function DVector.read(T, i)
2      addr ← &DVector._values[i]
3      T.RS.insert(addr)
4      if addr in T.WS then
5          return T.WS[addr]
6      return DVector._values[i]
7  Function DVector.write(T, i, val)
8      addr ← &DVector._values[i]
9      T.WS.insert(addr, val)
10     # begin atomic section
11     if T.priority < LT[ hash(addr)] then
12         # Smaller numbers mean higher priorities
13         LT[ hash(addr) ] ← T.priority
14     # end atomic section
```

reserve phase. If a transaction T from the batch is read-only, i.e., having an empty WS, T can simply commit as if it happens logically at the beginning of the batch. Skipping the commit phase for read-only transactions is an optimization that can reduce the RS storage as well as improve performance, while retaining determinism. For a read-write transaction T, the *run_commit_phase*(T) is executed, which checks whether T is able to commit or not. More detailed discussion will be presented in Section 7.3.5. If T is not able to commit in the current batch, it is appended to *next_batch* in a deterministic way and will be reprocessed in the next batch.

Finally, the *cleanup phase* assembles the next batch of transactions by taking the remainder of the current batch, and another prefix from the input transactions.

7.3.4 Reserve Phase

During the reserve phase, the *runTxn* function of each transaction is executed, wherein the *DVectors* are accessed. For each read or write operation to a *DVector* element, the logic in Algorithm 13 is executed. For a read by transaction T, the address of the accessed element is inserted into T's RS (line 3). If it is in T's WS, the value in the WS is returned (line 5), otherwise, the value stored in the *DVector* is returned (line 6).

Algorithm 14: The commit phase – implementation of the *run_commit_phase(T)* function.

```
1  Function run_commit_phase(T)
2      # Check whether T successfully reserved all elements
3      for addr in T.RS ∪ T.WS do
4          if T.priority < LT[ hash(addr) ] then
5              return false
6      # Copy elements in the WS to DVectors
7      for (addr, val) in T.WS do
8          memcpy( addr, &val, sizeof(val) )
9      return true
```

For a write by transaction T, both the address and the new value of the element are inserted in the T's WS (line 9). Then, the write is reserved in the LT by atomically replacing the corresponding entry with T's priority if the existing priority is lower (the value is larger) (line 10–14). Updating the value in the shared data structure is delayed until the commit phase.

Both the *read* and *write* functions in Algorithm 13 are deterministic. Each read always returns the value at the start of the batch (since there are no modifications to shared data structures yet). After the batch, the state of the LT is always the same, regardless of the order in which the transactions are scheduled — an element is always reserved by the transaction with the highest priority that writes to the element.

7.3.5 Commit Phase

The *run_commit_phase(T)* function is the main logic in the commit phase, which is defined in Algorithm 14. LiTM first checks whether a transaction T has successfully reserved all of the elements that it accesses during the reserve phase by comparing T's priority with the priority stored in the LT. If any element is reserved by another transaction with a higher priority, T is not able to commit and has to be moved to the next batch. If all elements are reserved by T, T's WS is copied back to the shared *DVector*s. The execution in the commit phase is deterministic because the states in the LT are deterministic and only the highest-priority transaction can reserve and hence possibly update a particular element.

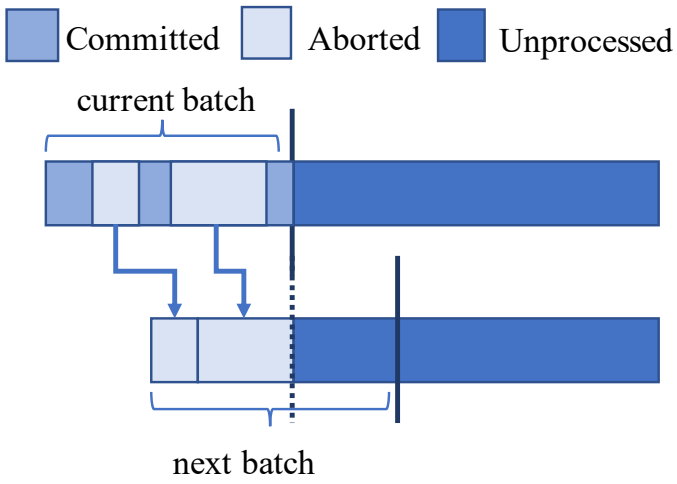


Figure 7-1: **The deterministic execution of batches of transactions.**

7.3.6 Cleanup Phase

After all transactions within a batch finish the reserve and commit phases, transactions that cannot commit due to conflicts are moved to the next batch. The job of the *cleanup phase* is to pack these transactions and obtain more transactions from the input so that the next batch has the desired size. Figure 7-1 depicts the idea of the cleanup phase. The additional transactions are always obtained by taking a prefix of the unprocessed transactions. Given that the current batch is deterministically executed, the next batch is also deterministic.

7.3.7 Parameter Selection

The parameter space of LiTM mainly consists of the *batch size* and the *lock table size*. A small batch size corresponds to less exploitable parallelism during the reserve and commit phases, and more overhead during the cleanup phase. A large batch size can better amortize the cost of cleaning up. If the batch size is too large, however, the maintained local RS and WS can consume significant space. Large batches may also increase the conflict rate of transactions within a batch, increasing the abort rate and hurting performance.

The lock table size has a similar tradeoff. A small lock table leads to more hash collisions and increased number of aborts. A large lock table that is too large may

not fit in cache leading more expensive data accesses from main memory.

Fortunately, as we show in Section 7.4.4, LiTM can achieve very close to the best performance for a large range of batch sizes and lock table sizes. Therefore, the programmer does not need to devote significant effort to tuning these parameters.

7.4 Micro-Benchmark Evaluation

In this section we compare the performance of LiTM with several baselines using the *Problem Based Benchmark Suite* (PBBS) [133, 134]. The experiments are performed on a single machine running Ubuntu 14.04 LTS. The machine has four Intel(R) Xeon(R) E7-4850 CPUs, with a total of 40 cores, and 128 GB of main memory. We turned off hyper-threading to get reliable measurements. When reporting the runtime of experiments, we exclude the time for loading the input files.

7.4.1 Workloads

We used six benchmarks from PBBS. The benchmark names, input files, and input sizes are summarized in Table 7.1. The three input graphs are generated as follows:

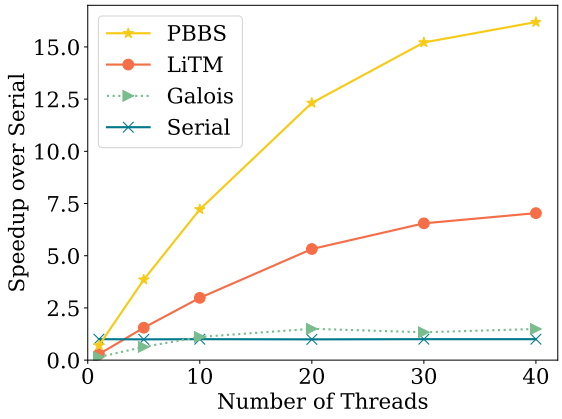
Random Local Graphs (random): We generate random local graphs where edges are sampled independently with probability inversely proportional to the difference between vertex IDs. The average node degree of the graph is set to 5.

Recursive-Matrix Graphs (rMat): The Recursive-Matrix graphs [47] are synthetic graphs that model a power-law degree distribution. Again, we set the average degree to 5.

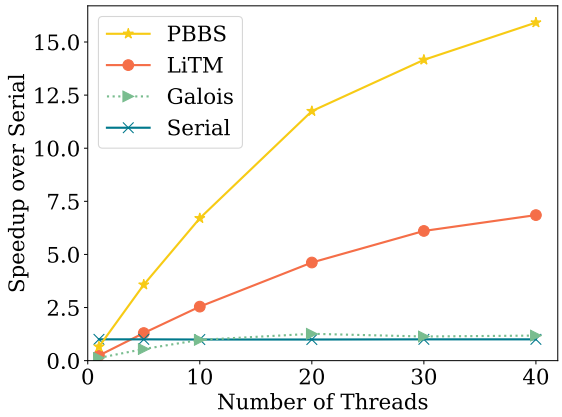
3D Grid Graphs (3DGrid): 3D Grid Graphs are graphs where n vertices are placed evenly on a 3-dimensional cube with side-length $n^{1/3}$, and each vertex has edges to its six closest neighbors (two in each dimension).

Benchmark	Input Files	Input Size
Maximal Independent Set	random	$n = 10^8$, $m = 5 \times 10^8$
	rMat	$n = 10^8$, $m = 5 \times 10^8$
	3DGrid	$n = 10^8$, $m = 3 \times 10^8$
Maximal Matching	random	$n = 10^8$, $m = 5 \times 10^8$
	rMat	$n = 10^8$, $m = 5 \times 10^8$
	3DGrid	$n = 10^8$, $m = 3 \times 10^8$
Spanning Forest	random	$n = 10^8$, $m = 5 \times 10^8$
	rMat	$n = 10^8$, $m = 5 \times 10^8$
	3DGrid	$n = 10^8$, $m = 3 \times 10^8$
Pagerank	random	$n = 10^8$, $m = 5 \times 10^8$
Random Permutation	random	$n = 10^9$
List Contraction	random	$n = 10^9$

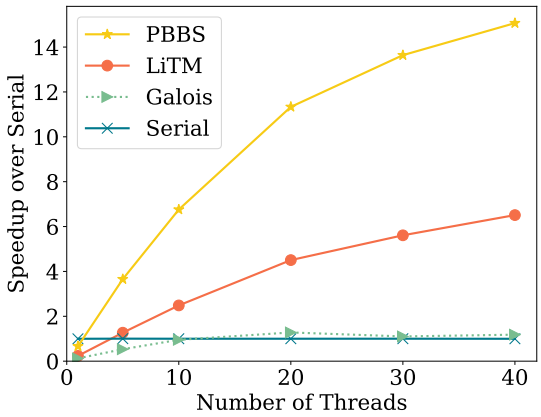
Table 7.1: **Benchmarks and Inputs.** For the graph inputs, n is the number of vertices, and m is the number of edges.



(a) Random

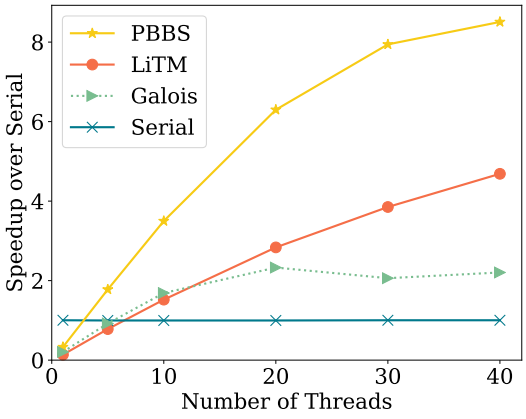


(b) rMat

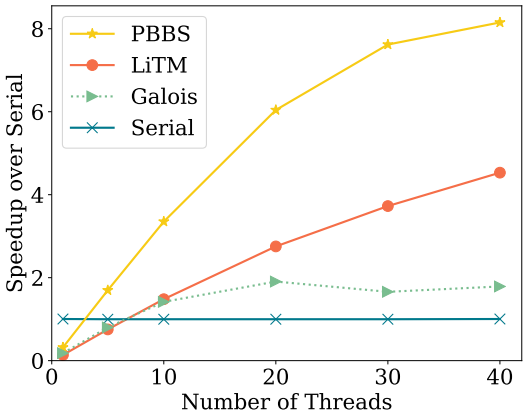


(c) 3DGrid

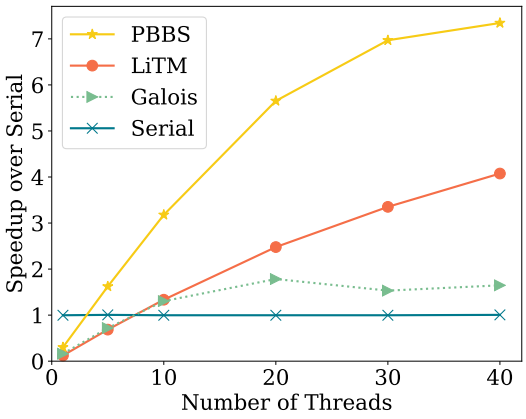
Figure 7-2: **Performance comparison on *maximal independent set*** — Speedup normalized to the serial baseline with varying thread count.



(a) Random

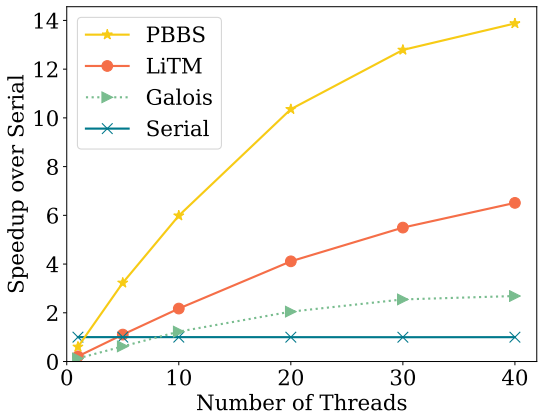


(b) rMat

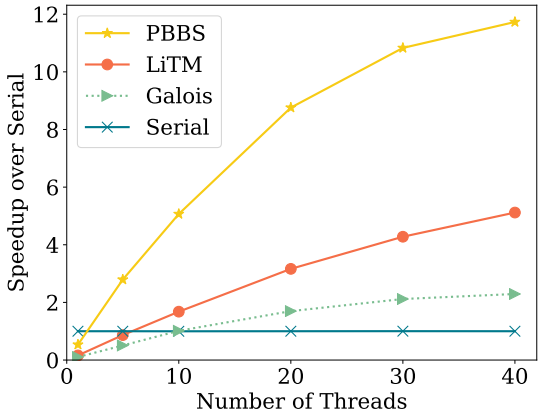


(c) 3DGrid

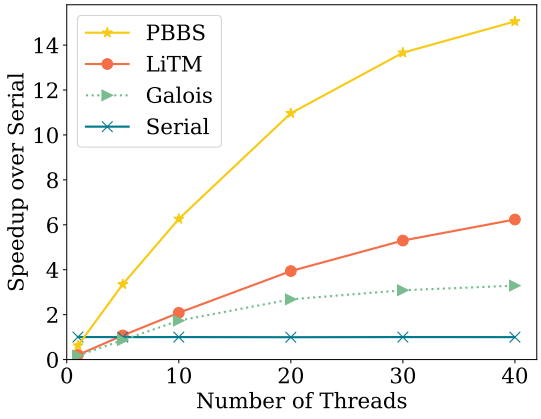
Figure 7-3: **Performance comparison on *maximal matching*** — Speedup normalized to the serial baseline with varying thread count.



(a) Random

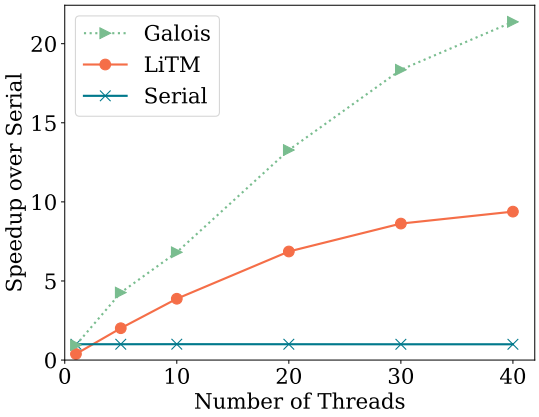


(b) rMat

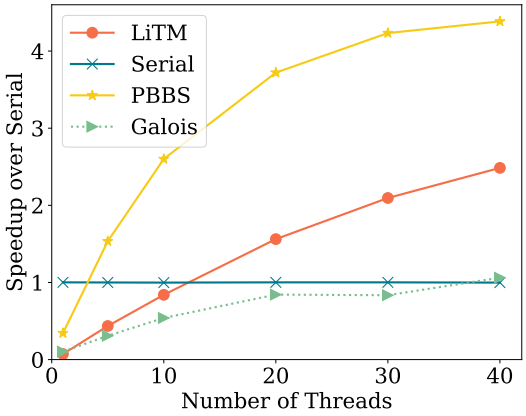


(c) 3DGrid

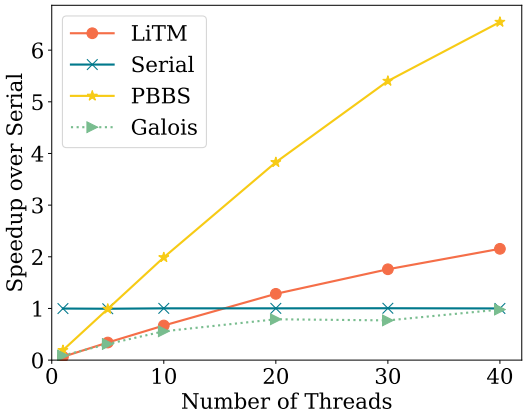
Figure 7-4: **Performance comparison on *spanning forest*** — Speedup normalized to the serial baseline with varying thread count.



(a) *PageRank*



(b) *random permutation*



(c) *list contraction*

Figure 7-5: **Performance comparison on *PageRank*, *random permutation*, and *list contraction*.** — Speedup normalized to the serial baseline with varying thread count.

7.4.2 Baseline Systems

PBBS Implementation. This is a suite of hand-optimized codes released with the Problem Based Benchmark Suite by Shun et al. [133, 134]. The PBBS implementations that we compare with are all deterministic. In contrast to the algorithm example we discussed in Algorithm 11, the PBBS implementations that we used for evaluation include application-specific optimizations. They require the programmer to be fully aware of the problem details as well as be skilled in parallel programming. In contrast, LiTM is easier to use due to its transactional interface. By being a general framework, we do not expect LiTM to outperform the PBBS implementations. However, later we will show that LiTM exhibits only a modest slowdown compared to the hand-optimized codes.

LiTM Implementation. Our implementation of LiTM is designed for multicore machines. The underlying parallel for-loops can be compiled with either Cilk Plus [99] or OpenMP [56] (we use Cilk Plus for our experiments). The code base is compiled with the GNU Compiler Collection version 5.4.0 with `-O2` flag. Unless otherwise stated, the lock table size is the number of vertices for the graph algorithms and number of elements for the others, and the batch size is 500,000. We will evaluate the sensitivity to these two parameters in Section 7.4.4.

Galois Implementation. The Galois implementation is adapted from the default applications shipped with Galois version 2.2.1. The transactional interface in Galois is more limited than that of LiTM. Galois requires transactions to be *cautious*, namely, the transactions must read everything it needs before writes anything. This limitation makes Galois less expressive than LiTM, as we will show later in Section 7.4.3. Galois offers special interfaces which take advantage of the programmer's hints. For example, if the programmer is aware that the transactions do not have any conflicts, he or she can use a special interface to turn off the conflict detection in order to gain extra performance. Such optimizations could also be applied to LiTM. To perform a fair comparison, we use the versions with the general Galois deterministic interface unless otherwise specified.

Benchmark	LiTM	PBBS	Galois
Maximal Independent Set	15	23	48
Maximal Matching	14	26	54
Spanning Forest	34	56	106
Pagerank	73	–	92
Random Permutation	20	27	32
List Contraction	13	22	32

Table 7.2: **Code Length.** — Number of lines of code of each benchmark in each of the frameworks.

Serial Execution. We compare against a serial implementation of each benchmark without any framework overheads.

Table 7.2 shows a summary of the code lengths of the implementations listed above. We only count the lines in the transaction payload, excluding comments and blank lines. These line numbers show that the LiTM programming interface is relatively simple.

7.4.3 Main Results

Figures 7-2–7-5 show the performance of LiTM, PBBS, Galois, and the serial code on all of our benchmarks. We report speedup numbers normalized to the serial baseline. For each workload, we increase the number of threads from 1 to 40. From the figures, we can see that LiTM scales well for all the benchmarks studied. For *maximal independent set*, LiTM achieves 24.7–28.5× speedup on 40 threads compared to its single-threaded performance on the random graph (Figure 7-2(a)), recursive matrix graph (Figure 7-2(b)), and 3D grid graph (Figure 7-2(c)). Compared to the serial baseline, LiTM achieves a 6.5–7.0× speedup on 40 threads. LiTM is 2.3× slower than the hand-optimized PBBS baseline on all 3 graphs.

For *maximal independent set*, deterministic Galois does not scale well when the number of threads is larger than 20 due to *non-uniform memory access* (NUMA) issues. This scalability result is consistent with Galois' performance reported in [113].

The results are similar for *maximal matching* and *spanning forest*. On the three

input graphs, LiTM outperforms the serial baseline by 4.1-4.7× and 5.1-6.5× for *maximal matching* and *spanning forest*, respectively. Note that the Galois performance reported in Figure 7-4 corresponds to the *blocked-asynchronous* version[3] shipped with Galois release 2.2.1. The algorithm is slightly different from the one used in PBBS, where the Union-Find Set (UNS) is explicitly implemented on the shared states. The Galois system provides a special node type *UnionFindNode* to support a hand-optimized lock-free UNS. From Figure 7-4 we can see that LiTM outperforms Galois by up to 1.9–2.4× at 40 threads. The fact that we did not implement the exact same algorithm of *spanning forest* in Galois as in PBBS is due to a limitation of Galois' programming model, where a transaction has to perform all the reads it needs before it writes to anything. The PBBS implementation uses a typical union-find set that performs path compression at the time of finding the root, resulting in reads and writes being interleaved.

We also tested LiTM on three additional workloads: PageRank, random permutation, and list contraction. We only show their performance on random graphs in Figure 7-5. We can see that LiTM scales well for all three workloads, achieving 24.4×, 34.7×, and 33.6× speedup compared to a single thread for PageRank, random permutation, and list contraction, respectively. Compared to the serial baseline, LiTM achieves a speedup of 9.4×, 2.5×, and 2.2×, respectively. LiTM achieves 2.2–2.3× faster performance over Galois'.[4] For PageRank, Galois did not provide a version using its general framework, but only a version with the application-specific optimization of turning off conflict detection. As shown in Figure 7-5(a), the optimized Galois version performs extremely well in our benchmark test for PageRank, achieving over 21.6× speedup against the serial version. LiTM is able to achieve about half of the performance using a more general framework (with conflict detection).

Finally, we observe that the performance of LiTM, PBBS, and Galois on a single

[3]Galois 2.2.1 comes with 3 versions of *spanning forest*. (1) 'Demo' is a demonstrative baseline that only works on strongly connected graphs. (2) 'Asynchronous' and (3) 'Blocked-Asynchronous' are two Kruskal-like algorithms, and the latter is claimed to be an improved version over the former.

[4]For Galois, we excluded the cost in computing the swapping destinations in the random permutation workload because this process is not easily decoupled from the input processing, and so the performance of Galois shown in Figure 7-5(b) is an upper bound.

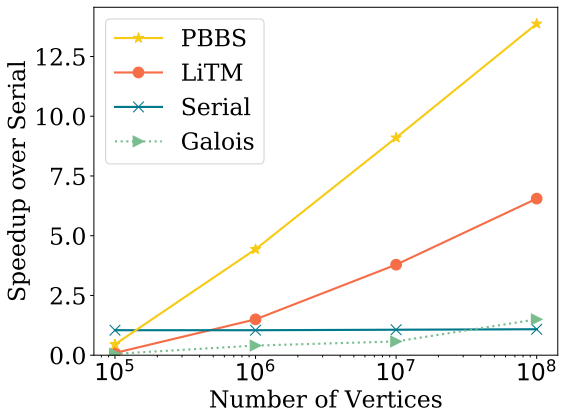


Figure 7-6: **Input Size Sweep** — The speedup over the serial baseline improves as the input size increases.

thread is worse than the serial baseline due to overheads in the frameworks and in parallel scheduling.

7.4.4 Sensitivity Study

In this subsection, we perform a sensitivity study on LiTM with respect to input size, batch size, and lock table size, to better understand its performance. We use *maximal independent set* as the workload and random local graphs as the inputs for these studies. Three metrics are reported for these results: (1) speedup over the serial baseline; (2) abort rate, which is defined as the ratio of the total number of aborts over the total number of transactions (note that if a transaction is aborted multiple times, it will be counted multiple times in calculating the abort rate); and (3) execution time breakdown of the different phases in LiTM.

Input Size. Figure 7-6 shows the results of *maximal independent set* on varying sizes of the input graph. We vary the number of vertices from 10^5 to 10^8 on the x axis (in log-scale) and set the number of edges to 5 times the number of vertices. We choose the batch size to be 200,000 and the lock table size to be the number of vertices. Unless otherwise specified, they are the default parameters we use in the following experiments. The larger the input is, the better the speedup we get from LiTM over the serial baseline. This is due to two factors: (1) larger inputs lead to

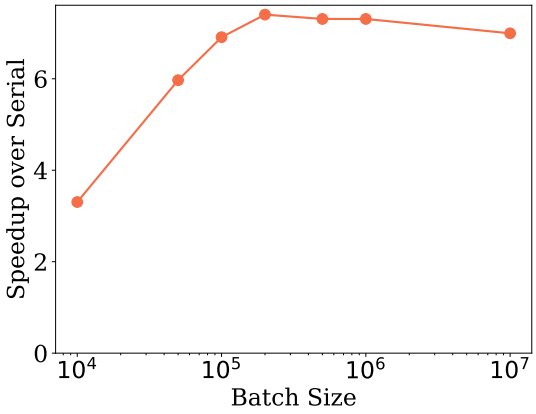
more parallelism in LiTM, and thus higher speedup, and (2) larger graphs make the benchmarks more memory bound, hiding the overhead of the extra computation in LiTM.

Batch Size. The batch size is critical to the performance in LiTM. The optimal batch size is determined by the intrinsic properties of the problem and the number of threads available. If the batch size is too small, there is not much parallelism that we can exploit within a batch, and hence adding more threads will not help. If the batch size is too large, the system will suffer from (1) higher abort rates due to conflicts among transactions within the same batch, and (2) higher overhead from managing large auxiliary data for the transactions (e.g., local read and write sets of each transaction).

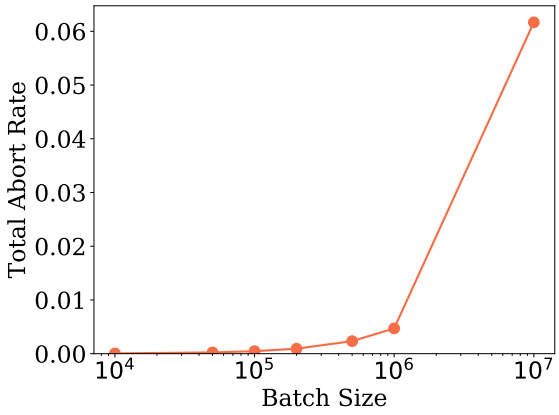
Figure 7-7(a) shows how the performance change as we increase the batch size from 10^4 to 10^7. We see that the speedup increases consistently when the batch size goes up to 2×10^5. When the batch size increases beyond that, the speedup starts to drop slowly. Such decrease is caused by higher abort rate and system overhead due to larger auxiliary data structures. Figure 7-7(b) shows that the abort rate increases when the batch size increases. When the batch size is larger, a transaction is more likely to conflict with another. Figure 7-7(c) shows the normalized time breakdown of LiTM. With a larger batch size, the time spent in the cleanup phase is amortized and hence decreases, giving better overall performance. Although the performance varies with batch size, we see that we can achieve close to the best performance for a wide range of batch sizes (i.e., from 10^5–10^7), so this parameter does not require significant tuning.

Lock Table Size. The lock table size is important to the performance as we frequently access the lock table during execution. A smaller lock table will result in more false conflicts caused by hash table collisions. A larger lock table leads to more cache misses which may also hurt performance.

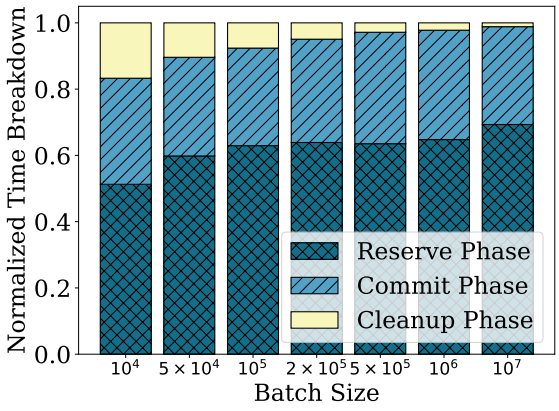
Figure 7-8 reports how sensitive the performance of LiTM is to the size of the lock table. The benchmark is executed with a batch size of 200,000. The x-axis is the ratio of the input graph size (number of vertices) to the lock table size (in



(a) Speedup over Serial

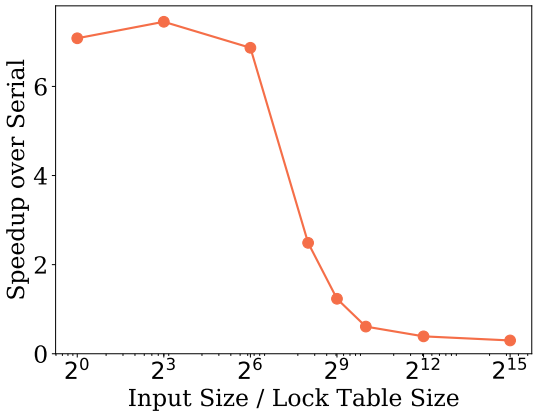


(b) Total Abort Rate

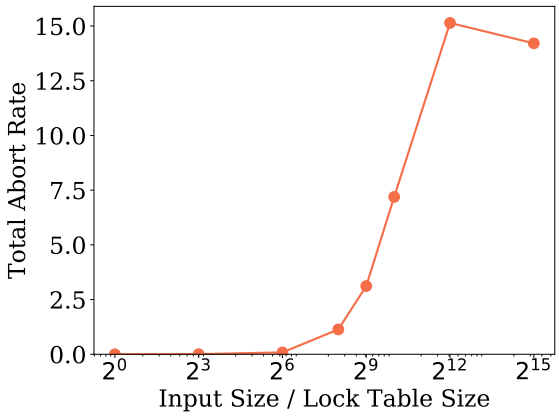


(c) Time Breakdown

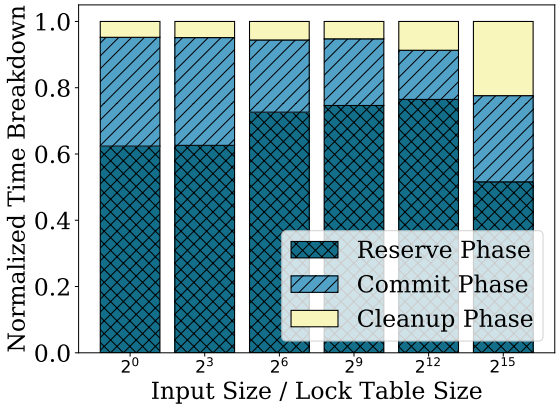
Figure 7-7: **Batch Size Sweep** — The speedup over the serial baseline, total abort rate, and time breakdown change as the batch size increases. The x-axis is in log scale.



(a) Speedup over Serial



(b) Total Abort Rate



(c) Time Breakdown

Figure 7-8: **Lock Table Size Sweep** — The speedup over the serial baseline, total abort rate, and time breakdown change as the lock table size increases. The x-axis is in log scale.

number of entries). A larger x-value corresponds to a smaller lock table. The false conflicts caused by a small lock table lead to more unnecessary aborts, which hurts performance. From Figure 7-8, we observe that the abort rate increases rapidly when the lock table size becomes small. Figure 7-8(c) shows that the time of the cleanup phase increases as the lock table size decreases because of more aborted transactions.

Chapter 8

A Multi-Threaded Design

In Chapter 6, we described a single-threaded baseline. This section presents 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 verifiable computation frameworks to work in parallel. Existing works on verifiable computation 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 verifiable computation framework efficiently outside of [144, 178].

8.1 Concurrency Control and Transaction Merging

We use the *deterministic reservation* protocol as the concurrency control algorithm for the normal DBMS part [27, 167]. It processes transactions by batches.

We first present a quick recap of the deterministic reservation algorithm in Section 7.3. For each batch, the concurrency control algorithm runs two phases in parallel. Algorithm 15 shows the pseudocode for the algorithm. It allocates a global array of reservation R and sets it to 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 immediately as there is a conflict. Otherwise, it applies the batch to the database. Finally, the function returns yes.

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 of calls to the memory integrity checker.* Exploiting the aggregatability of our authenticated dictionary 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 and produce the same circuit as that on the server thanks to the determinism property. Therefore, the server does not need to send the circuit to the client, and the client does not need to perform well-formedness check on the circuit.

- *(c) Flatten the circuit by reducing its depth.* The depth of the circuit is critical for the prover efficiency for some verifiable computation frameworks [72].

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 [142] that improves the prover efficiency.

To see why deterministic reservation improves the prover's performance, consider the three transactions $T1$, $T2$, and $T3$ in Figure 8-1. Both $T1$ and $T2$ read DB[A] and write DB[B]. $T3$ reads DB[C] and writes DB[D]. It is straightforward to see that $T1$ conflicts with $T2$. Given a pool of transactions, the deterministic reservation algorithm identifies a maximal non-conflicting batch of transactions from the transaction pool deterministically. In our example, this non-conflicting batch consists of $T1$ and $T3$, marked as Batch 1. The next non-conflicting batch has only one transaction, namely, $T2$.

We merge non-conflicting batches "horizontally" in the wrapped transaction. Intuitively, this brings two benefits. First, we can aggregate the proofs of multiple lookups together to form a single short proof. Second, we can update the digest after finishing all the transactions in the non-conflicting batch. We show an example of merging transactions in Figure 8-2. Given the splitting result in Figure 8-1, we chain the non-conflicting batches instead of single transactions. Specifically, we put $T1$ alongside $T3$ in the circuit. Before they start executing (in the circuit), we only perform a single integrity check on the aggregated lookup proof of DB[A] and DB[C]. After they finish executing, we only do a single-shot update of the digest, applying the changes of DB[B] and DB[D]. Then, we continue with the next non-conflicting batch, namely, $T2$ alone. Following the decision of the deterministic reservation concurrency control algorithm, we re-arrange the transactions into batches. Doing this might change the order of transactions. In this example, we run $T3$ before $T2$. This is acceptable since any sequential order leads to serializability.

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 Chapter 10, enabling batching yields a throughput gain of around 10×. Because the concurrency

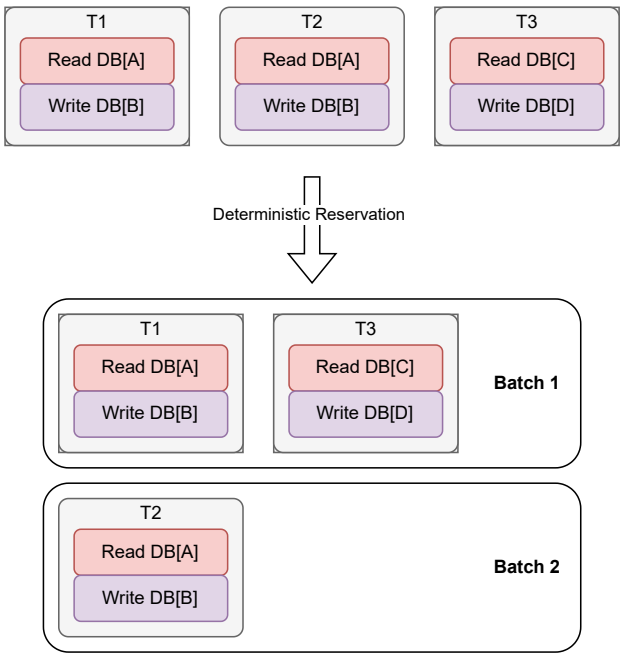


Figure 8-1: **Example Batching from Deterministic Reservation** — Deterministic Reservation produces maximal non-conflicting batches of transactions.

control 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 of TPC-C we evaluated). If the transactions' write set 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 or not the transactions in the same batch are non-conflicting 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 that the transactions access different places.

Although Litmus also supports non-deterministic concurrency control algorithms (c.f. Chapter 6), we justify our choice of deterministic 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

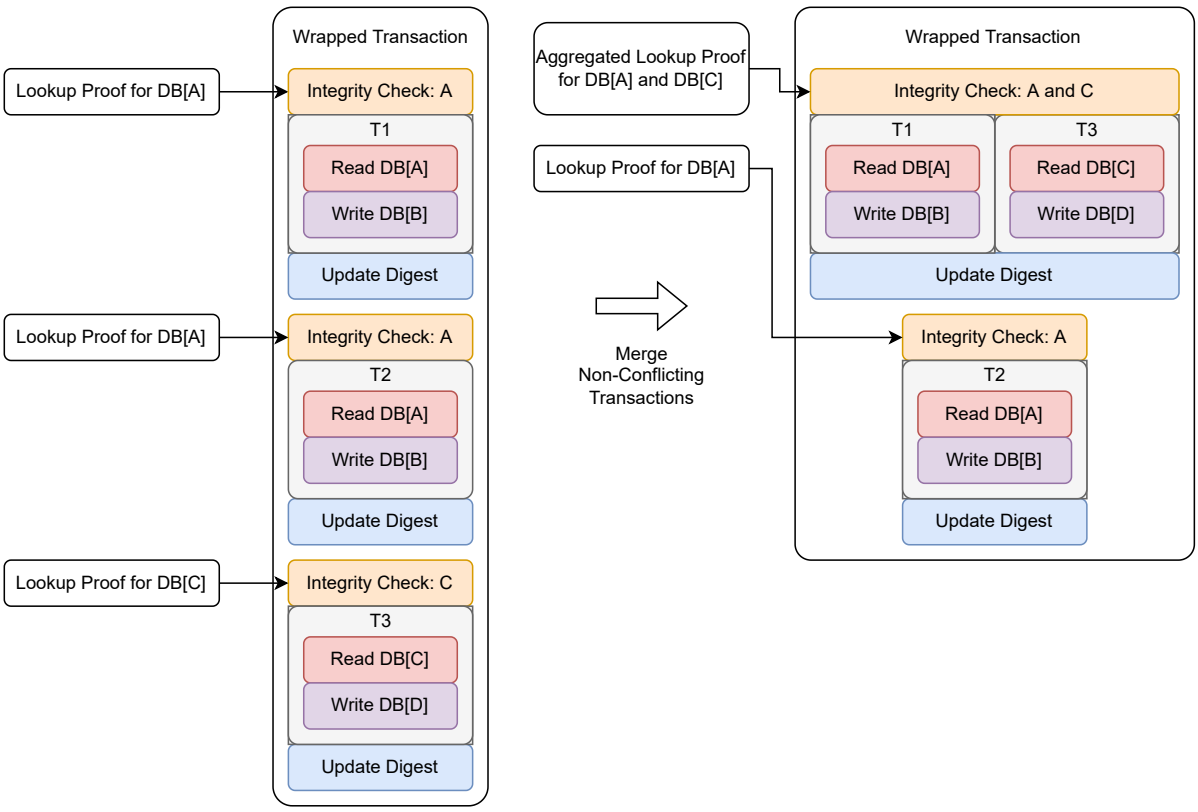


Figure 8-2: **Merging Non-Conflicting Transactions** — Given non-conflicting batches, we merge transactions in a batch horizontally.

matching. This adds to communication cost and increases latency. However, batching techniques are still feasible as long as the runtime traces can produce batches of non-conflicting transactions from the concurrency control algorithm's choices, which is not necessarily deterministic. For example, if every transaction only visits a single partition, we know these transactions will not conflict with each other. Therefore, we can organize them "horizontally" into a non-conflicting batch.

Figure 8-3 shows the chronological flow when we use deterministic reservation. Compared to the original chronological flow of Figure 4-4, the server no longer sends the circuit to the client, and the client no longer performs the well-formedness check. Because the concurrency control algorithm provides deterministic interleavings, the client can produce the circuit on its own. Therefore, it does not need to wait for the server to produce the circuit or perform the well-formedness check. While the server is processing the transactions, the client can efficiently generate the circuit and send it

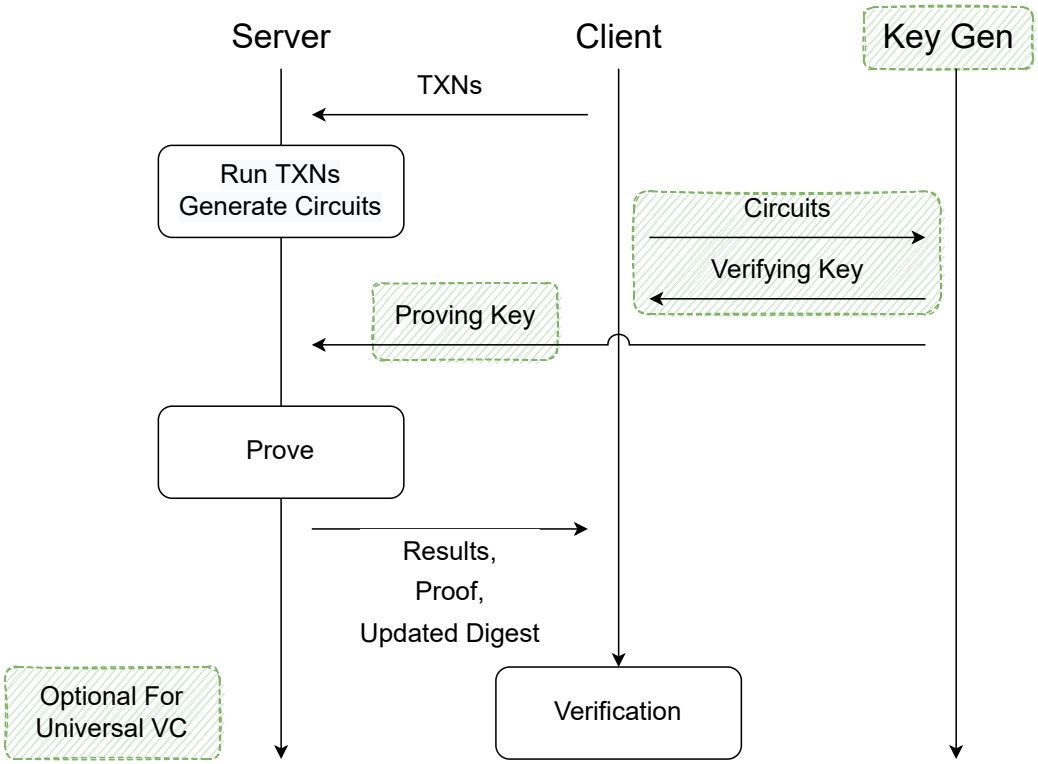


Figure 8-3: **Overview of Chronicle Steps with Deterministic Reservation** — Since the deterministic reservation concurrency control algorithm provides deterministic interleavings, the client can produce the circuit on its own.

to the key generator. Thus, we remove the time cost of key generation from the critical path. In the previous discussion in Section 4.2, the key generation process starts after the server sends the circuit and the client performs the well-formedness check. And the prover on the server needs the proving key from the key generator. Therefore, the key generation is on the critical path if the concurrency control algorithm is not deterministic.

8.2 Pipelining Provers

Following the determinism of the concurrency control algorithm, the transaction wrapper and the memory integrity provider are able to work on their own, and they do not have to wait for the traces from the normal database component. This not only enables the merging of non-conflicting transactions but also directly increases

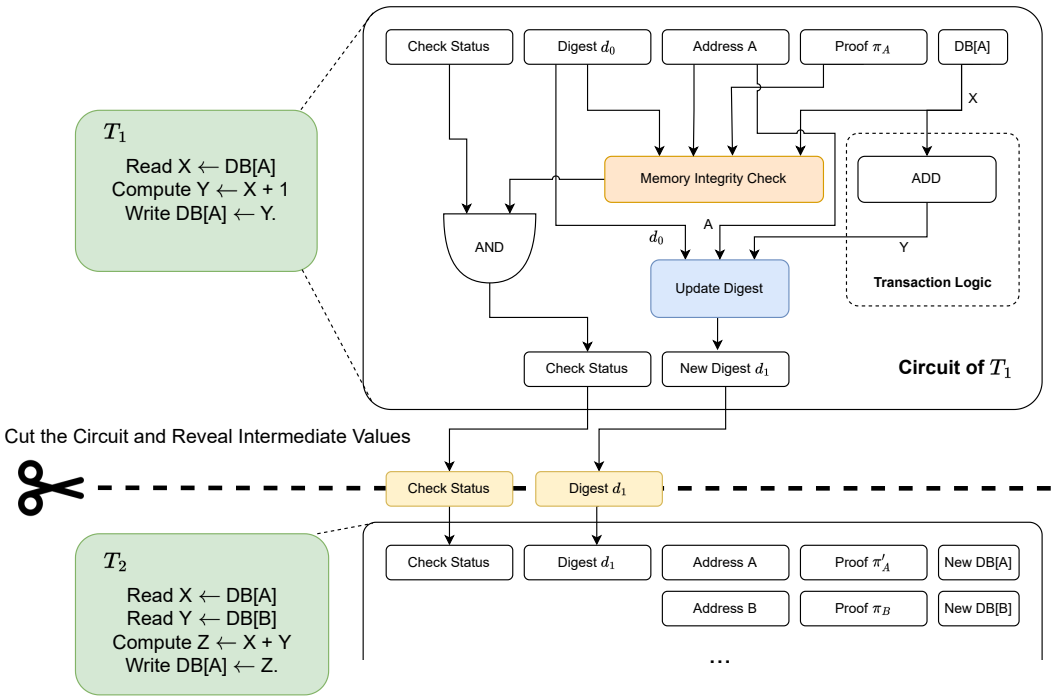


Figure 8-4: **Example Circuit Cutting** — We cut the wrapped transaction from Figure 4-2 into two pieces to pipeline and parallelize the proving process.

parallelism and reduces interaction between components inside the server.

We enable parallel proving without modifying the underlying verifiable computation framework. Namely, we *break* the whole wrapped transaction circuit into sequential pieces where each piece consists of multiple transaction batches, and we let a thread work on the proof of a single circuit piece. Figure 8-4 shows an example of circuit cutting. The wrapped transaction comes from the example in Figure 4-2. It consists of two transactions. We split it into two pieces and reveal the intermediate values to the client. The intermediate values, namely, the check status and the digest, now become the output values of the first circuit piece and the input values of the second circuit piece. Assuming the values of the connecting wires between pieces are consistent, proving both pieces are correctly evaluated is equivalent to proving the original wrapped transaction. Cutting the circuit incurs extra communication costs due to the intermediate values. However, in our case, this extra cost is linear in the number of cuts, and the constant factor is small since there are only two constant-sized values passed between the circuit pieces.

More generally, we have a general result for an arbitrary cut in the circuit.

Claim 1 (Circuit Splitting)**.** *Given a circuit C with its gate DAG $G = (V, E)$, any graph cut $c = \{e_i\}$ that splits the gates into V_1 and V_2 such that $c \subseteq V_1 \times V_2$, the statement $C_1(\boldsymbol{x}) = \boldsymbol{w} \wedge C_2(\boldsymbol{w}) = \boldsymbol{y}$ implies $C(\boldsymbol{x}) = \boldsymbol{y}$, where $\boldsymbol{w} = [\boldsymbol{w}^{(i)}]$ is a vector of values corresponding to $\{e_i\}$, and C_i is the circuit consisting of the gates V_i and wires $(V_i \times V_i) \cap E$ for $i \in \{1, 2\}$.*

Claim 2 (Soundness of Circuit Splitting)**.** *Given a verifiable computation framework with a soundness error bound E, splitting a circuit into k pieces and proving them individually has a soundness bound $k \cdot E$.*

However, we note that computationally sound verifiable computation frameworks only allow a polynomial number of cuts. Otherwise, the multiplier of the total soundness bound might reach the exponential level, resulting in a non-negligible soundness error.

Now, we talk about how to pipeline-prove the circuit pieces. As shown in Figure 8-5, 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 pieces and their proofs. If all the proofs generated by the prover threads are valid and the values passing through connecting circuit pieces 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 were successful. The cost of checking value consistency is minimal. Enabling multiple provers yields an extra gain of around 25×.

Algorithm 16 shows a simple example dispatcher. The dispatcher takes in a stream of non-conflicting batches, the number of batches n, and the number of prover threads t. At the beginning, it initializes a list L to be empty. Whenever a non-conflicting batch B is ready from the deterministic reservation algorithm, it appends B to the list L. If the number of batches in the list reaches $\lceil n/t \rceil$, it starts a prover to prove the circuit piece corresponding to the batches in L and clears the list L. This dispatcher

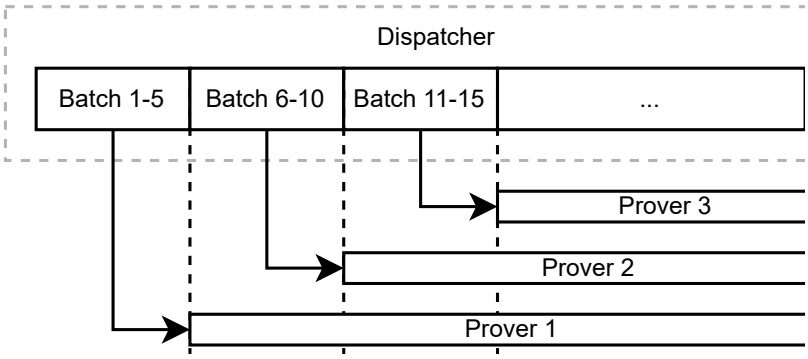


Figure 8-5: **Litmus Pipelining** — We start multiple provers with each producing on several batches.

is optimal if the workload is uniform. For a more general workload, this becomes an online load-balancing problem.

Algorithm 16: A Simple Dispatcher for Uniform Workload

Input: A stream of non-conflicting batches $B_1, B_2, \ldots$, the number of batches n, and the number of threads t.

```
1 Func Dispatch():
2     Initialize L ← [];
3     Whenever a batch B is ready:
4         Append B to L;
5         if |L| ≥ ⌈n/t⌉ then
6             Start a prover thread to prove L;
7             L ← [];
8         end
```

8.3 Efficient Memory Integrity

This subsection discusses some optimizations of the memory integrity design to increase parallelism and prover efficiency.

8.3.1 Commutative Operations

In some workloads, transactions frequently visit a small number of "hot spots". For example, a TPC-C payment transaction always updates the Yield-To-Date (YTD)

balance of the warehouse. These hot spots severely limit the utility of transaction merging discussed in Section 8.1 because transactions updating the hotspots conflict with each other, making every batch only contain a single transaction. The wrapped transaction degenerates to a single chain of transactions.

One important observation is that the transactions update the YTD balance by increasing it with the order payment balance. Such operations are, in fact, commutative. In other words, changing the order of increments will not change the final result. Therefore, we allow the transactions to conflict on the hot spots and accumulate the commutative operands after all of them finish. To do so, we let transaction programmers identify hotspot variables and create a "whitelist" of variables, where deterministic reservation allows transactions to conflict on them as long as the operations are commutative. For a non-conflicting batch, we update the variables in the whitelist after all the transactions finish.

Figure 8-6 shows an example of merging such conflicting transactions. We have two transactions, $T1$ and $T2$. $T1$ increases 1 to YTD. $T2$ reads YTD, writes it to DB[Z], and increases 2 to YTD. Obviously, $T1$ and $T2$ conflict with each other. However, the increments made by the transactions are commutative. So, we can still merge $T1$ and $T2$ and accumulate the increments after the batch. However, there is a caveat. Note that $T2$ uses the value of YTD to write to DB[Z]. Although we merge the transactions horizontally, we still need to keep an implicit order in them. In our example, we set this order to be $T1 \rightarrow T2$. This order implies that the YTD variable read by $T2$ should have already been incremented by one. To address this caveat, we add some logic in the circuit to add an offset to the YTD variable (shaded in green). Finally, after the logic of $T1$ and $T2$, we add the logic of updating the digest: it writes YTD + 1 to DB[Z] and adds three to YTD.

Note that this trick works when the commutative operations are "blind". In other words, they do not depend on the read values. Otherwise, some updates might not take effect and the offset (e.g., YTD += 1 in $T2$) we hardcoded into the circuit might be wrong. This will require the client to perform extra circuit checks.

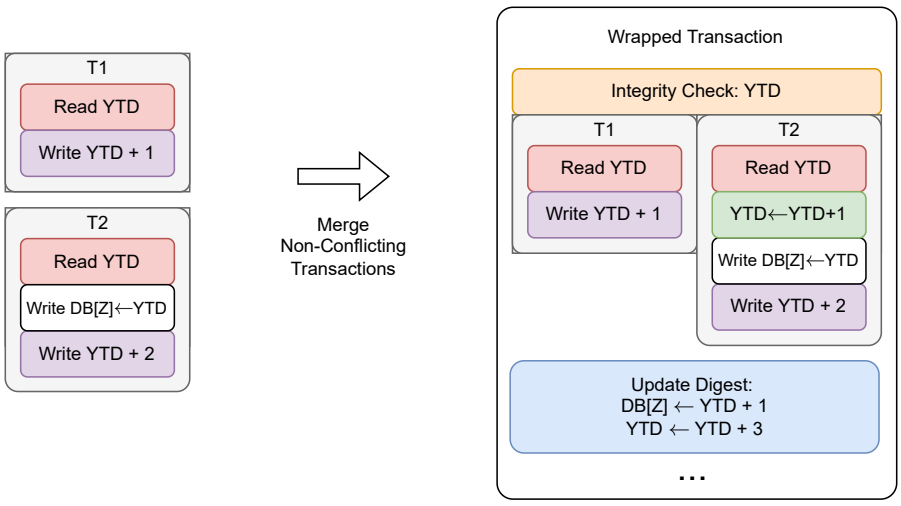


Figure 8-6: **Example Merging Conflicting Transactions** — We merge conflicting transactions if they conflict through commutative operations.

8.3.2 Shortcutting and Vertical Merging

Sometimes, the operations on the hotspots are not commutative. To address this problem, we introduce *variable shortcutting*. In short, we do not let memory integrity handle hotspot variables but use circuit wires instead — we can get the value from a gate in one of the previous transactions directly.

Figure 8-7 shows an example of shortcutting. All three transactions, $T1$, $T2$, and $T3$ read and modify DB[A]. They certainly conflict with each other. Noticing the pattern of frequent access to the same variable, we bypass the memory access operations and pass the value of DB[A] directly as a local wire. Note that $T2$ reads DB[A] right after $T1$ writes the value X + 1 to DB[A]. We let the write operation of $T1$ and the read operation of $T2$ cancel out and insert an assignment X $\leftarrow$ X + 1. Similarly, we remove the write operation of $T2$ and the read operation of $T3$ and insert an assignment X $\leftarrow$ 2 * X. However, we have to keep the last write operation for each place because future read operations might depend on this value. Therefore, we keep the write operation of $T3$. Effectively, we "vertically merge" the three transactions into a single one. Considering the corresponding wrapped transaction, we only need one integrity check and one final update.

As a remark, vertical and horizontal merging work in different situations and pose

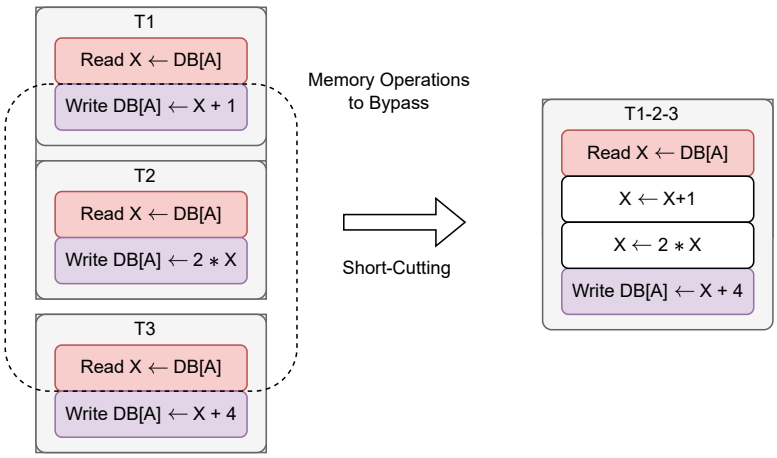


Figure 8-7: **Example Shortcutting** — We merge conflicting transactions vertically to reduce memory operations.

different workloads for the client to check the circuit correctness. But they share some commonalities, as described below.

- Both benefit from proof aggregation of the authenticated dictionary scheme. It is straightforward to see horizontal merging collects read values of non-conflicting batches and verifies their lookup proofs all at once. Vertical merging also collects read values from a vertical chain of transactions and verifies their lookup proofs together.

- They contribute to less memory integrity logic and a smaller circuit.

8.3.3 Sharding and Horizontal Circuit Cutting

The proposed design in Section 5.2 uses a single memory digest for all the data in the database. Though we have considerations that the circuit should be as computationally lightweight as possible, we realize that splitting the memory digest into multiple instances with each covering a portion of the database is favorable. Modern database workloads usually do not have significant data skew. Data partitioning is frequently seen in existing large-scale database systems like Kallman et al. [87]. Similarly, we can partition the data into many shards and maintain a memory digest for each of them,

and only provide essential proofs for every transaction. This will enable two advantages: (a) the server is able to process the memory integrity model in parallel so that multi-threading will boost up the performance; (b) the circuit's structure has more parallel patterns, a feature that enables an even faster proving process [156, 142].

We call a sharding scheme static if it does not change after the server starts. Given a static partitioning that maps the database memory domain M into a partition index in $[k]$, $\mathsf{Partition} : M \to [k]$, we maintain k different instances of memory integrity models on the server side. Moreover, the circuit will accordingly have k local digests. Whenever a transaction starts, it inspects every read value supplied from the server and infers the partition index for each of them. In parallel, the circuit checks the corresponding local digest against the values and the proofs. After every check succeeds, the circuit proceeds to the content of the transaction.

In the ideal case, every transaction only visits a single partition. We can arrange them into k independent subcircuits such that the i-th subcircuit only contains the transactions visiting the i-th partition. Since each partition has its local digest, there are no wires across different subcircuits. It suffices to prove that each subcircuit is correctly evaluated. This is similar to the circuit cutting we discussed in Section 8.2, except that we cut them horizontally instead of cutting vertically.

This heuristic applies to general cases as well. Intuitively, we identify transactions that visit different places and chain them to form independent subcircuits. When a transaction involving multiple partitions shows up, we merge the affected chains and connect it with this transaction. We split the chains again after the multi-partition transaction. For the parallel portion of the circuit, we cut it into horizontal pieces and prove them in parallel. Since the server has the freedom of arranging the transactions, it can create more parallelism by carefully placing multi-partition transactions.

We present an example in Figure 8-8. Assume the database is partitioned into three parts, and the majority of transactions only visit a single partition. We have 14 transactions in Figure 8-8, and only $T11$ visits both Partition 1 and Partition 2. For this example, we use three independent memory digests to track the data of the three partitions individually. If a transaction visits only a single partition, it only

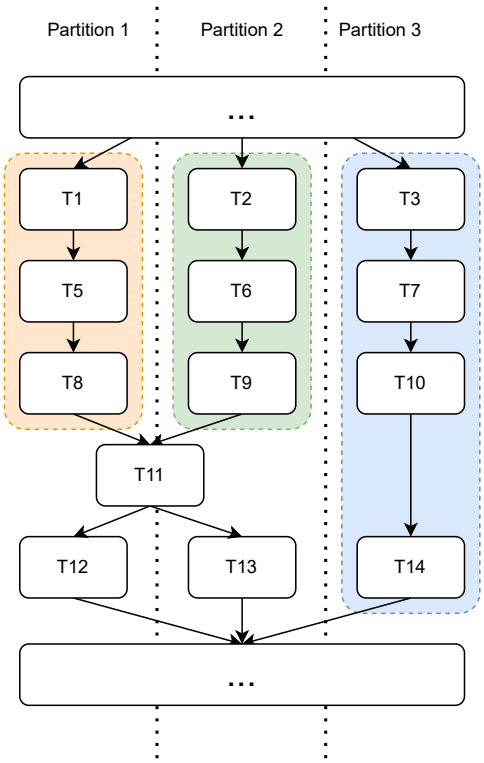


Figure 8-8: **Example Sharding** — We prove transactions on different shards in parallel.

checks the proof for the corresponding digest and updates that digest. Otherwise, it needs to check proofs against every digest involved and might update all of them. If transactions visit different digests, we can prove them in parallel. For the example in Figure 8-8, we prove three circuit pieces in parallel: $T1 \rightarrow T5 \rightarrow T8$ (shaded in orange), $T2 \rightarrow T6 \rightarrow T9$ (shaded in green), and $T3 \rightarrow T7 \rightarrow T10 \rightarrow T14$ (shaded in blue). Because $T11$ visits two partitions, it checks the proofs with the two digests of Partition 1 and Partition 2. Which digests $T11$ updates depends on the partitions it writes to. Proving $T11$ can be concurrent with proving the transactions visiting Partition 3.

There remain several challenges we need to explore and solve.

- The first question is how to arrange the transactions and how to split the circuit. With the freedom of putting any transaction in any place, we have an exponentially large space of valid parallel interleavings. It remains to model the prover time cost and convert the performance problem into a global optimization instance. This becomes harder when the workload is heterogeneous.

- The second question is how to support a dynamic sharding scheme. Assuming limited prior knowledge of the workload, an inappropriate sharding scheme could result in degraded performance if, for example, the skew is very high. Dynamic sharding schemes can adjust the partition function periodically. However, it is challenging to apply a dynamic sharding scheme in our scenario because we have a limited budget for circuit complexity of the wrapped transaction. It needs to determine partition indices efficiently. Regardless of the complexity budget, it remains to see how to coordinate a dynamic sharing scheme inside and outside the circuit. Both sides have to keep synchronized during the entire execution process. Otherwise, the circuit might use a wrong digest. We defer this to future work.

- Even with static sharding schemes, the circuit needs to maintain k accumulators locally, which in the worst case incurs an extra linear circuit overhead. A natural question would be, can we do better than this? A preliminary idea

is to maintain a *master accumulator* to coordinate a group of accumulators. However, maintaining the proofs needed by the *master accumulator* could again become a central bottleneck on the server.

Algorithm 15: Deterministic Reservation

Input: A set of transactions $\mathcal{T} = \{T_i\}$ and the batch size m

R $\leftarrow [\infty, \infty, \ldots, \infty]$; `/* size` n `*/`
Func Reserve (T_i)**:**
  Run T_i;
  Whenever T_i *reads* DB.Data $[x]$**:**
    T_i.read_set.$insert(x)$;
    if x *is in* T_i.WriteVals **then**
      return T_i.WriteVals$[x]$
    end
    return DB.Data $[x]$
  Whenever T_i *writes* val *to* DB.Data $[x]$**:**
    T_i.WriteVals.$insert(x, val)$;
    Atomic *do***:**
      if $T_i.\rho < R[x]$ **then**
        `/* smaller number means higher priority */`
        $R[x] \leftarrow T_i.\rho$
      end
Func Commit (T_i)**:**
  `/* Check the reservations */`
  for $x \in T_i$.read_set $\cup\, T_i$.WriteVals **do**
    if $T_i.\rho < R[x]$ **then**
      return no
    end
  end
  `/* Apply the updates */`
  for $x, val \in T_i$.WriteVals **do**
    DB.Data$[x] \leftarrow val$
  end
  return yes
Func Process $(\mathcal{T}, m)$**:**
  Generate priorities $T.\rho$ for every transaction $T \in \mathcal{T}$;
  do
    Reset R to be $[\infty]$;
    Take m transactions as $\mathcal{T}'$ from $\mathcal{T}$;
    In parallel call Reserve(T) for all $T \in \mathcal{T}'$;
    In parallel compute $r_T =$ Commit(T) for all $T \in \mathcal{T}'$;
    Provide a non-conflicting batch $B = \{T | r_T = \text{yes}\}$;
    $\mathcal{T} \leftarrow \mathcal{T} \backslash B$
  while $\mathcal{T} \neq \emptyset$;

Chapter 9

Interactive and Hybrid Verifiable Databases

The bulk of this thesis focuses on using techniques from verifiable computation to check the integrity and semantic properties of a database system. For certain types of workloads, we can potentially get away with building a verifiable database system from only lightweight cryptographic tools, such as Merkle trees [108] and authenticated data structures [4]. These techniques apply when a client's query makes only *local access* to the database—when the query reads or writes only a few rows of the database. In realistic database workloads, many, though not all, queries are "local" in this way. Verifying such local database operations can be extremely fast (nearly as fast as unverified execution), but systems that rely on this local verification cannot handle arbitrary SQL queries. And, how to exploit parallelism is not straightforward. We propose designing a hybrid system that achieves the desirable properties of both general verifiable databases (i.e., the main system we describe in previous chapters) and lightweight verification techniques based on authenticated data structures. This hybrid system would give good performance for local queries but could fall back to the general verifiable techniques to handle more complex and parallel queries that might read or modify the entire database state.

We first describe the lightweight interactive technique in Section 9.1. Then, we present the hybrid design in Section 9.2.

9.1 Interactive Design and Real-Time Transactions

A simple way to achieve serializability and atomicity is to let the server and client run the transactions one by one. The client again holds a digest to track the data. But, instead of letting the circuit check lookup proofs, the client does the checking by itself. As long as data integrity is guaranteed, running transactions one by one naturally implies serializability since the ground-truth interleaving is sequential.

In Litmus, we let the server generate a wrapped transaction, run it, and prove it using verifiable computation. If we let the client "run" a sequential wrapped circuit, we no longer need verifiable computation because the client is trusted. The dataflow between the server's native computation and the circuit now becomes network communications between the server and the client. Therefore, we call this an interactive design. This interactive design is not our contribution [179].

Figure 9-1 shows how the interactive design works. The server and the client process the transactions sequentially. Both of them maintain a digest to track the data. The client executes the main logic of the transactions. Whenever there is a read operation, the client sends a read request to the server. The server looks it up from the database and computes a lookup proof. It sends back the value and the proof to the client, who performs an integrity check. Whenever there is a write operation, the client sends a write request to the database with the new values. The server applies the changes to the database. For some ADS schemes, updating the digest needs auxiliary data, which might include the whole database. An example is the Merkle tree [108]. The client cannot update the digest by itself. In this case, we ask the server to provide the updated digest together with a proof. In the case of a Merkle tree, this proof is the Merkle path. Both the server and the client perform updates to the digest. Then, the client starts running the next transaction.

The interactive design has the following pros and cons compared to Litmus:

The most significant benefit is removing the long latency from the VC scheme. This is especially useful when the transactions are real-time, where the end-user is waiting for the output to make prompt decisions.

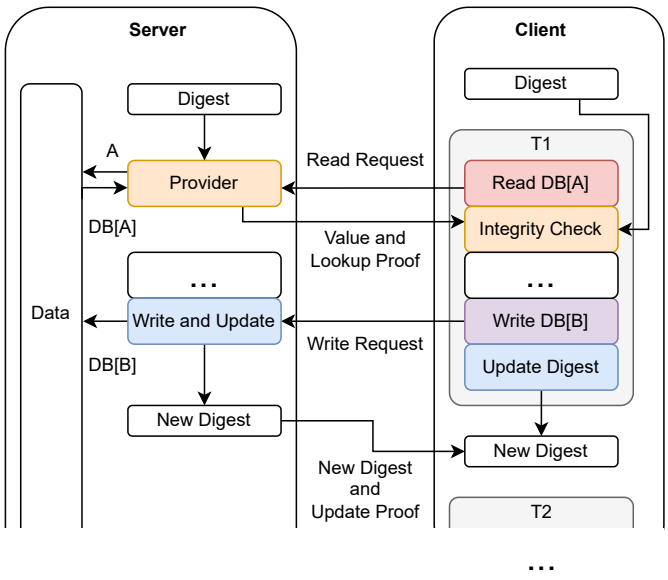


Figure 9-1: **Interactive Verifiable Database** — Conventional verifiable databases also provide serializability.

The throughput, however, is strictly bounded by the network latency and the transaction complexity because now every memory operation incurs a network round trip. If the transaction visits many places in memory, e.g., a full scan, it will incur significant communication overhead.

Lastly, it is impossible to exploit any parallelism in the workload because the interactive design relies on running the transactions one by one to achieve serializability.

A natural question is whether we could get the best of both worlds. Next, we discuss a hybrid option in Section 9.2.

9.2 Hybrid Design

So far, the Litmus design we proposed in Chapter 6 and the optimizations we discussed in Chapter 8 use verifiable computation schemes and authenticated dictionaries as the fundamental tools to prove correctness and transactional properties. Using massive batching is effective in improving the throughput. However, our current design has a fundamental issue of having long latencies. Due to the current status of cryptographic tools, the verification latency is inevitable for batched verification.

To address the long latency, 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 digests 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 aggregation, but it will have 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. Further, we can let the server store the proofs and send them to the client on demand. In some applications, the client might be satisfied with a probabilistic guarantee where it randomly picks transactions and requires proofs for them. There also could be special transactions that check for application invariant properties. As long as the invariants are satisfied, the client is good to go. In this case, the server only needs to prove these special transactions.

Suppose the client is happy to stay online and provide the transactions to the server one by one. In that case, the server is forced to run the transactions sequentially and atomically, as the whole process is literally sequential. Chapter 10 will include some interactive baselines and show that the network latency is likely a problem for them. The reader might wonder if an interactive baseline with deterministic reservation might outperform Litmus-DRM. We argue that such a baseline does not form a full solution to verifiable DBMS, simply because, in some cases, the client is not able to check whether the server performs deterministic reservation correctly. When the writing targets depend on the read values, the client itself is *not able* to locally produce the interleaving, so the server has to send the client all the values being read and all the values being written in plaintext. To verify these values, the client has to *locally execute all the transactions* to ensure the written values are correct. In other words, the system downgrades to a simple key-value storage, where the client only

Algorithm 17: A Simple Hybrid Verifiable DBMS

Input: Initial Digest d_0 and Data D.

Initialize $d \leftarrow d_0$;

`/* Batch Mode */`

Whenever *Receiving a transaction batch* B**:**

  Run B in the batch mode;

  Update d;

  Send the result, the new digest d, and the proof to the client;

`/* Interactive Mode */`

Whenever *Receiving a read Request* (*read,* k)**:**

  $v \leftarrow D[k]$;

  $\pi \leftarrow \mathsf{GenReadProof}(k, v)$;

  Send v, π;

Whenever *Receiving a write Request* (*write,* k, v)**:**

  `/* Update` d `*/`

  $\mathsf{UpdateWrite}(k, v)$;

sends requests of reads and writes and gets raw values as the response. If the client is powerful enough to run the transactions locally and host all the values demanded, there is no reason to delegate the database.

However, we present a hybrid system that supports both the interactive and batch modes. It decides which mode to use depending on the underlying workload, the network environment, and client preference. If the transactions are simple enough, the write destinations do not depend on the read values, the network latency is not significant, and the client is happy to stay online, the interactive mode is preferred. The batch mode always works, so it serves as a backup. These two modes are compatible if using the same memory integrity.

Algorithm 17 shows the idea of the hybrid system. For simplicity, we let the client decide which mode it wants to use. If it wants the server to process in batch mode, it sends a batch of transactions. Otherwise, it sends individual read and write requests.

Correspondingly, if the server receives a transaction batch, it runs the transactions with the batch mode of Litmus. Otherwise, it processes the message as an interactive request. If the message is a read request, the server calls the memory integrity provider GenReadProof to provide the value and proof. If the message is a write request, it updates the digest by UpdateWrite accordingly.

We defer the implementation of this hybrid scheme to future work.

Chapter 10

Evaluation Results

We have built a prototype verifiable database system [164] with prover pipelining and evaluated it against the Yahoo Cloud Serving Benchmark [52] and the TPC-C benchmark [94] 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 Section 10.2). 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 [58]. 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 verifiable computation framework with Pequin [130]. 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 [97], an optimized version [76] of the Pinocchio scheme [118].

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 concurrency control algorithm is deterministic. For interaction between the client and the server, we simulate a thread sleep of 1 ms or 100 ms, where 1 ms represents latency in a LAN network, and 100 ms is common for WAN networks and the Internet (e.g., from Boston to Barcelona). The implementation assumes existence of *division-intractable* hash functions [116] and therefore does not perform primality checks. The underlying curve we use in the proving system is the Barreto-Naehrig-128 curve (BN-128) [11]. To avoid the engineering effort of multi-precision scalar operations, we use a *small* RSA group. Specifically, the RSA group we use for evaluation has a modulus of 126 bits so that multiplication between two group elements will not cause overflows in the 253-bit base field of BN-128 element[1]. We also included a set of evaluation results of Litmus with a 1024-bit RSA group in Section 10.5 to study the effects of increasing the security strength of the RSA-based authenticated dictionary.

We run Litmus on a machine with two Intel Xeon 5218R CPUs (20 cores per CPU, 2× with hyperthreading). In total, the machine has 80 CPU threads. 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 concurrency control

[1]The base field of BN-128 is $\mathbb{F}_q$, where $2^{253} < q < 2^{254}$.

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 described in Chapter 6.

- *AD-Interact-1ms/100ms* is an interactive baseline using authenticated dictionaries 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. Issuing the transactions sequentially guarantees serializability and atomicity. The simulated latency 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-1ms/100ms* is the folklore approach to realize authenticated data delegation with simulated network latencies of 1ms/100ms. 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.

- *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.

10.1 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 10-1a 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 2.6 million. 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. Similar to the AD Interactive baselines, the Merkle Tree baselines hit the network latency bottleneck and plateau thereafter. However, the Merkle Tree baselines need a larger verification batch size to reach the bottleneck compared to the AD Interactive baseline. The reader might wonder whether the comparison is fair because the Merkle Tree baselines offer full-strength 256-bit security. Even our evaluation with a 1024-bit RSA group in Section 10.5 does not offer the 256-bit security because the BN-128 curve only offers a security strength of around 110 bits [107]. However, we argue that this is not a problem, since the interactive baselines are bounded by the network latency. Lowering the security of Merkle Tree to the level of Litmus does not improve the throughput. Further, 110 bits of security is enough for our application. 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 10-1b 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 verifiable computation framework has a significantly longer 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 concurrency control algorithm needs to wait for a large batch, and it synchronizes all the threads between the reserve phase and the commit phase. The latency of the interactive baselines remains stable because the latency is dominated by the simulated network roundtrip when the number of transactions is smaller than 10,000. Then, the latency starts to get dominated by the computation, as we can see clearly for the interactive-1ms baseline.

Figures 10-2a and 10-2b show the performance of Litmus and baselines on TPC-C New Order transactions and Payment transactions. We scanned the processing batch size for deterministic reservation and found that a smaller processing batch is preferable for both TPC-C transactions. The performances 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.

10.2 Sensitivity Study

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

Figure 10-3a shows how the throughputs 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 the concurrency control algorithm. We can see a factor of up to 36.2× between the Litmus-DRM and Litmus-DR because of prover pipelining. Figure 10-3b presents the latency information. When the processing batch size is extremely small, the deterministic reservation concurrency control 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 10-4 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 10-5 shows the time breakdown of Litmus-DRM running a verification batch of 2.6 million 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 a circuit template of a single transaction, the circuit generation only needs to duplicate the template and fill in the parameters. It always takes minimal time (not noticeable in the figure). The size of the proofs is constant, namely, 312 bytes per prover thread and 30 KB in total. It takes the client around 300 seconds to verify each proof. The key pair has a large size, but we can use universal VC schemes (Chapter 11) where key pairs are not necessary.

Figure 10-6 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 concurrency control 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 10-7 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 a 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.

10.3 Comparison with Elle

To understand how Litmus performs compared to alternatives, we evaluated Elle [92] 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 3.5 million transactions[2], Elle spent 576 seconds, reaching a throughput of $\sim$ 5.5k txn/sec. This is at the same level as reported in [92].

Both Cobra [140] 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 it 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.

10.4 Communication Cost

The size of the proofs is constant, namely, 312 bytes per prover thread. The client only needs to verify at most $75 \times 312 = 23,400$ bytes. The client spends around

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

2.13 seconds to verify each of the proofs. Altogether, the 60 proofs take around two minutes to check. However, the current verifiable computation framework generates large key pairs. As we can see from Figure 10-8, the communication cost is one of the limitations of Litmus. This issue can be addressed with universal verifiable computation schemes where we do not have to generate key pairs for every circuit piece.

10.5 Evaluation with 1024-bit RSA Group

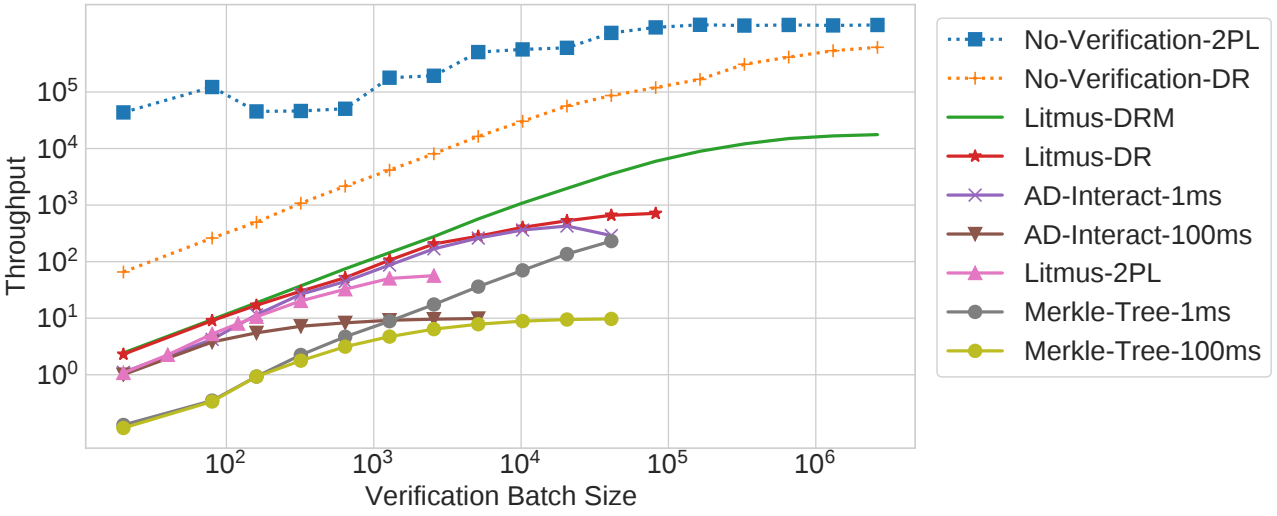
In this section, we present evaluation results of Litmus with a 1024-bit RSA group. As we mentioned at the beginning of this chapter, Litmus used BN-128 as the pairing-friendly curve. The BN-128 curve is over a finite field $\mathbb{F}_q$, where q is a 254-bit prime number. To avoid overflows during multiplication operations, each element can hold at most a 126-bit integer, which is far smaller than N, the modulus of the RSA group. A single element in $\mathbb{F}_q$ cannot hold an RSA group element. Therefore, we used $\lceil \frac{1024}{126} \rceil = 9$ field elements in $\mathbb{F}_q$ to represent an RSA group element. Specifically, we use a nine-dimensional vector $\mathbf{a} \in \mathbb{F}_q^9$ to represent an RSA group element h, where

$$h \equiv \sum_{i=1}^{9} \mathbf{a}^{(i)} \cdot p^{i-1} \mod N.$$

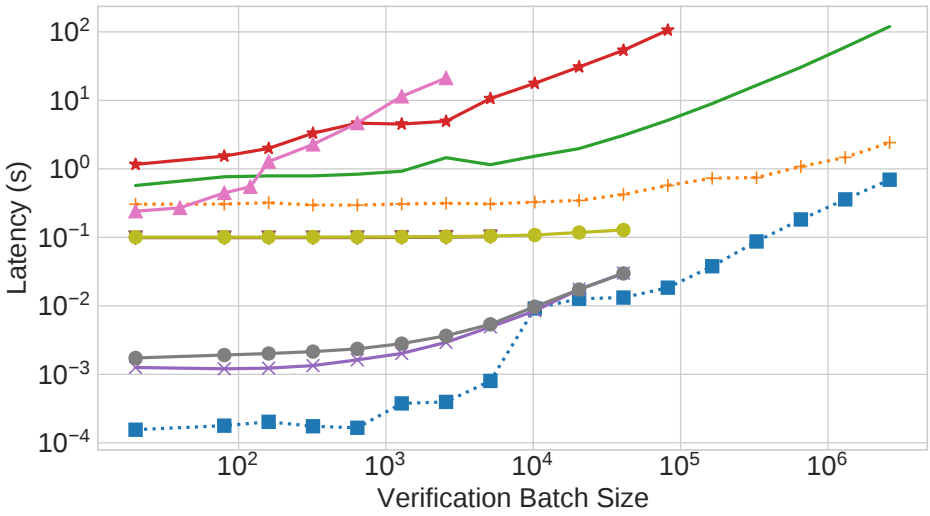
When performing fast exponentiations, we temporarily need to handle double-sized (18 elements) large integers before calculating the modulo residues. We refined our implementation with multi-scalar operations to handle large field arithmetic. We used a multi-scalar multiplication algorithm resulting in a linear number of R1CS constraints [95]. Specifically, we adopt the parameters of RSA group from the RSA Factoring Challenge's RSA-1024 [2]. To further improve the performance, we used the Proof-of-Exponent protocol to truncate the large exponent while doing the authenticated dictionary lookup / key-nonexistence proof verification [33].

Figure 10-9a and Figure 10-9b show the throughput and latency results on the YCSB benchmark, respectively. Overall, all the authenticated-dictionary-based base-

lines, including Litmus baselines and AD Interactive baselines, become slower due to the extra computation cost from the multi-scalar constraints necessary to support large RSA groups. The peak throughput of Litmus-DRM reaches around 7.4k txn/sec. Compared to Litmus with a smaller RSA group, increasing the RSA group size incurs a slowdown of 2.37×. The reason why the slowdown is smaller than 9× is that a significant proportion of the circuit is computing the batching exponents, which is not affected by increasing the RSA element size. Litmus only performs the RSA exponentiation a few tens to a hundred times (depending on the workload), thanks to the massive aggregation.

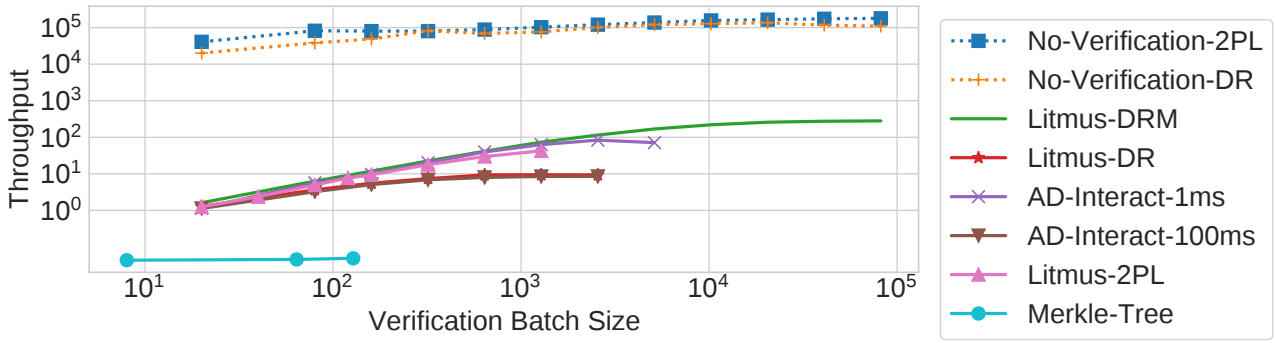


(a) Throughput

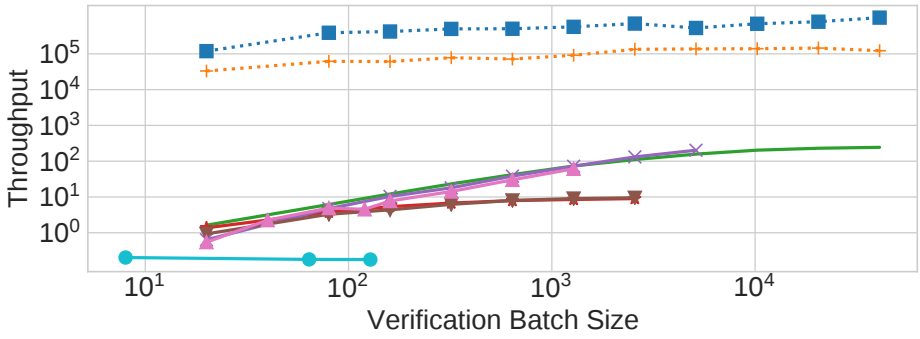


(b) Latency

Figure 10-1: Throughput and Latency vs Verification Batch Size - YCSB

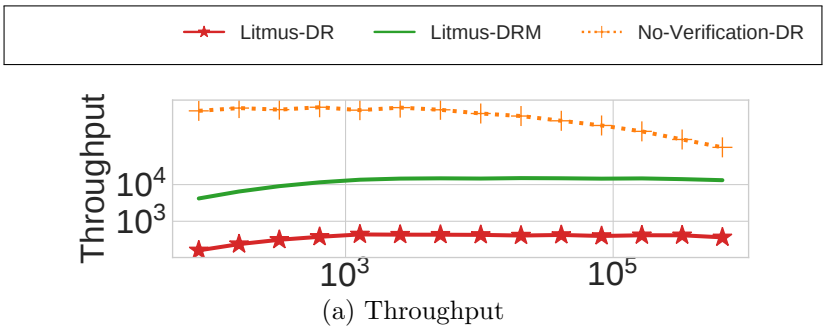


(a) New Order

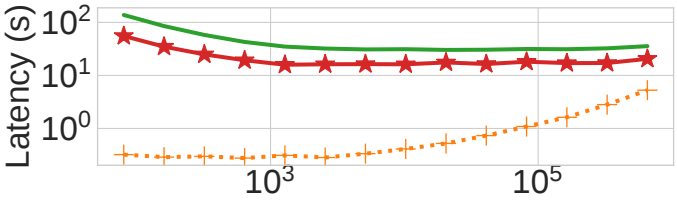


(b) Payment

Figure 10-2: Throughput vs Verification Batch Size - TPC-C



(a) Throughput



(b) Average Latency

Figure 10-3: Throughput and Average Latency vs Deterministic Reservation Processing Batch Size.

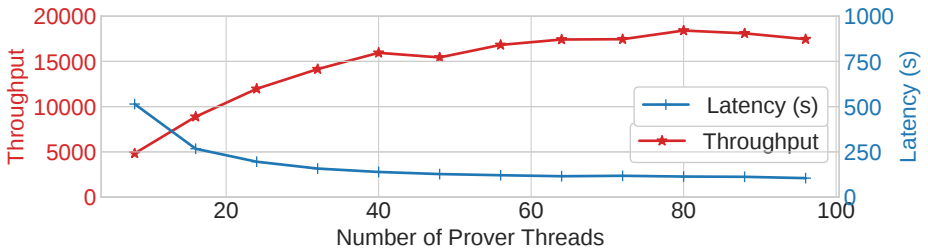


Figure 10-4: Throughput and Average Latency vs Number of Prover Threads for Litmus-DRM.

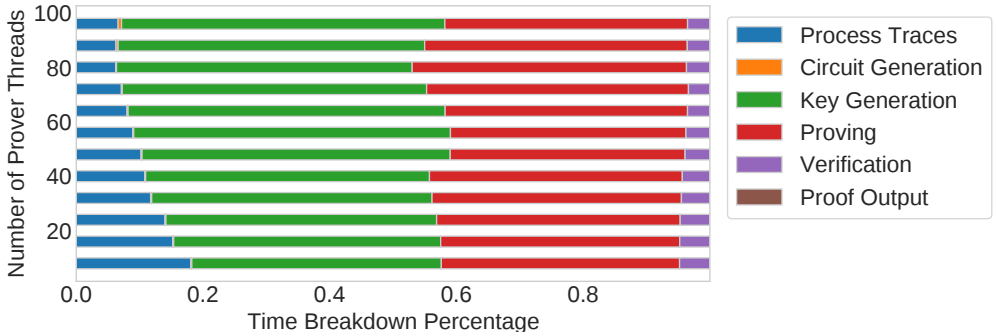


Figure 10-5: Time Breakdown vs Number of Prover Threads.

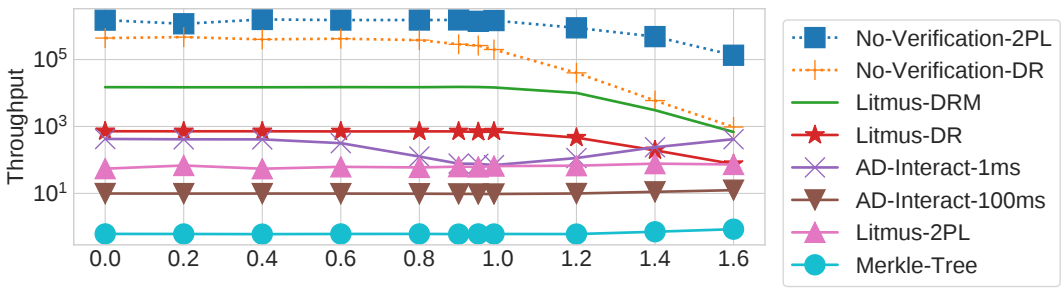


Figure 10-6: Throughput vs Contention Level

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

Figure 10-7: Performance of Litmus vs Table Size

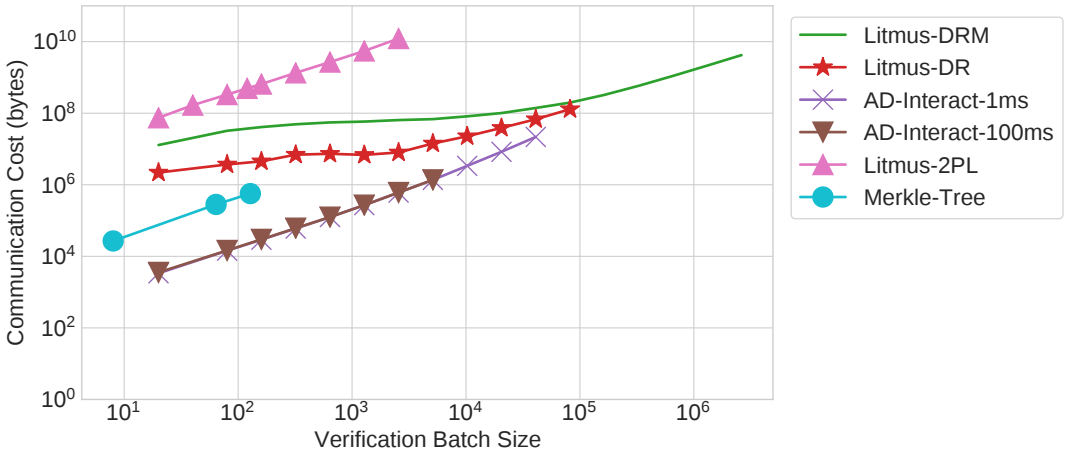
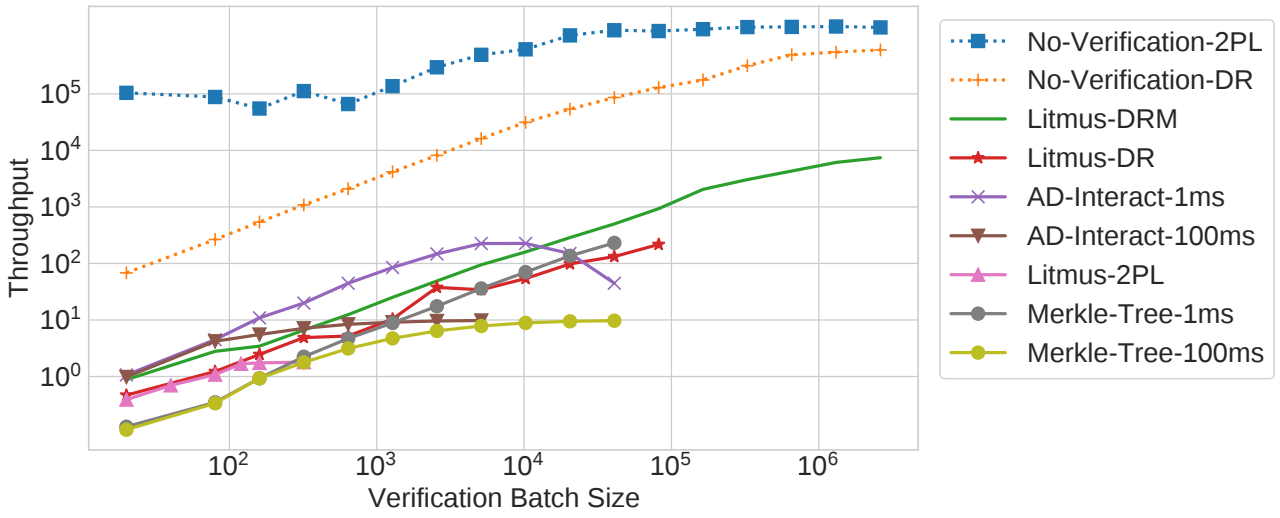
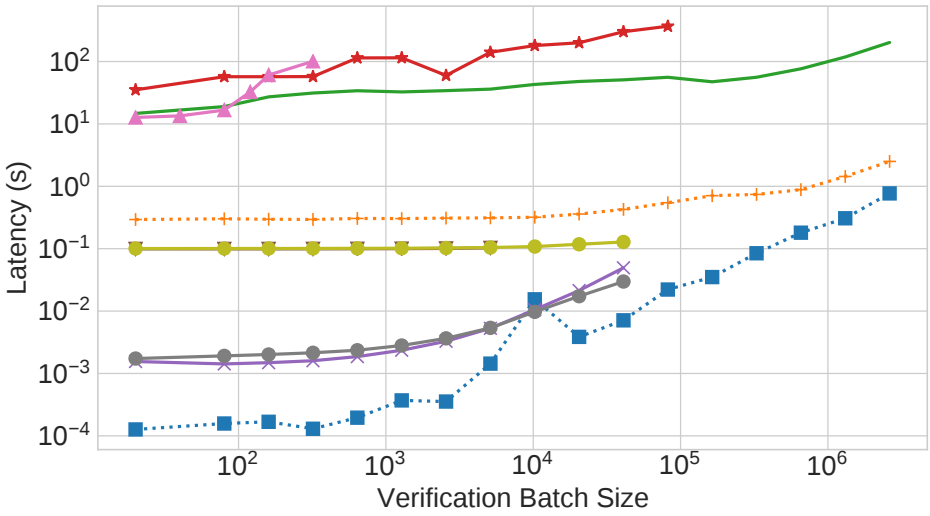


Figure 10-8: Communication vs Total Number of Transactions.



(a) Throughput



(b) Latency

Figure 10-9: Throughput and Latency vs Verification Batch Size with authenticated dictionaries using a 1024-bit RSA group.

Chapter 11

Extensions

11.1 Other Isolation Levels

In Section 3.2, we discussed the cryptographic formalizations of repeatable read as an example of other isolation levels and consistency verification. Now, we show how to modify Litmus to verify repeatable read and consistency.

Following the cryptographic formalization in Section 3.2, we take Repeatable Read isolation as an example. The cryptographic definition of Repeatable Read differs from that of Serializability mainly in the following two places.

- We organize the database states as a DAG in the former but as a list in the latter.
- The state transition requirements are different. For serializability, the transition is simply applying the transaction from a previous state to the next one. For repeatable read, the transition involves a node in DAG and its inward neighbors.

Although the definition of Repeatable Read seems complicated, it turns out we only need to modify how we use the memory integrity provider and checker.

Similar to serializability verification, we maintain a sequence of memory digests. This sequence reflects the time order of the data state changes. However, we allow a transaction T to verify the lookup witnesses against a historic memory digest. We

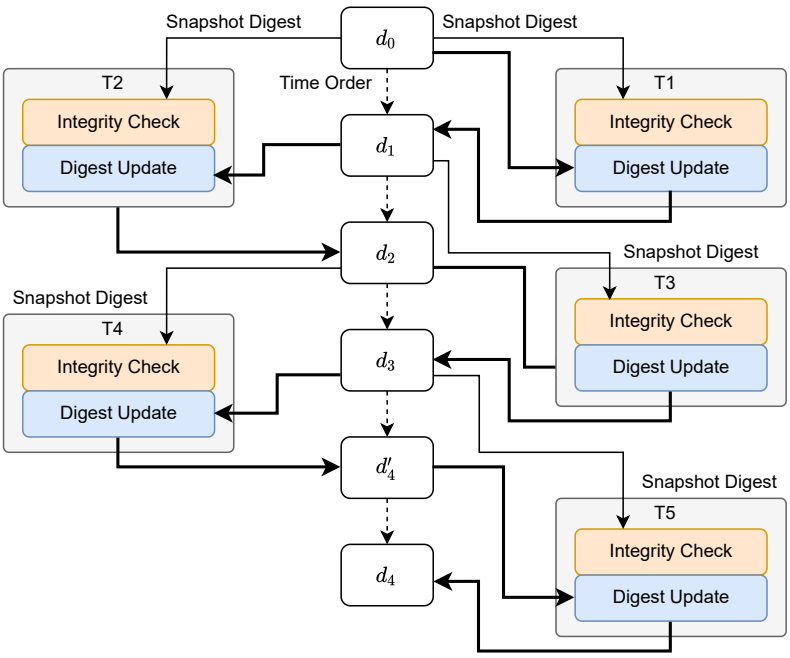


Figure 11-1: Example of Verifying the Transactions in Repeat Read.

call this digest the *snapshot digest* of the transaction. Effectively, this transaction started at the time when the data state corresponds to the snapshot digest. For the write operations, the transaction has to update the *latest* digest to a new one.

Figure 11-1 shows how we use the memory integrity checks. The transactions come from the example in Figure 3-1. In the middle of the figure, we have a line of digests. The digests $d_0, \ldots, d_3$ correspond to the data states $D_0, \ldots, D_3$ respectively. The digest d'_4 is an intermediate state after a transaction modifies the state but before the other transaction commits. Finally, the digest d_4 corresponds to the state D_4. The dotted arrows between the digests indicate the time order. Different from verifying serializability, a transaction can read from an earlier digest. For example, $T2$ reads from the snapshot digest d_0 but modifies d_1 into d_2.

There are two ways to reflect this change.

- The first one is to simply duplicate the wire corresponding to a historic digest by a fan-out gate and connect the wire all the way (potentially across millions of transactions) to the integrity checker. However, this makes the circuit well-formedness check a global property check. The client needs to go back and forth

to ensure the long-range wires are correct. This also affects the circuit cutting optimization because more wires cross the circuit pieces, and the server has to reveal all of these wires.

- The second one is to modify the memory integrity checker to accept a historic digest.

We mainly discuss the second approach here. How do we make sure that the digest is a historic digest instead of a fake one forged by the malicious server? We answer this question with a meta-accumulator, an accumulator that accumulates historic digests. Whenever the transaction updates a new digest, it adds the digest into the meta-accumulator. Whenever the memory integrity checker reads a historic digest, it also requires a proof that this historic digest exists in the meta-accumulator.

Algorithm 18 and Algorithm 19 show the memory integrity provider and checker for repeatable read isolation. Compared to the original memory integrity algorithms (c.f. Algorithm 1-2 and Algorithm 3-4), the new algorithms differ in the following places.

- The MemCheck function takes in a historic accumulator and a membership proof. It verifies the membership of the historic accumulator with the meta-accumulator. If the verification succeeds, the memory check proceeds with the historic digest instead of the latest digest.

- The provider now keeps a list of accumulators, AccList, to track the history. The GenReadProof function takes in the index of the historic digest and looks it up from the list. It attaches the historic accumulator and the membership proof to the returned result.

- Both the provider and checker add the updated digest into the meta-accumulator.

- After updating the digest, the UpdateWrite function additionally adds the digest to the list.

Algorithm 18: Memory Integrity Provider (on the Server Side) with initial database state agreed as $(\mathsf{meta\_acc}_0, g_0, D_0)$.

Input: AgreedInitState $:= \{\mathsf{meta\_acc}_0, g_0, D_0\}$

Func InitProvider ():
  $S \leftarrow \prod_{(k,v)\in D_0} H(k,v)$;
  acc $\leftarrow g_0$;
  meta_acc $\leftarrow$ meta_acc$_0$;
  $D \leftarrow D_0$;
  AccList $\leftarrow$ []; /* initialization */
Func GenReadProof $(i,\ k,\ v)$:
  historic_acc $\leftarrow$ AccList[i];
  if k *is in* D **then**
    $\pi \leftarrow g^{S/H(k,v)}$; /* generate the lookup proof */
    return (historic_acc, π)
  else
    $(A, B) \leftarrow$ Bézout$(S, \mathsf{Sample}(\lambda, 0, k))$; /* non-existence proof */
    return (historic_acc, A, B)
  end
Func UpdateWrite $(k,\ v)$:
  $v' \leftarrow D[k]$; /* old value */
  $\pi \leftarrow$ GenReadProof(k, v') ; /* must be a lookup proof */
  acc $\leftarrow \pi^{H(k,v)}$; /* update the digest */
  $S \leftarrow S/H(k,v') \cdot H(k,v)$; /* update the product */
  $D[k] \leftarrow v$; /* update the dictionary */
  /* Update AccList and the meta-accumulator */
  Append acc to AccList;
  meta_acc $\leftarrow$ Acc.Add (meta_acc, acc);
  return

11.2 Consistency

A *consistent* transaction changes the database only in certain ways. For example, 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 specialize the memory integrity checker into customized checkers.

If the transaction correctness is guaranteed, the programmer can add the consistency check in the transaction code. For example, the consistency check in a bank transfer system could be "the sum of involved accounts remains the same after the transfer transaction." If the consistency check fails, the transaction aborts the task.

Algorithm 19: Memory Integrity Checker (inside the Wrapped Transaction) with initial database state (g_0, s_0).

Input: AgreedInitState := $(\mathsf{meta\_acc}_0, g_0, s_0)$

Global variable acc, meta_acc maintained by a dedicated wire;
Func MemInit ():
  acc $\leftarrow g_0$; /* initialize the local digest */
  meta_acc $\leftarrow \mathsf{meta\_acc}_0$; /* initialize the meta-accumulator */
Func MemCheck $(\mathsf{historic\_acc}, \pi_h, k, v, \pi, A, B)$:
  /* Check the historic accumulator */
  if Acc.VerifyMem$(\mathsf{historic_acc}, \pi_h, \mathsf{meta_acc}) = \mathsf{no}$ **then**
    return 0
  end
  /* Continue the verification with historic_acc */
  if $\pi^{H(k,v)} = \mathsf{historic\_acc}$ *or* $(\mathsf{historic\_acc}^A \cdot g^{B \cdot H(k,v)} = g$ *and* $v = 0)$ **then**
    return 1 ; /* verification passes */
  end
  return 0
Func MemUpdate *(k, v′, v, π)*:
  if $\pi^{H(k,v')} != \mathsf{acc}$ **then**
    return 0 ; /* verification failure */
  end
  $\mathsf{acc} \leftarrow \pi^{H(k,v)}$; /* update the local digest */
  /* Add the new digest to the meta-accumulator */
  meta_acc $\leftarrow$ Acc.Add (meta_acc, acc);
  return 1

In Litmus, we provide an interface for aborting the whole batch — the transaction can simply set CommitFlag to 0 (c.f. Algorithm 7).

11.3 Special Verifiable Computation Schemes

We currently use Groth16 [76] to instantiate the verifiable computation primitive. However, this comes with a trade-off regarding the trusted setup. Namely, Groth16 assumes 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 here because the server might generate a malicious setup that allows it to cheat. However, this is not a problem for the situations in our evaluation, where the transactions' logic (i.e., circuits) is fixed and

we move the addresses to the input parameter. For example, we use the following template for the YCSB transactions:

```
1 Func YCSB_TXN(A, B):
2     Read X = DB[A]
3     Y = 2 * X
4     Write DB[B] = Y
```

Figure 11-2: Example Transfer Transaction.

The function YCSB_TXN takes the read address A and the write address B as inputs. Although the transactions might access different places, their (logic) circuits are the same. Therefore, the setup only needs to be done once.

However, if the transactions are not generated from a fixed template, the client (or a trusted party) 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 [64], whose setup is circuit-independent. This change might slow down the prover, but it can free the client from computing the trusted setup for an arbitrary circuit. Besides universal verifiable computation schemes, a *transparent* verifiable computation scheme like STARK [16] does not need trusted setups at all.

If we use a transparent verifiable computation scheme or a universal verifiable computation scheme, the chronological flow in Figure 8-3 can be further simplified into Figure 11-3. Both transparent and universal VC schemes do not require circuit-dependent trusted setup. This makes the whole process a single network round trip.

11.4 Multiple Clients and Multiple Servers

Thus far, we have only looked into the simple case where we only have a single server and a single client. Naturally, we wish to extend it to the case where there are

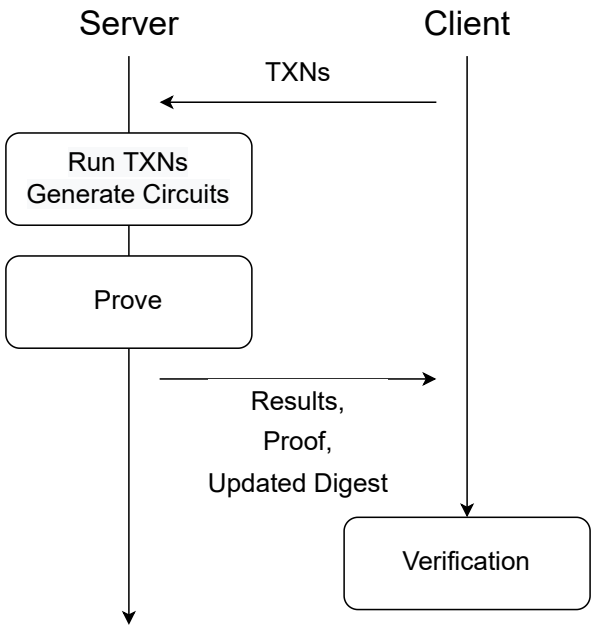


Figure 11-3: **Overview of Chronological Steps with Deterministic Reservation and Transparent/Universal Verifiable Computation Schemes** — With a transparent verifiable computation scheme, Litmus does not need any trusted setup. Alternatively, Litmus does not need circuit-dependent trusted setups if using a universal verifiable computation scheme.

multiple clients that could mutually distrust each other, or further, multiple servers that might collude.

An obvious required change with multiple clients is to the memory integrity model. Because the data could be written by other clients, from the view of a single client, the memory will no longer be consistent. This introduces many challenges and design choices, especially if we care about the privacy of each client. For example, it is hard to distinguish between a malicious server tampering with the data and another honest client writing the data. We have to rely on a public key infrastructure to authenticate an honest client's request. However, we need to hide the operation details of the client to protect privacy. Many existing tools like general-purpose zero-knowledge protocols can help us achieve this. However, for practicality, we might need application-specific designs to control the computation cost.

Having multiple servers enables us to serve massively large data and run distributed transactions, namely, transactions that visit more than one database server.

However, it also raises further questions: how can we verify that a transaction is indeed committed and has a consistent state among the multiple servers? We observe that with the help of secure multiparty computation (MPC) schemes, we do not actually need every server to perform verifiable computation. Given a secure MPC scheme that tolerates $n-1$ malicious parties [88], we can let just a single server run the verifiable computation framework and act as the "honest user" in the MPC. This observation can help us greatly reduce the computational overhead of verifiable computation.

11.5 Efficient Runtime Trace Collection

The readers might recall that the transaction wrapper collects runtime traces from the normal database component (c.f. Section 6.1.1). The runtime traces contain sufficient information to infer interleaving choices made by the concurrency control algorithms. For example, in Algorithm 8, the runtime traces contain partial orders of transactions determined by the Two-Phase Locking algorithm. With the partial orders, the server performs a topological sort of the transactions. The resulting sequential sorting order satisfies all the partial order constraints. Finally, the server proceeds with the transaction wrapper to arrange the transaction logics in the circuit according to the sorting order.

Section 4.1 mentioned that the runtime traces resemble logging records because both are information streams coming from the real-time concurrency control decisions and transaction processing. In fact, any existing logging scheme can provide the functionality of collecting runtime traces. The logging records contain information to replay the transactions to reach the latest database state. The correct replaying process implies a valid transaction order. Concretely, we plug the logging scheme into the normal database and log the records to the main memory. At the same time, we run the logging scheme's recovery algorithm with a dedicated thread to consume the log records in the memory and infer the transaction order from them. When the recovery algorithm tries to "replay a transaction", we let it construct the circuit

part instead. Efficient parallel logging schemes like Silo-R [183] and Taurus [168] can parallelize the runtime tracing collection. Similarly, parallel database replication schemes like Kuafu [79] can provide runtime tracing collection as well. This observation helps us migrate mature designs and novel techniques from the database logging literatures.

Chapter 12

Discussion

We next discuss some of our design insights and future directions. As mentioned in Chapter 1, 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.

12.1 Why Cryptography

Now, we justify the extra complexity of cryptography in Litmus. 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 aggregatability 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 a 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 all the data to the client in [83]).

12.2 Durability

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 access it. This property is indiscernible to software. One approach would be to design new hard drives with built-in secure enclaves [55, 98]. We believe recent advances in in-storage computation that enable data storage to perform programmable tasks [152] may be promising.

12.3 Limitations

There are a few limitations of the design and the implementation.

12.3.1 Design Limitations

The long latency is the most significant limitation of the design. It comes from the underlying cryptographic implementation. To verify a transaction, the system needs to perform the following steps:

1. The server prepares the values required by the transaction and their corresponding authenticated dictionary lookup proofs.
2. The server compiles the transaction's logic into cryptographic circuits.
3. Then, the server converts the circuit into an R1CS instance.
4. To prove the R1CS instance, the prover interpolates several large polynomials and performs polynomial arithmetic operations between them.
5. The prover produces polynomial commitments to convince the verifier that the polynomials are correct.

All these steps are computationally intensive. These steps happen in a sequential order except for Step 1. Steps 2-5 contribute directly to the latency.

The second design limitation lies in the optimizations. For example, we argued that using deterministic reservation can move the key generation and circuit well-formedness check off the critical path by letting the client generate the circuit on its own. However, this technique only applies to a subset of workloads where the transactions' write sets do not depend on the read values. If this is not the case, e.g., the transfer transaction in Figure 2-1 writes to Account B only when Account A's balance is no less than 100 USD, the client cannot learn the exact write sets of the transactions because it does not know the read values without the server's help. Therefore, the client cannot generate the circuit on its own — it cannot infer the deterministic batches with local knowledge. The deterministic concurrency control algorithm decides batches based on their access values.

12.3.2 Implementation Limitations

This current implementation is a proof-of-concept preliminary build for performance and speedup estimation purposes. As discussed previously in Chapter 10, there are two limitations:

- We include the key generation in the critical path while it is unnecessary. This factor results in performance underestimation. In the future, we will work on porting the code to Rust and replacing the verifiable computation implementation with Plonk, a Zero-Knowledge proving protocol with a universal setup.
- We assume the existence of a division-intractable hash function, for example, a hash function that is collision-resistant and outputs primes. In the implementation, we use a fake hash function $h(x)$ that outputs $P + x$ where P is a large prime number. This brings two effects: (a) the non-existence proof of key lookups might occasionally fail because the large integers could have common divisors, and (b) this skips the Pocklington verification. It results in performance overestimation (by a constant factor).

Meanwhile, various better engineering choices showed up. Rust is a system-oriented programming language that became popular for cryptographic libraries in

the recent years. Many existing implementations of cryptographic primitives were written in Rust. To name a few, `arkworks` [6] is a Rust ecosystem for zkSNARK programming; `Jellyfish` [138] is a Rust implementation of the Plonk proving system; `Circom` [80] is a language for arithmetic circuit definition; `Bellman-Bignat` [116] and `xJsnark` [95] provide efficient large RSA group operations in zk-SNARKs. These implementations provide a strong incentive to re-implement Litmus in Rust.

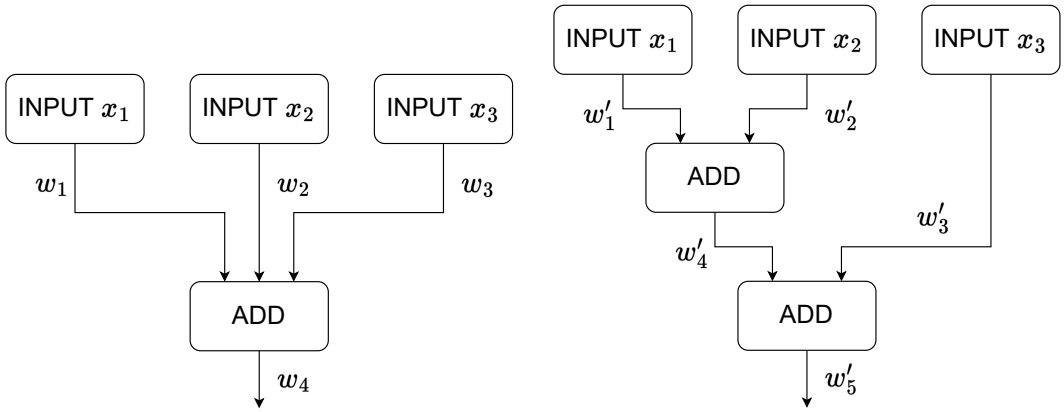
In theory, Litmus only requires information from the normal database component to infer data accesses and transaction orders. Therefore, the database component can be decoupled from the codebase. In other words, Litmus can exist as a middleware, and users can plug it into existing database distributions like PostgresSQL, as long as the database distribution exposes data logging records.

Chapter 13

Hydra: Succinct Fully Pipelineable Interactive Argument of Knowledge

This chapter presents Hydra, a succinct fully pipelineable interactive argument of knowledge protocol. It can replace Groth16 [76], the zkSNARK system we use as a verifiable computation scheme in Litmus. Compared to Groth16, Hydra has an asymptotically better prover computational complexity but a worse verifier computation complexity. When the circuit has horizontally repeated structures (e.g., the optimizations in Section 8.3), optimizations from the CMT protocol [54] apply to Hydra as well. Hydra directly accepts an arithmetic circuit as input, so it is programmer-friendly, whereas Groth16 needs the programmer to convert the circuit into a constraint system. Moreover, Hydra is friendly for parallel machines and easy to pipeline since all the circuit layers can be proven simultaneously. Lastly, the trusted setup of Hydra comes from the polynomial commitment. Therefore, the trusted setup is circuit-independent (when the circuit width is bounded by a known constant) and therefore one-time. In other words, Hydra is universal.

Specifically, Hydra uses 2-fan-in arithmetic circuits as its input, whereas Groth16 takes R1CS instances. To use Hydra in Litmus, we convert the general arithmetic circuit into a 2-fan-in arithmetic circuit instead of an R1CS instance. Figure 13-1



(a) Multi-Fan-in Gate (b) Two-Fan-in Gates

Figure 13-1: Converting a Multi-Fan-in Gate to Two-Fan-in Gates.

shows an example of converting a gate with multiple input wires (Figure 13-1(a)) into several gates with two input wires (Figure 13-1(b)) without changing the logic. Theoretically, any commutative operation with k input wires can be replaced with $k-1$ two-fan-in gates.

Hydra can proceed either interactively or non-interactively. For the non-interactive approach, the server produces a proof and sends it to the client, who accepts or rejects it. The chronological steps are similar to that described in Litmus so far. The non-interactive Hydra directly fits in the framework. For the interactive approach, the prover and the verifier need to communicate back and forth. In the end, the verifier decides whether to accept or not. The interactive Hydra occurs $O(\log w)$ rounds of network trips, where w is the circuit width. Although the asymptotic communication complexity is better than the interactive mode of Litmus (when the width w is no more than 2^d, where d is the circuit depth, which is true for the vast majority of workloads), Hydra could suffer from the network latency when the circuit is wide.

Now, we discuss the details of the *Hydra* proving system.

The following sections in this chapter present *Hydra*, a novel verifiable computation system. Hydra introduces two new disjoint interactive argument schemes geared towards the efficient pipelining of circuit verification. The first is specific to subcircuits, where a deep circuit is broken up into smaller parts and proven concurrently.

The second is a more general scheme where all layers of the circuit can be proven in parallel, removing the dependency on the layer-wise synchronous execution of the protocol. Compared to non-interactive SNARKs which rely on knowledge type assumptions (or the Random Oracle model) and non-interactive arguments based on standard assumptions that are not useful in practice, Hydra achieves a sweet spot with a practical approach. From standard assumptions, Hydra collapses the round complexity to polylogarithmic to the width of the circuit, but only incurs a polylogarithmic blowup in bandwidth and verifier time complexity. We implement the full verification flow, including both protocols and a logic parser used to convert traditional logic circuit compilation outputs into provable layered arithmetic representations. We perform experimental evaluations of our proposals and demonstrate protocol time efficiency improvements of up to 34.8× and 4.3× respectively compared to traditional approaches on parallel hardware.

13.1 Interactive Proofs and Arguments

With the rise of cloud-based computing, blockchain technology, and other applications relating to offloaded computation, the demand for security and trust in the accurate execution of a delegated task has grown prominent. This has motivated the field of verifiable computation and the concept of an *interactive proof*, where a computationally weak verifier (denoted as $\mathcal{V}$) can be efficiently convinced that, with an overwhelming probability, a computationally unbounded prover (denoted as $\mathcal{P}$) has correctly computed a requested task. The protocol downgrades to an *interactive argument* if the prover is computationally bounded. The key property in these notions is that $\mathcal{V}$ is able to check the proof of correctness that $\mathcal{P}$ provides in asymptotically less time than it would take to recompute the said task by itself, thus resulting in an advantageous benefit for $\mathcal{V}$ in outsourcing the computation.

Interactive proofs and arguments were first introduced by [73] in the 1980's and have been discussed in the literature since then, leading to important advances in cryptography and complexity theory such as the celebrated proof of IP=PSPACE

[132]. However, they were largely considered to be of theoretical promise and not applicable for production or practical use, mainly due to the high computation overhead required to generate such proofs. It is only recently that work to refine and scale these theoretical concepts for practical use became prevalent.

Particularly, a significant amount of such research has been based on the seminal general purpose interactive proof due to Goldwasser, Kalai, and Rothblum [71] (henceforth referred to as GKR or the GKR protocol). In this environment, the prover convinces the verifier of the validity of an arithmetic circuit evaluation composed of addition and multiplication gates of fan-in size two over some finite field. A notable property of their powerful protocol is that it is doubly efficient, meaning that the prover runs in polynomial time in the circuit size and the verifier runs in time sublinear in the circuit size. Despite this progress, the underlying GKR protocol runtime as well as the round complexity still remain entirely dependent on the depth of the circuit, as the protocol is fundamentally based on a layer-by-layer proof approach to ensuring the validity of the output. This quickly becomes a major bottleneck for a variety of deep circuits comprised of many layers, where the protocol efficiency begins to seriously degrade. Unfortunately, it is not straightforward to exploit parallelism naively across the depth of the circuit because, at a high level, it would expose information about adjacent layers while they are being proven, which would destroy the security of the protocol. As a result, GKR-based proofs have been generally considered to be impractical in such instances with deep circuits and restricted only to circuits of shallow constant-bounded depth.

In this chapter, we contribute theoretical proposals and practical improvements which come together to form a complete, robust framework that has wide applicability. In summary, this chapter introduces the following succinct[1] interactive argument protocols:

- The subcircuit protocol. This argument scheme allows for a large, deep circuit to be split up depthwise into multiple smaller data-dependent subcircuits. Each of these subcircuits are then proved in parallel and the proofs are combined at

[1] "succinct" refers to the property that the proof size is polylogarithmic in the size of the circuit.

the end. With this protocol, the application can pipeline the circuit generation and proving process.

- The Hydra protocol. This allows for the complete layer-wise independent parallelization of the GKR protocol. This argument scheme removes the need for a sequential layer-by-layer evaluation proof so that all layers can be proven in parallel. Hydra achieves a sweet spot between non-interactive argument protocols that rely on non-standard assumptions such as the Fiat-Shamir heuristic and interactive arguments based on standard assumptions that are not applicable in practice. From standard assumptions, we reduce round complexity of the GKR protocol to polylogarithmic in the width of the circuit, but only incur polylogarithmic increase to the bandwidth and verifier complexity.

In addition, we contribute:

- A novel logic handler that can take DAG logic circuits [111] compiled from C-style code and translate them on-the-fly into more efficient and provable circuit representations to be used with the GKR protocol.

- A full implementation of both protocols proposed as well as the parser. Through our experimental evaluations, we show efficiency improvements of up to 34.8× for the subcircuit protocol and 4.3× for the Hydra protocol on parallelized hardware.

13.2 Related Works

Cryptographers have made significant improvement to the GKR protocol. The prover runtime was reduced to $O(|C|log|C|)$ in Cormode et al. [54], close to linear for specific circuit compositions [143], and eventually reduced to where it is linear $O(|C|)$ with respect to the size of the circuit in Zhang et al. [169]. In a practical setting, the utilization of parallelism has additionally introduced great speedups for the proving process within individual layers of the circuit [145], across data-independent parallel circuits [157], and in different sub-copies [180].

Also, cryptographic primitives have been used with the GKR protocol for further improvements. The Fiat-Shamir heuristic [62] is commonly used to convert interactive proofs such as GKR to non-interactive arguments. However, it is important to note that these arguments are based on non-standard assumptions (either the random oracle model [12] or knowledge type assumptions) that are non-falsifiable and have been proven insecure by Goldwasser and Kalai [70] in the absence of a random oracle. In addition, polynomial commitments and other primitives are also utilized alongside GKR as the base for efficient argument schemes [22] [179] [40] [159] [169] [126], especially ones that guarantee zero-knowledge (zk-SNARKs). These zero-knowledge schemes are applicable in authentication systems and blockchains where $\mathcal{P}$ wants to prove to $\mathcal{V}$ that it knows some value x without revealing any information about x itself. One important aspect of these schemes is that they all rely on similar Fiat-Shamir transformations to convert them into non-interactive protocols. In fact, it is known due to Gentry et al. [66] that one cannot construct such SNARKs with constant $O(1)$ round complexity from standard assumptions. The latest state-of-the-art non-interactive protocol from standard assumptions is due to Kalai et al. [86], with a protocol runtime of $O(poly(|C|))$.

Table 13.1 compares the Hydra protocol with existing verifiable computation systems. Compared to other state-of-the-art protocols, Hydra's main advantage is a near-constant round complexity from standard assumptions while keeping reasonable prover and verifier time complexity. Groth16 [76] is one of the first protocols widely used in practical secure applications. It has an $O(|C| \log |C|)$ prover complexity and constant verifier complexity and proof sizes. The protocol is non-interactive, so there is only one round. Groth16 requires a per-statement trusted setup. Compared to Groth16, Hydra has a lower prover complexity, and Hydra's trusted setup (namely, the KGZ polynomial commitment setup) does not depend on the circuit as long as the circuit size is bounded. Therefore, Hydra can re-use a trusted setup for different computations. Ligero [3] by Ames et al. does not require a trusted setup. However, Ligero's communication complexity is at the level of the square root of the circuit size. Besides, the verifier's complexity is linear in the circuit size. Compared to Ligero,

Hydra has a lower prover complexity, a lower verifier complexity, a lower round complexity, and smaller proofs. Bulletproofs [40] does not require trusted setups, but it has a linear verification time, whereas Hydra verification is faster. Hyrax [159] requires no trusted setups but incurs higher prover complexity and requires more rounds. libSTARK [15] is the first transparent zero-knowledge proving system with fast verification. Aurora [19] is also transparent but suffers from a linear verifier complexity. Kalai, Paneth, and Yang converted the GKR protocol to a non-interactive verifiable computation scheme and made it publicly verifiable [86]. However, this comes at the cost of higher computational complexity and communication complexity. Compared to Libra [169] by Zhang et al., Hydra has a lower round complexity. Setty [126] presents the first family of transparent zkSNARK constructions with sublinear verifier complexity and an optimal prover complexity. Compared to $\text{Spartan}_{\text{DL}}$, $\text{Spartan}_{\text{RO}}$, and $\text{Spartan}_{\text{CL}}$, Hydra has a better prover complexity. Especially, Hydra has a better proof size compared to $\text{Spartan}_{\text{DL}}$ when the circuit is wide ($d = o(\frac{\sqrt{w}}{\log^2 w})$, where w is the circuit width). Compared to $\text{Spartan}_{\text{KE}}$ with knowledge-type assumptions, Hydra only requires standard assumptions.

Overall, Hydra provides a new option of verifiable computation schemes in the tradeoff between prover time, verifier time, round complexity, proof size, and setups. Particularly, we expect Hydra to run fast when the circuit is deep, which is common for many applications, including verifiable deep learning inference and verifiable database schemes like Litmus.

13.3 Background

13.3.1 Interactive Arguments

We formally define an interactive argument below. Given a function g, a prover $\mathcal{P}$ and a verifier $\mathcal{V}$, an interactive proof allows for $\mathcal{P}$ to convince $\mathcal{V}$ that $g(x) = y$ through a multi-round conversation, where x is an input given by $\mathcal{V}$, and y is the output claimed

[2]It requires an $O(C)$ per-statement trusted setup phase.

[3]The round complexity $O(\log n)$ is constant in the depth of C, logarithmic in the width of C.

	Groth16[2] [22]	Ligero [3]	Bulletproofs [40]
$\mathcal{P}$	$O(C \log C)$	$O(C \log C)$	$O(C)$
$\mathcal{V}$	$O(1)$	$O(C)$	$O(C)$
$\mathcal{R}$	$O(1)$	$O(\sqrt{C})$	$O(\log C)$
$\|\pi\|$	$O(1)$	$O(\sqrt{C})$	$O(\log C)$
Setup	Per-Statement	No	No
	Hyrax [159]	libSTARK [15]	Aurora [19]
$\mathcal{P}$	$O(C \log C)$	$O(C \log^2 C)$	$O(C \log C)$
$\mathcal{V}$	$O(\sqrt{w} + d \log C)$	$O(\log^2 C)$	$O(C)$
$\mathcal{R}$	$O(d \log C)$	$O(\log^2 C)$	$O(\log^2 C)$
$\|\pi\|$	$O(\sqrt{w} + d \log C)$	$O(\log^2 C)$	$O(\log^2 C)$
Setup	No	No	No
	KPY [86]	Libra [169]	**Hydra**[3]
$\mathcal{P}$	$O(\text{poly}(C))$	$O(d \cdot w)$	$O(d \cdot w \log w)$
$\mathcal{V}$	$O((d + w) \cdot \text{polylog}(C))$	$O(d \log w)$	$O(d \log^2 w)$
$\mathcal{R}$	$O(1)$	$O(d \log w)$	$O(\log w)$
$\|\pi\|$	$O(d \cdot \text{polylog}(C))$	$O(d \log w)$	$O(d \log^2 w)$
Setup	No	One-time	One-time

Table 13.1: Comparison of the Hydra protocol to existing state of the art proof systems, specifically the interactive versions of protocols that only rely on standard assumptions (e.g., discrete logarithm, bilinear maps, etc.). Notably, this does not include knowledge type assumptions and the Fiat-Shamir heuristic under the Random Oracle model. The symbols $\mathcal{P}$, $\mathcal{V}$, $\mathcal{R}$, and $|\pi|$ are the prover time, verification time, round complexity, and proof size, respectively. C is the size of a logspace uniform circuit with depth d and width w.

by $\mathcal{P}$. An interactive proof must satisfy two conditions as follows:

- **Completeness.** For every x such that $g(x) = y$, it holds that

$$\Pr[\langle \mathcal{P}, \mathcal{V} \rangle(x) = \mathit{accept}] = 1.$$

  In other words, a prover who follows the protocol honestly will always convince the verifier of the validity of the function evaluation.

- **Soundness.** For any x with $g(x) \neq y$ and any computationally efficient, malicious $\mathcal{P}^*$, it holds that

$$\Pr[\langle \mathcal{P}^*, \mathcal{V} \rangle = \mathit{accept}] \leq \epsilon.$$

  In other words, the verifier will only accept a cheating prover with probability less than or equal to some ϵ. Formally, $\epsilon = \frac{1}{3}$, however, in practical settings this can be made arbitrarily small by repeating the protocol.

When the prover is computationally unbounded, we call the scheme an *interactive proof*.

13.3.2 Sumcheck Protocol

Lund et al. introduced the sumcheck protocol [104], providing an interactive proof for the problem of summing a multivariate polynomial $f : \mathbb{F}^\ell \to \mathbb{F}$ over the Boolean hypercube:

$$\sum_{b^{(1)}, b^{(2)}, \cdots, b^{(\ell)} \in \{0,1\}} f(b^{(1)}, b^{(2)}, \cdots, b^{(\ell)}).$$

The protocol proceeds in ℓ rounds as follows, where in each round, $\mathcal{V}$ samples a random field element $r \in \mathbb{F}$, commonly referred to as a "coin toss".

1. For the first round (denoted as round 1), $\mathcal{P}$ sends the claimed summation value

H and a univariate polynomial

$$f_1(x^{(1)}) := \sum_{b^{(2)},\cdots,b^{(\ell)}\in\{0,1\}} f(x^{(1)}, b^{(2)}, \cdots, b^{(\ell)}).$$

$\mathcal{V}$ checks if $H = f_1(0) + f_1(1)$. Afterwards, $\mathcal{V}$ samples a random field element $r^{(1)} \in \mathbb{F}$ and sends it to $\mathcal{P}$.

2. For every round j where $2 \leq j \leq \ell - 1$, $\mathcal{P}$ sends a univariate polynomial

$$f_j(x^{(j)}) := \sum_{b^{(j+1)},\cdots,b^{(\ell)}\in\{0,1\}} f(r^{(1)}, \cdots, r^{(j-1)}, x^{(j)}, b^{(j+1)}, \cdots, b^{(\ell)}).$$

$\mathcal{V}$ checks if $f_{j-1}(r^{(j-1)}) = f_j(0) + f_j(1)$ and sends another random field element $r^{(j)} \in \mathbb{F}$ to $\mathcal{P}$.

3. For the last round ℓ, $\mathcal{P}$ sends a univariate polynomial

$$f_\ell(x^{(\ell)}) := f(r^{(1)}, r^{(2)}, \cdots, r^{(\ell-1)}, x^{(\ell)}).$$

$\mathcal{V}$ checks if $f_{\ell-1}(r^{(\ell-1)}) = f_\ell(0) + f_\ell(1)$. Then, $\mathcal{V}$ samples another random field element $r^{(\ell)} \in \mathbb{F}$ that is not revealed to $\mathcal{P}$, and evaluates $f(r^{(1)}, r^{(2)}, \cdots, r^{(\ell)})$ on its own or with the help of an oracle. $\mathcal{V}$ accepts if $f_\ell(r^{(\ell)}) = f(r^{(1)}, r^{(2)}, \cdots, r^{(\ell)})$. Otherwise, if any equality check does not hold during the protocol, $\mathcal{V}$ rejects.

The sumcheck protocol is complete and has soundness $\epsilon = \frac{d\ell}{|\mathbb{F}|}$, where d is the total degree of f. The proof size is $O(d\ell)$, and the verifier time is also $O(d\ell)$.

13.3.3 GKR Protocol

As stated before, the GKR protocol [71] is a powerful interactive proof technique that allows for the efficient verifiable evaluation of circuit computations. From a bird's-eye view, the protocol proceeds layer by layer, starting from the output layer and ending at the input layer. For each layer, $\mathcal{P}$ gives a claim about the values in that layer i, namely, the i-th layer counting from the output layer towards the input

layer, and $\mathcal{P}$ reduces it to a claim about the values in the subsequent layer $i+1$, one-layer closer to the input layer, through an instance of the sumcheck protocol on a certain polynomial. Since $\mathcal{V}$ does not have access to the values in intermediate layers (computing those values would require evaluating the circuit, which is precisely what $\mathcal{V}$ wants to avoid) $\mathcal{V}$ cannot check these claims directly. Therefore, this cycle continues until $\mathcal{P}$ gives a claim about the values in the input layer, which $\mathcal{V}$ can check by itself and conclude the protocol.

Notation

Here, we describe the notations throughout the rest of the chapter. Formally, the GKR protocol contains two parties, the prover $\mathcal{P}$ and the verifier $\mathcal{V}$, where $\mathcal{P}$ and $\mathcal{V}$ agree on a layered logspace uniform arithmetic circuit C over a finite field $\mathbb{F}$, of fan-in size 2 and composed of addition and multiplication gates. The objective is for $\mathcal{P}$ to convince $\mathcal{V}$ that the circuit evaluation (the gate values at the output layer) is correctly computed.

Let d be the depth of the circuit, with layer 1 as the output layer and layer d as the input layer. We use S_i to denote the number of gates in the i-th layer, and $s_i := log(S_i)$. Then, we define a function $W_i : \{0,1\}^{s_i} \to \mathbb{F}$ that takes in a binary string $b \in \{0,1\}^{s_i}$, which is the label of a gate at layer i, and $W_i(b)$ returns the value of that gate. Thus, W_1 is the function for the values of the output layer, and W_d is the function for the values of the input layer.

We also define two more functions, add_i and $mult_i : \{0,1\}^{s_i} \times \{0,1\}^{s_{i+1}} \times \{0,1\}^{s_{i+1}} \to \{0,1\}$, known as the *wiring predicate functions*. In essence, these encode how the gates from level i are connected to the wires from level $i+1$. These functions take in a gate label a at level i and two gate labels b and c at level $i+1$. They will return 1 if gate a takes in the values of gates b and c, and the type of gate a is of the corresponding type to the function (addition for add_i, multiplication for $mult_i$). Otherwise, they will return 0.

With this in mind, we provide the expression for W_i as

$$W_i(x) = \sum_{(a,b)\in\{0,1\}^{2s_{i+1}}} [add_i(x,a,b)(W_{i+1}(a) + W_{i+1}(b)) + mult_i(x,a,b)(W_{i+1}(a) \cdot W_{i+1}(b))] \tag{13.1}$$

for $x \in \{0,1\}^{s_i}$. As W_i is expressed as a summation, $\mathcal{P}$ and $\mathcal{V}$ can engage in a sumcheck protocol to verify the validity of a claimed evaluation. We can rewrite the expression as a polynomial in the field $\mathbb{F}$ by use of *multilinear extensions*. Namely, given a function $f : \{0,1\}^\ell \to \mathbb{F}$, the multilinear extension of a function is the unique polynomial $\widetilde{f} : \mathbb{F}^\ell \to \mathbb{F}$ where $\widetilde{f}(x) = f(x), \forall x \in \{0,1\}^\ell$ and the degree of the polynomial in each variable is at most 1. We represent this polynomial by the Lagrange interpolation as

$$\widetilde{f}(x^{(1)}, x^{(2)}, \cdots, x^{(\ell)}) = \sum_{b\in\{0,1\}^\ell} \prod_{i=1}^{\ell} [(1 - x^{(i)})(1 - b^{(i)}) + x^{(i)}b^{(i)}] \cdot f(b),$$

where $x^{(i)}$ is the i-th element of x. Applying this to W_i, we get

$$\widetilde{W_i}(x) = \sum_{(a,b)\in\{0,1\}^{2s_{i+1}}} \left[\widetilde{add_i}(x,a,b)(\widetilde{W}_{i+1}(a) + \widetilde{W}_{i+1}(b)) + \widetilde{mult_i}(x,a,b)(\widetilde{W}_{i+1}(a) \cdot \widetilde{W}_{i+1}(b))\right], \tag{13.2}$$

where $x \in \mathbb{F}^{s_i}$.

The Goldwasser-Kalai-Rothblum Protocol

Let $\mathbb{F}$ be a prime field and C be a layered arithmetic circuit with depth d. The protocol proceeds in d instances sequentially as described below.

1. $\mathcal{P}$ first sends the claimed output of the circuit to $\mathcal{V}$. $\mathcal{V}$ samples a random $x_1 \in \mathbb{F}^{s_1}$ and sends it to $\mathcal{P}$. $\mathcal{P}$ and $\mathcal{V}$ both compute $\widetilde{W}_1(x_1)$. Note that $\widetilde{W}_1(x_1)$ is based on the claimed output values sent by $\mathcal{P}$. The following steps reduces the truthfulness of $\widetilde{W}_1(x_1)$ all the way to the truthfulness of the input layer,

which is known by $\mathcal{V}$.

2. For all layers $1 \leq i \leq d$, $\mathcal{P}$ and $\mathcal{V}$ execute sumcheck over

$$\widetilde{W}_i(x_i) = \sum_{(a,b)\in\{0,1\}^{2s_{i+1}}} \Big[\widetilde{add}_i(x_i,a,b)(\widetilde{W}_{i+1}(a) + \widetilde{W}_{i+1}(b)) + \widetilde{mult}_i(x_i,a,b)(\widetilde{W}_{i+1}(a)\cdot\widetilde{W}_{i+1}(b))\Big]. \tag{13.3}$$

   Note that at the end of the sumcheck (the last round), $\mathcal{V}$ cannot directly check the evaluation of $\widetilde{W}_i$ at a random point through an oracle. $\mathcal{V}$ performs the $\widetilde{add}_i$ and $\widetilde{mult}_i$ calculations by itself. However, $\mathcal{V}$ receives claims from $\mathcal{P}$ about $\widetilde{W}_{i+1}(u_{i+1})$ and $\widetilde{W}_{i+1}(v_{i+1})$ for the final check on that layer, where $u_{i+1}, v_{i+1} \in \mathbb{F}^{s_{i+1}}$ are the random coin tosses of a and b in the sumcheck protocol. Therefore, $\mathcal{V}$ needs to be convinced that those values are correctly provided with sumchecks on layer $i+1$. Observe that a claim is reduced to two more claims on the subsequent layer. If this continues, the number of claims would increase exponentially in terms of the circuit depth d. In order to prevent an exponential blowup in the number of claims, we combine the two claims $\widetilde{W}_{i+1}(u_{i+1})$ and $\widetilde{W}_{i+1}(v_{i+1})$ at each layer into one claim as follows.

   - $\mathcal{V}$ defines a canonical line z such that $z_i(0) = u_{i+1}$ and $z_i(1) = v_{i+1}$ and sends $z_i(x)$ to $\mathcal{P}$.
   - $\mathcal{P}$ sends a degree s_{i+1} univariate polynomial $h_i(x) = \widetilde{W}_{i+1}(z_i(x))$.
   - $\mathcal{V}$ checks if $h_i(0) = \widetilde{W}_{i+1}(u_{i+1})$ and $h_i(1) = \widetilde{W}_{i+1}(v_{i+1})$. $\mathcal{V}$ then samples $r \in \mathbb{F}$ and computes $h_i(r) = \widetilde{W}_{i+1}(z_i(r))$. In this sense, $\mathcal{P}$ and $\mathcal{V}$ can now go to the next level on $x_{i+1} = z_i(r)$.

3. At the input layer d, $\mathcal{V}$ will receive two claims $\widetilde{W}_d(u_d)$ and $\widetilde{W}_d(v_d)$. As $\mathcal{V}$ has complete access to the input layer gates, it can simply calculate the polynomials at those two random locations and see if they are equal to the claimed values. If they are, $\mathcal{V}$ is sufficiently convinced and accepts. Otherwise, if any check has failed, $\mathcal{V}$ rejects.

The GKR protocol is complete and has soundness $O(\frac{d \cdot log|C|}{|\mathbb{F}|})$. For bounded-depth circuits, it is composed of $O(d \cdot log|C|)$ interactive rounds. The verifier time is $O(log|C|)$, and with the improvement by Zhang et al. [169], the prover time is $O(|C|)$.

13.4 Insecure Layer-wise Independent Protocols

Our main target in this chapter is to remove the requirement for a sequential layer-by-layer approach in the GKR protocol. In this section, we first present the case why a naive parallel approach is not secure. Next, we show an attempted fix and explain why it is again insecure.

13.4.1 Naive Method

At a first glance, the GKR protocol seems trivially parallelizable. A naive approach to achieve layer-wise parallelism would be simply to have $\mathcal{V}$ initially choose a list of multi-linear extension points $r_1, r_2, \ldots, r_d$ where $r_i \in \mathbb{F}^{s_i}$ for all $i \in [d]$, where d is the number of layers. Then, $\mathcal{V}$ sends $\{r_i\}$ to the prover $\mathcal{P}$, and $\mathcal{P}$ executes d parallel instances of sumcheck over

$$\widetilde{W}_i(x_i = r_i) = \sum_{(a,b)\in\{0,1\}^{2s_{i+1}}} \Big[\widetilde{add}_i(r_i, a, b)(\widetilde{W}_{i+1}(a) + \widetilde{W}_{i+1}(b)) + \widetilde{mult}_i(r_i, a, b)(\widetilde{W}_{i+1}(a) \cdot \widetilde{W}_{i+1}(b))\Big] \tag{13.4}$$

However, we establish that this is not secure.

The problem with this approach is that the random points are not necessarily correlated such that $\mathcal{V}$ can conclude anything about the two claims $\widetilde{W}_{i+1}(u_{i+1})$ and $\widetilde{W}_{i+1}(v_{i+1})$ given at the last step of the sumcheck protocol for each layer i. Recall that, in the original protocol, u_{i+1} and v_{i+1} were correlated with the help of a line z, and another degree-s_i polynomial h to determine the expected value of the evaluation at a random point on that line. Therefore, the next sumcheck for layer $i+1$ takes place at $W_{i+1}(z(r))$, which is expected to be equal to $h(r)$. In order for the reduction to make

sense, the random point r at the i-th sumcheck has to equal to the starting point of the $(i+1)$-th sumcheck. However, in this case, the soundness of the protocol overall is compromised because the prover knows the evaluation point of the next layer before the sumcheck protocol concludes. Intuitively, $\mathcal{P}$ already knows the random challenges from the verifier before going further through each sumcheck, and the soundness of sumcheck protocols relies on the random challenges being unpredictable.

13.4.2 Correlation Fix

In this section, we make a slight change to the naive parallelization in order to establish a correlation of points because we need to hide the random evaluation point in the adjacent layer. It is important to note that this "fix" is in fact still insecure, and we will present an attack in Section 13.4.4.

Recall that in the original protocol, two claims $\widetilde{W}_{i+1}(u_{i+1})$ and $\widetilde{W}_{i+1}(v_{i+1})$ are reduced to a single claim with the help of a line $z(x)$ such that $z(0) = u_{i+1}$ and $z(1) = v_{i+1}$. $\mathcal{P}$ sends a degree-s_{i+1} univariate polynomial $h(x) = \widetilde{W}_{i+1}(z(x))$, and $\mathcal{V}$ checks whether $h(0) = \widetilde{W}_{i+1}(u_{i+1})$ and $h(1) = \widetilde{W}_{i+1}(v_{i+1})$. $\mathcal{V}$ then samples $r \in \mathbb{F}$ and computes $h(r) = \widetilde{W}_{i+1}(z(r))$. $\mathcal{P}$ and $\mathcal{V}$ finally go to the next level on the point $x_{i+1} = z(r)$.

To maintain correlation of points, we change random choices of the verifier. Instead of choosing the evaluation points of all the layers randomly at the beginning, the verifier chooses two random points u_{i+1} and v_{i+1} to form a random line $z(x)$ for each layer. During the online phase of the protocol, $\mathcal{V}$ can designate the random point to evaluate on by sampling $r \in \mathbb{F}$ and requesting the evaluation point at $z(r)$ such that $\mathcal{P}$ gives a claimed value for $\widetilde{W}_i(z(r))$ at each layer i, which is reduced to claims for $\widetilde{W}_i(u_{i+1})$ and $\widetilde{W}_i(v_{i+1})$ via sumcheck. Hence, when the end of the sumcheck is reached, $\mathcal{V}$ can request $h(x)$ (the prover's claimed polynomial) and evaluate it at r to check for consistency in the claim $\widetilde{W}_{i+1}(z(r))$ for the subsequent layer.

13.4.3 Technical Description

Let $\mathbb{F}$ be a prime field and C be a layered arithmetic circuit with depth d, with circuit layers labeled from the output layer 1 to the input layer d. S_i denotes the number of gates in the i-th layer, and $s_i := log(S_i)$. The protocol proceeds as follows.

Protocol: Naive Parallelized GKR (Correlation Fix)

1. $\mathcal{V}$ randomly samples points $u_{i+1}, v_{i+1} \in \mathbb{F}^{s_{i+1}}$ and line $z_i(x)$ such that $z_i(0) = u_{i+1}$ and $z_i(1) = v_{i+1}$ for all $1 \leq i < d$.

2. For all $1 \leq i < d$, $\mathcal{V}$ randomly samples point $r_i \in \mathbb{F}$. $\mathcal{P}$ and $\mathcal{V}$ then execute sumcheck over

$$\widetilde{W}_i(x_i = z_i(r_i)) = \sum_{(a,b)\in\{0,1\}^{2s_{i+1}}} \Big[\widetilde{add}_i(z_i(r_i), a, b)(\widetilde{W}_{i+1}(a) + \widetilde{W}_{i+1}(b)) + \widetilde{mult}_i(z_i(r_i), a, b)(\widetilde{W}_{i+1}(a) \cdot \widetilde{W}_{i+1}(b))\Big] \tag{13.5}$$

   where x_i is the input to the sumcheck. The first half of the sumcheck's coin tosses will be set to u_{i+1}, and the second half of the sumcheck's coin tosses will be set to v_{i+1}.

3. For all $1 \leq i < d$, $\mathcal{P}$ sends $\mathcal{V}$ the degree-s_{i+1} univariate polynomial $h_i(x) = \widetilde{W}_{i+1}(z_i(x))$. $\mathcal{V}$ checks that $h(0) = \widetilde{W}_{i+1}(u_{i+1})$ and $h(1) = \widetilde{W}_{i+1}(v_{i+1})$.

4. $\mathcal{V}$ checks if the claim from the sumcheck in the subsequent layer $\widetilde{W}_{i+1}(z_i(r_{i+1}))$ is equal to $h_i(z_i(r_{i+1}))$. If it is equivalent across all $1 \leq i < d$ and the claims for the input layer d are independently verified to be correct, $\mathcal{V}$ accepts. Otherwise, if any check has failed, $\mathcal{V}$ rejects.

13.4.4 Attack

We demonstrate a simple attack method that shows the insecurity of a traditional layer-wise independent GKR protocol, even when the points are correlated. Our

attack stems from the fact that, unlike the traditional protocol, the evaluation point r can be deduced by $\mathcal{P}$ before sending h to $\mathcal{V}$. Notice that after u_{i+1} and v_{i+1} are revealed to $\mathcal{P}$, it can deduce the line $z(x)$, and subsequently determine at which point r the verifier $\mathcal{V}$ requested the evaluation of $\widetilde{W}_{i+1}(z(r))$, because $\mathcal{P}$ knows $z(r)$. Once a malicious $\mathcal{P}^*$ knows r, it can craft the degree s_{i+1} polynomial $h(x)$ such that it agrees with $\mathcal{V}$'s checks for $h(0) = \widetilde{W}_{i+1}(u_{i+1})$ and $h(1) = \widetilde{W}_{i+1}(v_{i+1})$, but pegging $h(r)$ to be an incorrect $\widetilde{W}^*_{i+1}(z(r)) \neq \widetilde{W}_{i+1}(z(r))$. Thus, when $\mathcal{V}$ checks for consistency, $h(r)$ is already compromised.

13.5 Subcircuit Protocol

In this section, we propose a method to introduce some degree of depth-wise parallelism while circumventing the security issues previously shown with a naive parallelized GKR protocol.

Using pipelining, we split up a circuit C depth-wise by treating C as k subcircuits (denoted as c_i for all $1 \leq i \leq k$), where the input of subcircuit c_i directly corresponds with the output of subcircuit c_{i-1}. In other words, for all $i > 1$, the input of a subcircuit c_i is precisely the output of the subcircuit c_{i-1} above it. Then, the original GKR protocol can be executed on each subcircuit concurrently, maintaining the security guarantee of GKR with the scope of each subcircuit. The only remaining need to guarantee full security of the protocol in regard to the entire circuit C would be proving the relationship between the output and input of adjacent subcircuits.

Recall that, in the GKR protocol, the prover generates claims about the multilinear polynomial $\widetilde{W}_i(x)$. For any two adjacent subcircuits, Let $\widetilde{W}_{out}(r)$ be the claim regarding the gate values of the output level of the first circuit, and $\widetilde{W}_{in}(r)$ be the claim regarding the gate values of the input level of the second circuit. As the claims should effectively be the same, $\mathcal{P}$ wants to convince $\mathcal{V}$ that $\widetilde{W}_{out}(r) = \widetilde{W}_{in}(r)$.

13.5.1 Naive Approach

Naively, $\mathcal{P}$ and $\mathcal{V}$ can simply iterate over the gate values at each input and output level, explicitly providing the guarantee that they are in fact equal. In this case, all of the gate values that comprise those specific levels would be revealed, and $\mathcal{V}$ would perform a check for the $\widetilde{W}_i(r)$ values claimed by $\mathcal{P}$ for all random points sampled $r \in \mathbb{F}^{s_i}$.

However, by revealing the plain points of those layers, $\mathcal{V}$ incurs a huge overhead in terms of computation and communication costs. Now, instead of receiving and sending only s_i points and evaluations, $\mathcal{P}$ has to give $\mathcal{V}$ all 2^{s_i} gate values, which is an exponential increase that we want to avoid.

13.5.2 Polynomial Commitment Scheme

A better approach would be to utilize cryptography, specifically *polynomial commitments*, which are crucial in succinct arguments in that they force $\mathcal{P}$ to answer the queries by $\mathcal{V}$ such that they must be in accordance with a specific bounded-degree polynomial. In essence, these polynomial commitments start with $\mathcal{P}$ sending a value $s \in \mathbb{F}$ which is the claimed value of an $f(z)$ polynomial evaluation where $\mathcal{V}$ knows z. Then, $\mathcal{P}$ sends a corresponding opening proof π that the evaluation is indeed correct.

More specifically, we use the efficient constant-size polynomial commitment scheme described in [49] and let $\mathcal{P}$ commit to the polynomials $\widetilde{W}_{in}(x)$ and $\widetilde{W}_{out}(x)$ (which should be the exact same) for every pair of adjacent subcircuits. This way, the gap in security between the subcircuits is effectively bridged, and the verification process can be efficiently pipelined across the subcircuits.

13.5.3 Technical Description

Let $\mathbb{F}$ be a prime field and C be a layered arithmetic circuit with depth d, with circuit layers labeled from the output layer 1 to the input layer d. C is split into k equal-depth subcircuits denoted as c_i for all $1 \leq i \leq k$. The input of subcircuit c_i directly corresponds with the output of subcircuit c_{i-1}. For each subcircuit c_i, let $\widetilde{W}_{in_i}$

denote the multilinear extension of the input layer, and $\widetilde{W}_{out_i}$ denote the multilinear extension of the output layer. Figure 13-2 depicts the protocol, which proceeds as follows.

Protocol: Subcircuit (Figure 13-2)

1. For all $1 \leq i \leq k$, $\mathcal{P}$ commits (polynomial commitment) to $\widetilde{W}_{in_i}$ and $\widetilde{W}_{out_i}$ in subcircuit c_i.
2. $\mathcal{P}$ and $\mathcal{V}$ engage in batched traditional GKR protocol for all k subcircuits.
3. For all $1 \leq i \leq k$, $\mathcal{P}$ opens the proofs at all randomly sampled locations for the evaluations of $\widetilde{W}_{in_i}$ and $\widetilde{W}_{out_i}$ during the batched traditional GKR protocol and sends these proofs to $\mathcal{V}$.
4. $\mathcal{V}$ checks the proofs to verify that $\widetilde{W}_{in_i}(x) = \widetilde{W}_{out_i}(x)$ at all randomly sampled points for all adjacent subcircuits.

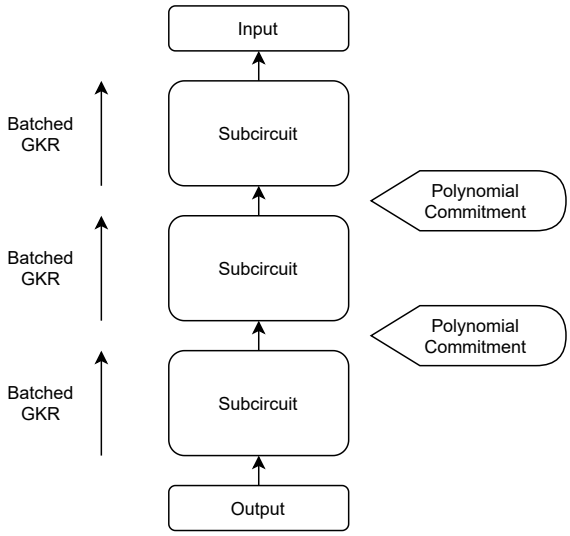


Figure 13-2: Subcircuit Protocol

13.5.4 Trade-off

We recognize that the total number of layers in the representation of C increases by an additive factor of k, as in each adjoining subcircuit the input and output are repeated.

However, in the perspective of the proposed subcircuits, we essentially slash the depth and number of rounds by a *multiplicative* factor of k. This has great appeal for $\mathcal{P}$ and $\mathcal{V}$ assuming they have access to multithreaded hardware and can concurrently run the protocol on the separate subcircuits. In addition, assuming the subcircuits are all of the same depth and the layers are logspace uniform (s_i is equivalent across all i), the number of rounds can decrease by a factor of k as well. $\mathcal{V}$ can use the same random coin tosses for sumchecks across all the subcircuits simultaneously.

13.6 Hydra Protocol

In this section we present a GKR-based protocol with full layer-wise independence. Although the subcircuit protocol presented previously does allow for splitting a deep circuit up into multiple batched subcircuits, it does not allow for us to completely parallelize the proof across all of the layers in the circuit. Intuitively, this is because when we split the circuit up, the internal subcircuit itself still has to engage in an original sequential GKR protocol. Here, we propose an interactive protocol with a series of $O(\log n)$ rounds. Recall that n is the circuit width. We run $d \log n$ sumchecks concurrently. Despite the increase in number of sumchecks, the advantage of this protocol is the reduction in round complexity from the traditional $O(d \log n)$ rounds. Compared to the subcircuit protocol, the Hydra protocol does not require the duplication of adjacent layers. More importantly, the round complexity is no longer dependent on the depth, so it does not degrade based on the number of layers in the circuit. Notably, Hydra does not depend on any non-standard assumptions (knowledge type, Fiat-Shamir/Random Oracle). Instead, Hydra only relies on the standard cryptographic assumptions presented in [49].

13.6.1 Context

For each layer i of the original GKR protocol, denote the evaluation point as x_i. The claim of $\widetilde{W}_i(x_i)$ is reduced into claims of $\widetilde{W}_{i+1}(a_i)$ and $\widetilde{W}_{i+1}(b_i)$, where $x_i \in \mathbb{F}^{s_i}$ and

$a_i, b_i \in \mathbb{F}^{s_{i+1}}$. More specifically, the polynomial evaluated

$$\widetilde{W}_i(x_i) = \sum_{(a,b)\in\{0,1\}^{2s_{i+1}}} \Big[\widetilde{add}_i(x_i,a,b)(\widetilde{W}_{i+1}(a) + \widetilde{W}_{i+1}(b)) + \widetilde{mult}_i(x_i,a,b)(\widetilde{W}_{i+1}(a) \cdot \widetilde{W}_{i+1}(b))\Big] \tag{13.6}$$

over the sumcheck ultimately results in two claimed values for $\widetilde{W}_{i+1}$ at the locations a_i and b_i determined by the random points sampled by $\mathcal{V}$ during the protocol. During the sumcheck, each coin toss for a and b determines an additional dimension for a_i and b_i, respectively.

At the beginning of each sumcheck, $\mathcal{P}$ claims the value for $\widetilde{W}_i(x_i)$. We call x_i a *query point*, and $\widetilde{W}_i(x_i)$ a *query point evaluation*. At the end of each sumcheck, $\mathcal{P}$ claims the values for $\widetilde{W}_{i+1}(a_i)$ and $\widetilde{W}_{i+1}(b_i)$. We call a_i and b_i *challenge points*, and $\widetilde{W}_{i+1}(a_i)$ and $\widetilde{W}_{i+1}(b_i)$ *challenge points evaluations*.

The concept of layer-wise independence ultimately boils down to proving that the claimed challenge point evaluations ($\widetilde{W}_{i+1}(a_i)$ and $\widetilde{W}_{i+1}(b_i)$) given at the last step of the sumcheck at layer i is consistent with the claimed query point evaluation ($\widetilde{W}_{i+1}(x_{i+1})$) given at the beginning of the sumcheck run at layer $i+1$. Note that a_i, b_i, and x_{i+1} should be uniformly random and uncorrelated in the perspective of $\mathcal{P}$. This is important because all of the query points are revealed at once to $\mathcal{P}$. If a_i, b_i can be deduced given x_{i+1}, the soundness of sumcheck will be compromised.

Thus, the core of our protocol lies in the power of polynomial interpolation. If we let $\mathcal{V}$ sample m query points ($\boldsymbol{x}_{i+1}^{(1)}$, $\boldsymbol{x}_{i+1}^{(2)}$, ..., $\boldsymbol{x}_{i+1}^{(m)}$) for layer $i+1$, where $\boldsymbol{x}_{i+1}^{(j)} \in \mathbb{F}^{s_{i+1}}$ $\forall j \in [m]$, it is simple to see how the evaluation claims for these values could be obtained through m instances of the sumcheck protocol at layer $i+1$. Then, $\mathcal{V}$ can interpolate using points in the form of $\big(\boldsymbol{x}_{i+1}^{(j)}, \widetilde{W}_{i+1}(\boldsymbol{x}_{i+1}^{(j)})\big)$ for $1 \le j \le m$ to obtain the unique polynomial $\widetilde{W}_{i+1}$ assuming m is large enough. Once $\widetilde{W}_{i+1}$ is established, $\mathcal{V}$ can use it to verify against the claimed challenge points evaluations $\widetilde{W}_{i+1}(a_i)$ and $\widetilde{W}_{i+1}(b_i)$ given at the last step of sumcheck at layer i. Given a set of $n+1$ points in the form of $(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)})$ where $\boldsymbol{x}^{(i)}$ is unique across all points, polynomial interpolation

defines a linear bijection such that there exists a unique polynomial p of total degree at most n that agrees with all of the points.

If the query points are sampled from the full space of $\mathbb{F}^{s_{i+1}}$, $\mathcal{V}$ would need $2^{s_{i+1}}+1$ claimed evaluation points in order to interpolate $\widetilde{W}_{i+1}(x)$. Note that $\widetilde{W}$ is multilinear, therefore, the total degree is $2^{s_{i+1}}$. Unfortunately, similar to the plain gate reveal in the naive method for the subcircuit protocol, this is an exponential increase in claims that we want to avoid.

13.6.2 Subspace Reduction

Here, we reduce the number of points needed for the interpolation $\widetilde{W}_{i+1}(x_{i+1})$ by restricting the query and challenge points over a *subspace* of $\mathbb{F}^{s_{i+1}}$. This subspace is generated by the output domain of a vector polynomial $f : \mathbb{F} \to \mathbb{F}^{s_{i+1}}$ (not to be confused with the polynomial f presented in the sumcheck background) sampled by $\mathcal{V}$. The coefficients of f are hidden from $\mathcal{P}$. Evaluations and claims all take place over $\widetilde{W}_{i+1}(f(\mu))$ at random points $\mu \in \mathbb{F}$. More precisely, given a multilinear function f and three lists of randomly sampled field elements $\mu_j, \phi_j, \theta_j \in \mathbb{F}$, where $j \in [m]$, $\mathcal{P}$ and $\mathcal{V}$ run sumchecks of the form

$$\widetilde{W}_i(x_i = f(\mu_j)) = \sum_{(a,b)\in\{0,1\}^{2s_{i+1}}} \Big[\widetilde{add}_i(f(\mu_j), a, b)(\widetilde{W}_{i+1}(a) + \widetilde{W}_{i+1}(b)) + \widetilde{mult}_i(f(\mu_j), a, b)(\widetilde{W}_{i+1}(a) \cdot \widetilde{W}_{i+1}(b))\Big]. \quad (13.7)$$

During the sumcheck protocol, the first half of $\mathcal{V}$'s s_{i+1} random coin tosses will be set equal to $f(\theta_j)$, and the second half of $\mathcal{V}$'s s_{i+1} random coin tosses will be set equal to $f(\phi_j)$ for all j. Thus, at the end of a single sumcheck, one claimed query point evaluation, namely, $\widetilde{W}_i(f(\mu_j))$, is reduced to two claimed challenge points' evaluations on $\widetilde{W}_{i+1}$, namely, $\widetilde{W}_{i+1}(f(\phi_j))$ and $\widetilde{W}_{i+1}(f(\theta_j))$. We set $m \leftarrow deg(f) \cdot s_i + 1$ and run $deg(f) \cdot s_i + 1$ sumchecks per layer, such that for each layer i, at the end of all sumchecks $\mathcal{V}$ has $deg(f) \cdot s_i + 1$ claimed query point evaluations and receives $2(deg(f) \cdot s_i + 1)$ claimed challenge point evaluations. Then, $\mathcal{V}$ can interpolate the

query point evaluations to obtain the new polynomial $g_i(x) := \widetilde{W_i}(f(x))$. $\mathcal{V}$ can check that $g_i(x)$ is in fact a degree-$(deg(f) \cdot s_i)$ polynomial, and subsequently verify it on the claimed challenge point evaluations. Because for any $k \in \mathbb{Z}^+$, no two distinct degree-k polynomials can agree on more than k points, this setup protects from a dishonest $\mathcal{P}$ by the Schwartz-Zippel lemma [125].

13.6.3 Maintaining Sumcheck Soundness

In addition, we need to make sure that $\mathcal{P}$ cannot predict any challenge points. If $\mathcal{P}$ determines f or learns the value $f(x)$ before the entire vector is revealed, it can ruin the soundness of sumcheck. Recall that for the sumcheck protocol, $f(\mu_j)$ is revealed to $\mathcal{P}$ as it is the randomly sampled vector that $\widetilde{W_i}$ is to be evaluated at (the input of the sumcheck). During the sumcheck protocol, the two locations of claims ($f(\phi_j)$ and $f(\theta_j)$) on $\widetilde{W_{i+1}}$ are the random coin, which are also eventually revealed. Thus, if f is known to $\mathcal{P}$, after the first random coin toss (the first field element of an $f(x)$ evaluation) is sent, $\mathcal{P}$ can determine the composition of the rest of the vector, which corresponds to the following $s_i - 1$ coin tosses. As the soundness of sumcheck relies on the fact that subsequent coin toss challenges are not known to $\mathcal{P}$, the security is compromised. We assert that for any polynomial $f : \mathbb{F} \to \mathbb{F}^{s_i}$, the number of points to determine $g_i(x) = \widetilde{W_i}(f(x))$ by interpolation is $deg(f) \cdot s_i + 1$. However, f and the points at which f is evaluated must not be revealed simultaneously to preserve security. If f is revealed, the evaluation point x can be easily determined by $\mathcal{P}$. In addition, as $\mathcal{V}$ reveals more than $deg(f)$ evaluations of $f(x)$, the points x's have to be hidden because f can be interpolated by $\mathcal{P}$ itself when given more than $deg(f)$ point-evaluation pairs. We present the full soundness proof later in Section C.2.

The benefit of the subspace reduction method is twofold. First, the number of claims needed is no longer exponential. In this case, the number of concurrent sumcheck protocols (claims needed per layer) is proportional to s_i, which is ideal for even extremely wide circuits. The degree of the subspace generating polynomial f can subsequently be seen as a security parameter, where a higher degree will require more points of evaluation in order for $\mathcal{V}$ to interpolate the function and be fully convinced.

Second, as the coefficients of f are not known to $\mathcal{P}$ and f and x are hidden, from the perspective of $\mathcal{P}$, the evaluation points are completely random and not correlated. Thus, $\mathcal{P}$ cannot predict ahead of time the random coin tosses of any sumcheck, nor the challenge points for another layer before they are revealed by $\mathcal{V}$.

13.6.4 Malicious Interpolation Points

Finally, we note that the query point evaluations for interpolation, which come from the polynomial $\widetilde{W}_{i+1}(f(x))$, need additional attention. As the initial claims are exactly what $\mathcal{V}$ will be checking upon when $\mathcal{V}$ interpolates, a malicious $\mathcal{P}$ could be able to choose an incorrect $\widetilde{W}_i^*$ such that it interpolates to the correct values *only in the given subspace of the protocol*. In other words, $\mathcal{P}$ can proceed with an incorrect polynomial that in the scope of $\mathcal{V}$ is completely consistent with the correct one, and $\mathcal{P}$ fully cooperate with $\mathcal{V}$ throughout all sumchecks. After the sumchecks are completed, the values $\mathcal{V}$ interpolates will be correct only in the scope of those claimed points. We present a formalized attack below.

1. Within any layer i in the circuit, $\mathcal{P}$ chooses an incorrect polynomial $\widetilde{W}_i^*$ constructed such that it agrees with the correct $\widetilde{W}_{i+1}(f(x))$ in the subspace provided. Namely, for each layer i, the specific subspace, which the incorrect polynomial $\widetilde{W}_i^*$ agrees with $\widetilde{W}_{i+1}(f(x))$ upon, is composed of the $f(x)$ evaluations given to $\mathcal{P}$ on $\widetilde{W}_{i+1}(f(x))$.

2. For each sumcheck, $\mathcal{P}$ engages in a completely honest interaction with $\mathcal{V}$, answering correctly according to the polynomial $\widetilde{W}_i^*$ for all of $\mathcal{V}$'s challenges.

3. $\mathcal{V}$ interpolates $\widetilde{W}_i^*$. The malicious polynomial bypasses this interpolation check. In the scope of the given claims, the interpolation shows complete consistency between adjacent layers.

Therefore, this attack will inductively lead to an incorrect claimed output that will be verified as correct by $\mathcal{V}$.

The reason why this is a significant vulnerability is that, at the end of each sumcheck for a layer, $\mathcal{P}$ cannot give $\mathcal{V}$ an oracle access to $\widetilde{W}_{i+1}(x)$ in order to verify the claims presented. In the original GKR protocol, the entire protocol on the subsequent layers $\geq i+1$ essentially serves as the oracle for the current sumcheck on layer i.

In order to prevent this scenario, the protocol must guarantee that $\widetilde{W}_i$ is not malicious by having $\mathcal{P}$ engage in a polynomial commitment to the polynomial, similar to the subcircuit protocol. In such an environment, an adversarial $\mathcal{P}$ will not be able to cheat with an incorrect $\widetilde{W}_i$ because the polynomial commitment re-enables the oracle. $\mathcal{P}$ is forced to bind to the polynomial before it knows the points that $\widetilde{W}_{i+1}$ will be evaluated on.

13.6.5 Technical Description

Let $\mathbb{F}$ be a prime field and C be a layered arithmetic circuit with depth d. Recall that we use S_i to denote the number of gates in the i-th layer, and $s_i := log(S_i)$. For the sake of simplicity, assume all of the layers obey the same logspace and let all of the individual functions in the dimension of f be linear. Figure 13-3 depicts the protocol, which proceeds as follows.

Protocol: Hydra (Figure 13-3)

1. Let $f : \mathbb{F} \to \mathbb{F}^{s_{i+1}}$ be a random polynomial sampled by $\mathcal{V}$ for all $1 \leq i \leq d$.
2. For each $1 \leq j \leq s_i \cdot deg(f) + 1$, $\mathcal{V}$ samples random $\mu_j, \phi_j, \theta_j \in \mathbb{F}$. Let

$$r_{i,j} = f(\mu_j), a_{i,j} = f(\phi_j), b_{i,j} = f(\theta_j)$$

   for all $1 \leq i \leq d$.
3. For each $1 \leq i \leq d$, $\mathcal{P}$ commits (via polynomial commitments) to $\widetilde{W}_i(x)$.
4. For $1 \leq i \leq d$ and $1 \leq j \leq s_i \cdot deg(f) + 1$, $\mathcal{P}$ and $\mathcal{V}$ concurrently execute

sumchecks over

$$\widetilde{W}_i(x_i = r_{i,j}) = \sum_{(a,b)\in\{0,1\}^{2s_{i+1}}} \Big[\widetilde{add}_i(r_{i,j}, a, b)(\widetilde{W}_{i+1}(a) + \widetilde{W}_{i+1}(b)) + \widetilde{mult}_i(r_{i,j}, a, b)(\widetilde{W}_{i+1}(a) \cdot \widetilde{W}_{i+1}(b))\Big] \quad (13.8)$$

When the sumcheck is run, the first half of $\mathcal{V}$'s s_{i+1} random coin tosses will be set to $a_{i,j}$, and the second half of $\mathcal{V}$'s s_{i+1} random coin tosses will be set to $b_{i,j}$. At the beginning of sumcheck, $\mathcal{P}$ claims the value for $\widetilde{W}_i(r_{i,j}) = \widetilde{W}_i(f(\mu_j))$. At the end of each sumcheck at layer i, $\mathcal{P}$ provides claimed values for $\widetilde{W}_{i+1}(a_{i,j}) = \widetilde{W}_{i+1}(f(\phi_j))$ and $\widetilde{W}_{i+1}(b_{i,j}) = \widetilde{W}_{i+1}(f(\theta_j))$. We call the beginning claims for $\widetilde{W}_i(r_{i,j})$ across $s_i \cdot deg(f) + 1$ points as the *query point evaluations* and the ending claims for $\widetilde{W}_{i+1}(b_{i,j})$ at a total of $2(s_{i+1} \cdot deg(f) + 1)$ points as the *challenge point evaluations*.

5. For each $1 \leq i \leq d$ and $1 \leq j \leq s_i \cdot deg(f) + 1$, $\mathcal{P}$ opens and sends the polynomial commitment proofs of $\widetilde{W}_i(x)$ for all points $r_{i,j}$.

6. For all layers $1 < i < d$, $\mathcal{V}$ uses the $s_i \cdot deg(f) + 1$ query point evaluations from layer i to interpolate the unique polynomial $g_i(x) := \widetilde{W}_i(f(x))$.$\mathcal{V}$ then checks to see if $g_i(x)$ is indeed a degree $s_i \cdot deg(f)$ polynomial, and verifies consistency by evaluating $g_i(x)$ at the $2(s_i \cdot deg(f) + 1)$ challenge point evaluations reduced from layer $i - 1$, accepting only if $g_i(\phi_j) = \widetilde{W}_i(a_{i-1,j})$ and $g_i(\theta_j) = \widetilde{W}_i(b_{i-1,j})$ for all ϕ_j, θ_j points. As $\mathcal{V}$ already has access to the input layer d and the output layer 1, it can check the $\widetilde{W}_d$ and $\widetilde{W}_1$ claims on its own.

13.7 Practical Considerations

This section presents proposals on the practical aspects of optimization we utilize in our implementation of the verifiable computation system.

For context behind the practical usage of our protocols, note that the arithmetic

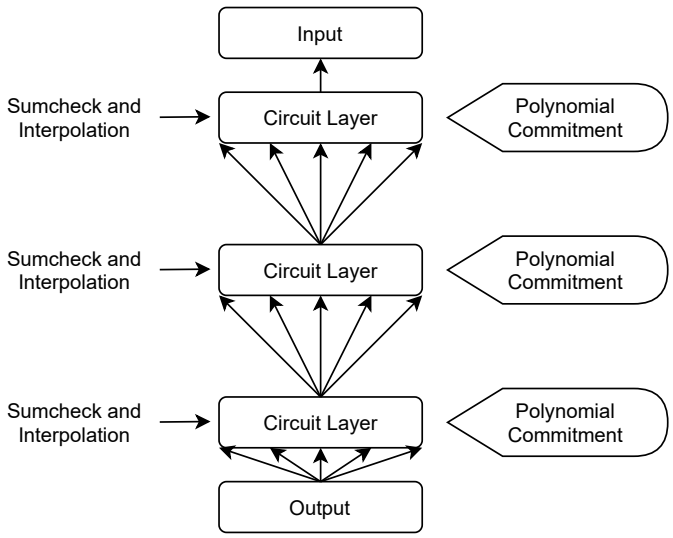


Figure 13-3: Hydra Protocol

circuit structure used in the protocol is computationally complete. At a high level, this means that any computation can be represented with an arithmetic circuit. Hence, given code for a computation, we can translate this to an arithmetic circuit representation and then delegate the computation with the use of our protocols.

13.7.1 Circuit Representation

When dealing with practical usage of circuit evaluation proofs, we cannot assume that the circuit is sufficiently "regular" to conform with the original GKR protocol specifications. In the real world, circuits often have wires connecting non-adjacent layers. The computations are often compiled into arbitrary direct acyclic graphs. Therefore, we discuss the transformations needed in order to proceed with the proving process.

Pass-through Gates

A naive approach is to relay the values of a cross-layer wire through the layers in between. Namely, we need pass-through gates. Conforming to the original GKR arithmetic circuit with only addition and multiplication gates, we can realize a pass-through gate by having a dedicated wire of value 0 and use addition gates to add 0

to the target wire value.

To make it clean, we propose to add a specific "pass-through" gate of fan-in size 1. The sole purpose of this gate would be to pass the value of a gate at a level i to level $i+1$. This new gate can be rather conveniently expressed as an additional term in the polynomial being evaluated across the GKR protocol:

$$\widetilde{W}_i(x) = \sum_{(a,b)\in\{0,1\}^{2s_{i+1}}} \Big[\widetilde{op}_i(x,a,b)(op(\widetilde{W}_{i+1}(a),\widetilde{W}_{i+1}(b)) + \cdots + \widetilde{passthru}_i(x)(\widetilde{W}_{i+1}(x))\Big] \tag{13.9}$$

where op is an arithmetic operation and $\widetilde{op}$ can be viewed as the multilinear extension of the Boolean function relating to said operation.

13.7.2 NAND Parser

Here, we note that the construction and complexity of the $\widetilde{W}_i(x)$ depends solely on the number of distinct gates in the arithmetic circuit. As more types of gates are added, $\mathcal{V}$ needs to undergo more work to perform its final sumcheck verification, and $\mathcal{P}$ also is subject to a similar computation cost when calculating the polynomial evaluations. While in the original GKR protocol, there are only addition and multiplication gates. Existing practical circuit compilers [111] translate high-level programming languages into Boolean logic circuits with a multitude ($2^{2^2} = 16$) of gate types. This does not pose a theoretical problem in our situation, as we easily can find mappings for such systems. For example, $A \wedge B \implies A \cdot B$, $A \barwedge B \implies A + B$, and so forth for the rest of the gates (note that the logical circuit representation is under a Galois field of size 2).

However, this does pose a practical issue as the number of multilinear extensions (one for each type of gate) that have to be computed by both $\mathcal{P}$ and $\mathcal{V}$ have now grown. To handle the increase in the type of gates, we can exploit the functional completeness of the NAND gate. By creating a parser to convert all types of logical gates into chains of NAND gates represented as $A + B$, we can efficiently handle

logic circuits. This way, all computation can be represented in a simplified structure with less cost of verification for $\mathcal{V}$, as it will now only have to check against one $\widetilde{nand}$ operation in the $\widetilde{W}_i(x)$ polynomial. In addition, the parser also handles the pass-through gate creation in the case of a DAG logic circuit given as input.

13.7.3 Engineering the Pipeline

In this section, we briefly discuss the engineering process and benefits of our protocols, specifically relating to the massive pipelining of the verification process.

Parallelization of Subcircuits and Layers

With regards to subcircuits in the subcircuit protocol and layers in the Hydra protocol, the widespread parallelization of these components will lead to not only reduced round complexity, but also faster protocols in general. This effectively reduces the computational overhead for the circuit depth, with deeper circuits subject to more benefit with our protocols.

Streaming Upload

Another scope of the pipeline is the use of a streaming upload. In other words, the client can upload the circuit to the cloud in chunks such that we can take advantage of our subcircuit protocol. When doing so, the proving process can begin when the first circuit chunk is ready, before the upload process finishes. The polynomial commitments prevent $\mathcal{P}$ from cheating in connecting layers, and the soundness of the GKR protocol itself guarantees the rest of the security.

13.8 Experimental Evaluation

13.8.1 Environment Setup

We use the Frigate [111] compiler to convert C-style code to logical circuits, and our parser to convert the circuits into provable representations for both the subcircuit

protocol and the Hydra protocol. We implemented our proposals in around 1,500 lines total of C++ code with the polynomial and arithmetic logic based on Thaler's implementation of [54]. Finally, we used the PolyCommit Rust library [5] from Marlin [49] for the multilinear extension composition-based polynomial commitments. Our experiments were executed on 40 physical Intel Xeon CPU E7-4850 cores with hyperthreading (80 virtual cores) and 128 GB of RAM.

13.8.2 Results and Discussion

In this section, we present the results of our experiments, along with discussion and analysis. All experiments described in the following two subsections were run on randomly generated layered circuit structure and random input values which obeyed certain depth and width properties. We compared our protocols against an implementation of the GKR protocol. For the subcircuit protocol, we compare it with the GKR protocol proving the entire circuit. For the Hydra protocol, we benchmark separately using the original GKR protocol under the same conditions. Both the subcircuit protocol and the Hydra protocol were also tested with a circuit of *verifiable delay function* (VDF) [32]. In simple terms, a VDF is a sequential step-by-step evaluation that produces a specific output. In particular, we choose to iterate the SHA-256 cryptographic hash function using the efficient Boolean SHA-256 logic circuit from [45].

Subcircuit Protocol

For the subcircuit protocol (Figure 13-4), we evaluated multiple different circuit compositions, and tested various subcircuit depths and amounts. We mainly focused on testing long, skinny circuits that are known to be inefficient to prove with the traditional GKR approach. The tests were evaluated at two main circuit depths, $2^{16} = 65536$ and $2^{20} = 1048576$, as well as two main circuit widths, $2^7 = 128$ and $2^8 = 256$, We split the circuit into evenly distributed subcircuits ranging from $2^0 = 1$ subcircuit (serving as the control, where the entire circuit is treated as one large

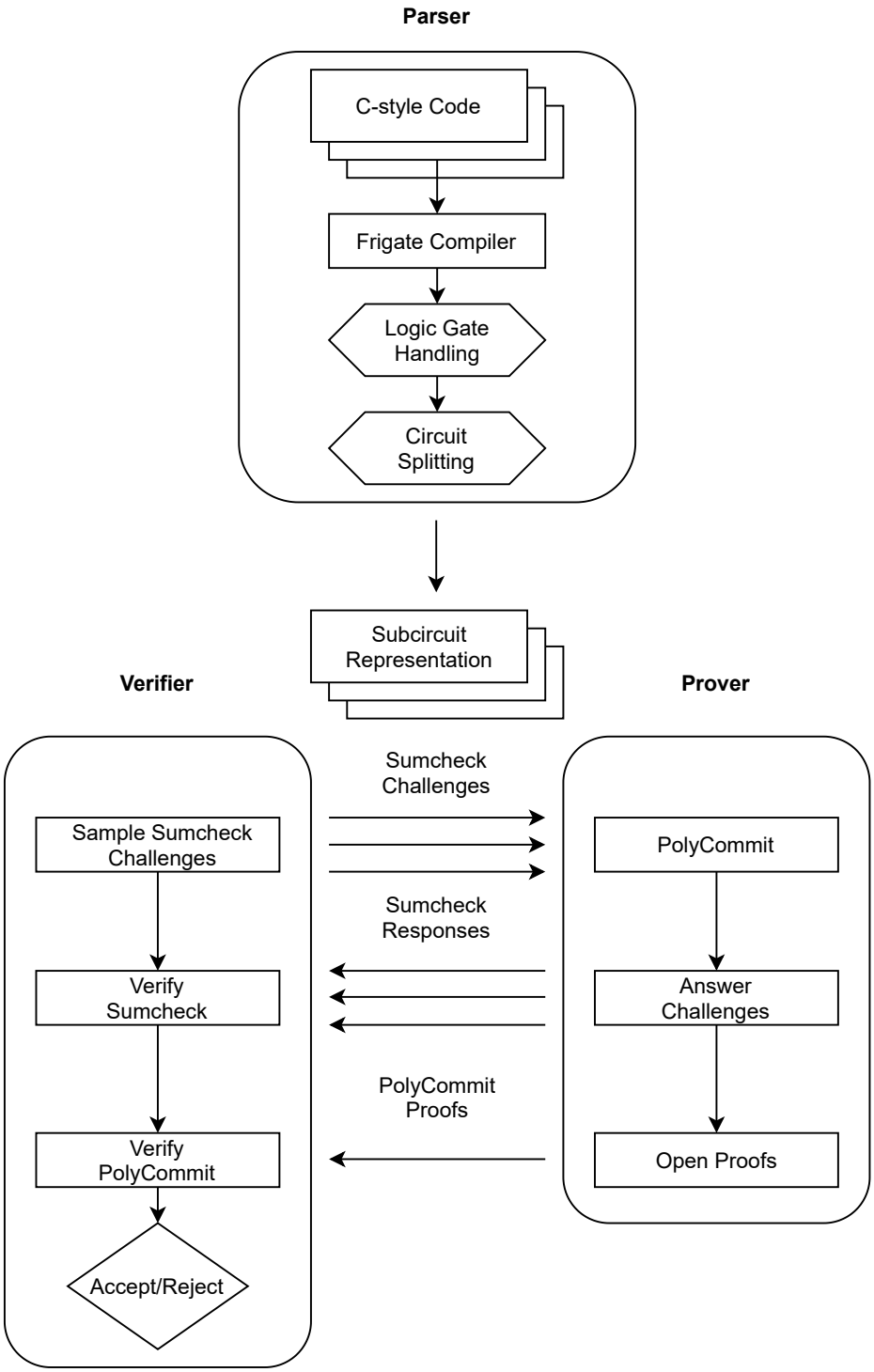


Figure 13-4: Subcircuit Environment.

subcircuit) up to $2^8 = 256$ subcircuits and prove them in parallel.

Figure 13-5 and Figure 13-6 show the results of our tests. The running time decreased in proportion to the number of subcircuits the circuit was split up into. We note that the protocol time in fact started to increase ever so slightly after $2^6 = 64$ subcircuits. Intuitively, this is because of the number of threads exceeds the virtual core count of the machine, where expensive context switches kick in.

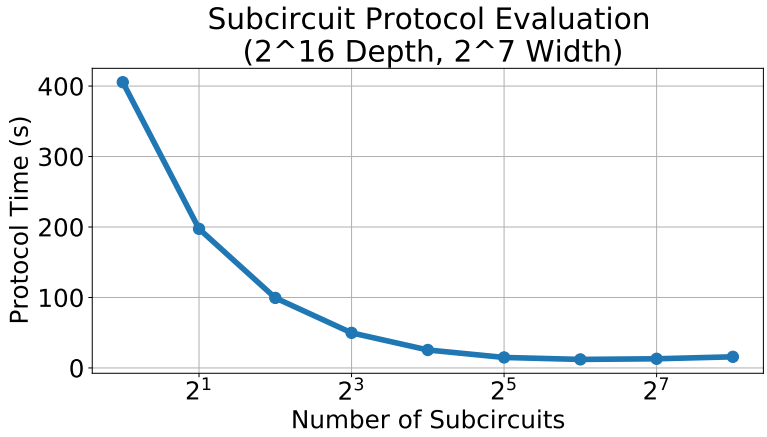
We see that the best increase in protocol time was when we tested with long and deep circuits that were split up into as many subcircuits as the thread count would allow. In our case, the number of virtual cores was 80, and once the number of subcircuits increased from 2^6 to 2^7, the protocol time began to increase. For depth 2^{16} and width 2^7 we see efficiency increase of 33.6×, and for depth 2^{16} and width 2^8 we see efficiency increase of 32.8×. Furthermore, for depth 2^{20} and width 2^7 we see efficiency increase of 33.8×, and for depth 2^{20} and width 2^8 we see efficiency increase of 34.8×.

For an ideal verifiable computation setup, the circuit would be split exactly in proportion with the number of virtual cores that are available. The case that subcircuits are dynamically added to be proven does not pose a problem because we take advantage of the streaming upload. Intuitively, the subcircuits earlier on are already proven, which frees computational resources for the subcircuits that are just being added.

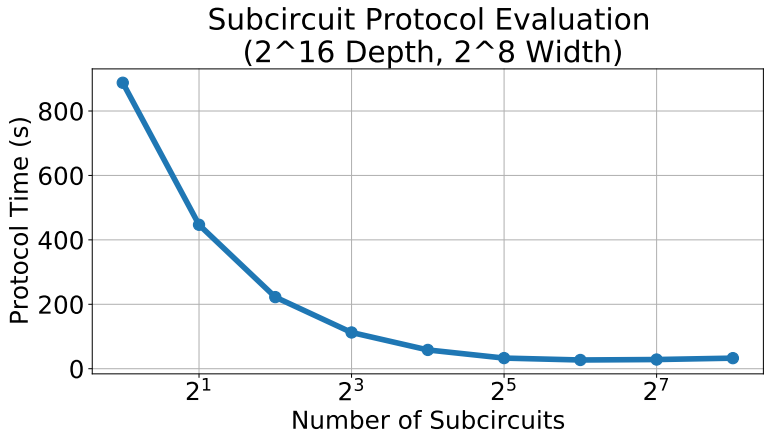
With the SHA-256 VDF circuit evaluations (Figure 13-7), we also see favorable results with the subcircuit protocol. With around ∼80 subcircuits generated per iteration, we see efficiency improvements of up to 26×, which is consistent with our self-constructed randomly generated circuits.

Hydra Protocol

For the Hydra protocol (Figure 13-8), we tested circuits composed of depth $2^{16} = 65536$ and widths of $2^7 = 128$ and $2^8 = 256$ (Table 13.2) . Because Hydra requires more overall computational power than a traditional approach as it consists of $d(s_i \cdot deg(f) + 1)$ sumchecks instead of only d sumchecks, we demonstrate our re-



(a)

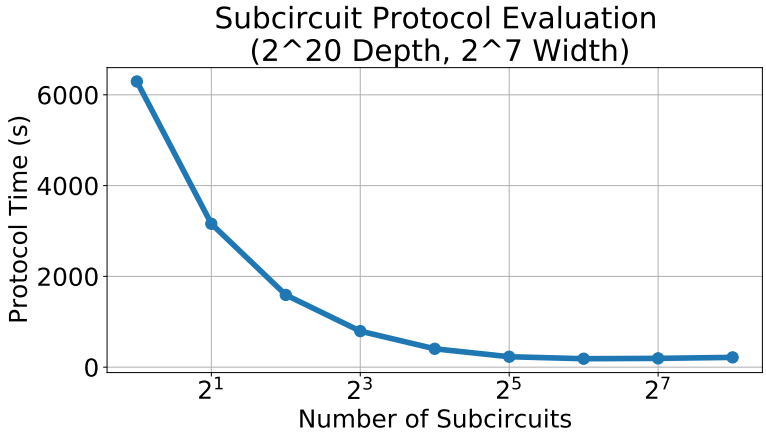


(b)

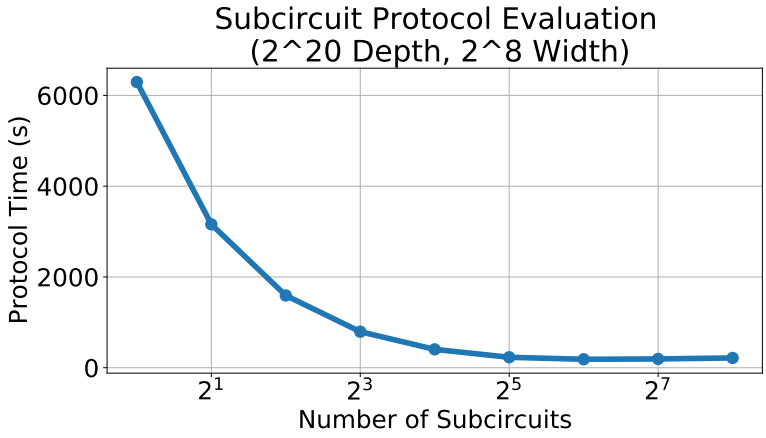
Figure 13-5: Subcircuit Evaluations.

sults with an artificial latency that highlights the benefit of Hydra's reduced round complexity. We argue that this latency is more representative of a real-life scenario. For example, when ensuring the validity of a cloud computation, there will clearly be a non-negligible amount of latency for conversation rounds between the client and the server. We evaluate Hydra with a 0ms latency as the ground truth, and from those results show the protocol times with 10ms and 20ms artificial latency period in between $\mathcal{P}$ and $\mathcal{V}$ interactions.

We see that with no latency, a traditional GKR protocol outperforms the Hydra protocol, however, once any amount of latency is added the true power of Hydra is



(a)



(b)

Figure 13-6: Subcircuit Evaluations (Cont.)

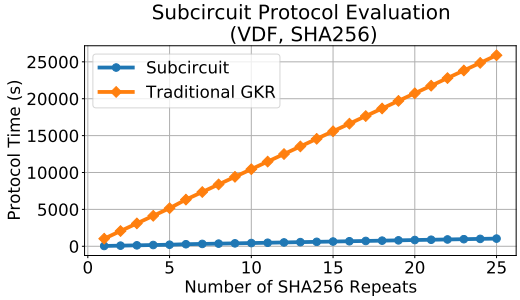


Figure 13-7: Subcircuit Protocol Evaluation (VDF, SHA256).

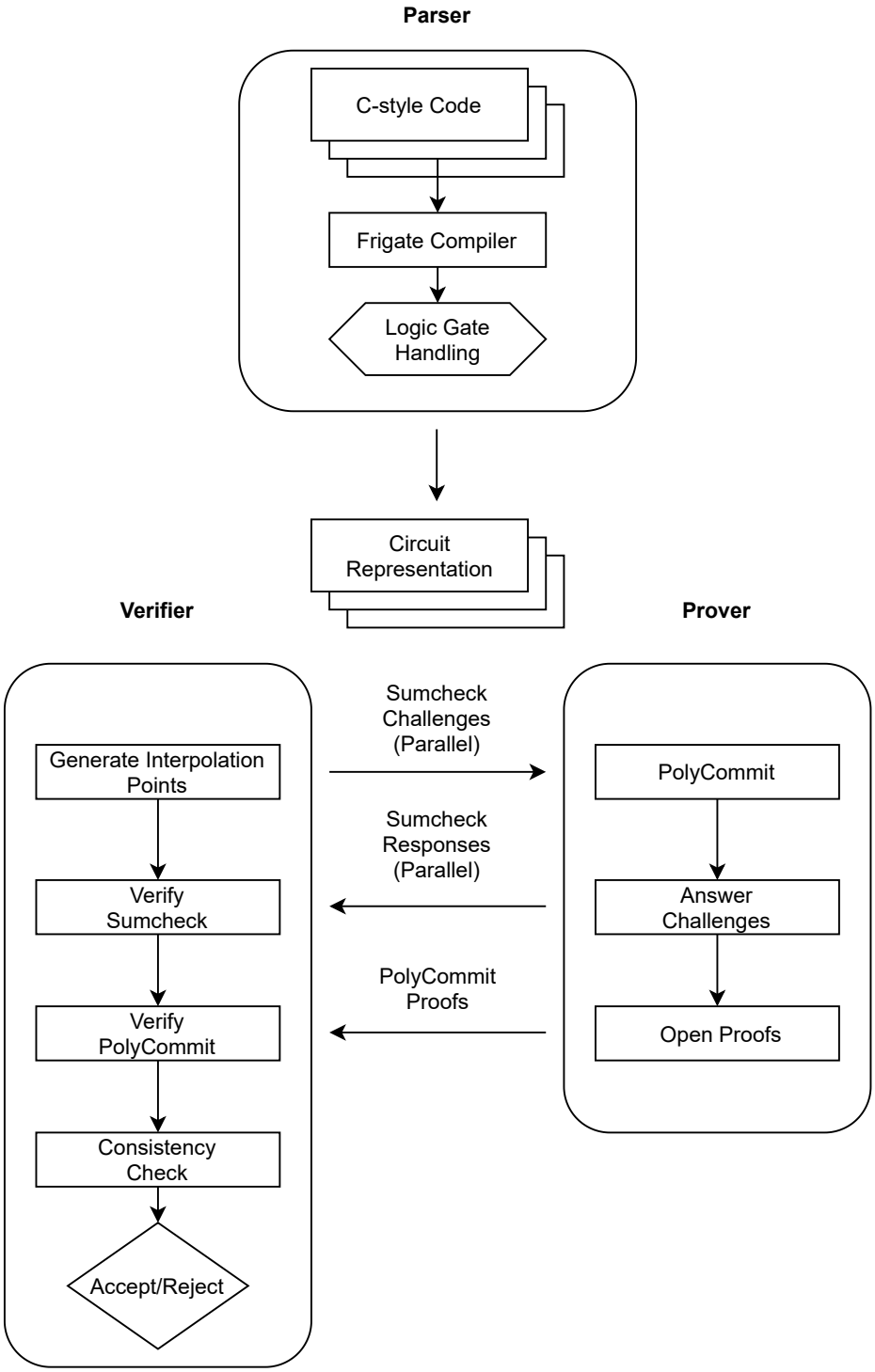
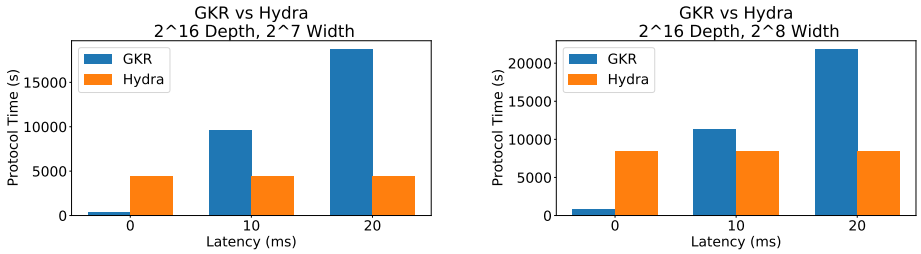


Figure 13-8: Hydra Environment.

	Protocol Time (s)			
Protocol	n	0ms Latency	10ms Latency	20ms Latency
GKR	2^7	399.8s	9574.8s	18749.9s
Hydra	2^7	**4384.4s**	**4384.5s**	**4384.7s**
GKR	2^8	883.5s	11369.3s	21855.0s
Hydra	2^8	**8459.9s**	**8460.1s**	**8460.2s**

Table 13.2: GKR vs Hydra: 2^{16} Depth, n Width.



(a) 2^{16} Depth, 2^7 Width

(b) 2^{16} Depth, 2^8 Width

Figure 13-9: GKR vs Hydra: 2^{16} Depth, 2^7 Width and 2^8 Width (Smaller protocol time is better).

revealed (Figure 13-9a and Figure 13-9b). With only a 10ms latency, Hydra comfortably approaches 2× protocol time efficiency for 2^{16} depth and 2^7 width, also improving by more than 1.3× for 2^{16} depth and 2^8 width. With 20ms of latency, these results are further compounded by efficiency improvements of 4.3× and 2.6×, respectively. We observe that these results play well with the advantageous long and skinny circuit structure, where the longer/skinnier the circuit is, the more the efficiency increases over the traditional GKR approach.

However, we do note that the Hydra protocol is not as flexible as the subcircuit protocol in that it does not fare nearly as well for circuits with rather long widths and short depths, such as with the SHA-256 circuit we tested with the VDF implementation (Figure 13-10). In this case, with 512ms latency per conversation round, Hydra is able to improve over the traditional GKR protocol by a factor of 2.7×. These results show that Hydra is still certainly feasible in select real-life scenarios where high latency is a driving factor and bandwidth is available to spare.

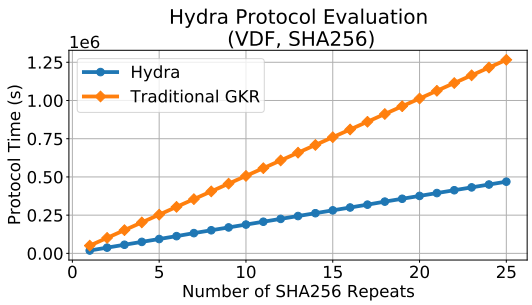


Figure 13-10: Hydra Protocol Evaluation (VDF, SHA-256, 2^9 ms Latency).

13.9 Future Work

Both the subcircuit protocol and the Hydra protocol rely on the cryptographic primitive of polynomial commitments to guarantee security. Therefore, they are formally categorized as interactive arguments. We are very interested to see whether this dependence is truly necessary for our protocols. However, more research is needed to see if this approach is viable. Without the polynomial commitments, $\mathcal{P}$ can cheat in the subcircuit protocol by executing GKR on malicious circuit compositions where the input and output layers of adjacent subcircuits do not match. Similarly, without polynomial commitments in the Hydra protocol, $\mathcal{P}$ can cheat by providing incorrect values that only agree in the certain subspace in which $\mathcal{V}$ interpolates.

13.10 Conclusion

In this chapter, we have described two new verifiable computation protocols: the subcircuit protocol for breaking a large circuit up into smaller circuits that can be proven in batches/streaming upload, and the Hydra protocol for the generalized parallel proving of all layers in the circuit. We implement the full verification system, compiling C-style code into logical circuits and passing it into our novel parser to convert them into provable representations for our protocols. Hydra provides a new option in the tradeoff round complexity and other factors for verifiable computation schemes. From standard assumptions, we collapse the round complexity to polylogarithmic in the width of the circuit, but only incur polylogarithmic blowup in band-

width and verifier time complexity. Our experimental results show that compared to traditional GKR implementations, the subcircuit protocol improves efficiency by 33.3× and the Hydra protocol improves efficiency by 4.3× in practical scenarios.

Chapter 14

Future Directions and Conclusion

14.1 General Framework for Isolation Levels

We have described how Litmus handles *serializability* and *repeatable read* as examples for verifiable isolation levels. However, the definition and approaches are ad-hoc, namely, they are specific to each isolation level. Naturally, we would like a general framework of cryptographic definitions of isolation levels, and we want to generalize Litmus to provide verifiability for such framework.

Here, we present a high-level idea. Per the definition of isolation levels, they define when a transaction can observe other concurrent transactions' effects. To catch transactions' effects, we list all the write operations $w_1, \ldots, w_n$ in the time order. We use $D_0, D_1, \ldots,$ to represent all the data states. A write operation w_i from a transaction $T(w_i)$ changes D_i to D_{i+1}. The isolation level determines which data state D_i a transaction reads a value from. Note that, the data state that the transaction sees is not necessarily the most recent one. For example, repeatable read transactions see the snapshot at the time when they start. Depending on the requirements of the isolation level, we define a function $\texttt{SnapshotForRead}(T, r, k, \{T_i\}, \{D_i\})$ that takes in the transaction T, the read operation r, the address k, the concurrent transactions $\{T_i\}$, and the data states $\{D_i\}$. It returns the data state which this particular read operation r should access. However, this is not enough. The updating order in the operation-level granularity is restricted by the isolation level. We define a function

`WriteOpOrderCheck`($\{D_i\}, \{w_i\}$) that takes in the data states and write operations and decides whether the write interleaving is valid or not. For example, consider two transactions, $T1$ and $T2$, where $T1$ writes $DB[A] \leftarrow 1, DB[B] \leftarrow 2$, and $T2$ writes $DB[A] \leftarrow 3, DB[B] \leftarrow 4$. An interleaving of $DB[A] \leftarrow 1, DB[A] \leftarrow 3, DB[B] \leftarrow 4, DB[B] \leftarrow 2$ results in the final state $DB[A] = 3, DB[B] = 2$. There is no permutation of $T1$ and $T2$ that produces this state. Therefore, this interleaving is not serializable. We expect the `WriteOpOrderCheck` function to return `no` in this case.

How do we adapt Litmus to the generalized definition? If we track all the data states D_i's with an append-only vector commitment mapping from the indices to the data states, we can plug the `SnapshotForRead` function into the memory integrity checker. At the same time, updating the digest should follow the `WriteOpOrderCheck` function. For example, in both the serializability and repeatable read examples, all the write operations of a transaction happen consecutively. Effectively, the digest updates proceed per transaction instead of per write operation.

However, both `SnapshotForRead` and `WriteOpOrderCheck` are likely complicated. It remains to see how we can support general isolation levels efficiently.

14.2 Full Database Schema Support

So far, we model the database as several relational tables consisting of numerical values, searchable texts, and unsearchable values. In the real world, DBMS supports many more data types. For example, modern DBMSs support structured data types like an array or a dictionary (e.g., a JSON object), and the queries can access the objects' attributes. Our current design is completely blind to the data structures. One way to support structured data is by flattening the object into a number of columns to make the attributes accessible. The user could create new attributes at any time. As mentioned previously in Section 5.3.6, Litmus supports dynamically adding and removing columns. However, this will create many authenticated dictionaries (as parts of the wildcard accumulator) if one row has a particularly long list.

The current approach to supporting blurry text search decomposes a pattern into

many different wildcard filters because it has to cover all the possibilities of wildcard lengths to align with the text. A more efficient approach is worth exploring.

14.3 Expressive Queries

Our current design only provides very low-level database access interfaces, namely, reading a value, writing a value, adding a row, deleting a row, adding a column, and removing a column. In theory, this can handle any kind of database operations. However, this would put too much logic inside the transaction circuit, resulting in degrading performance significantly.

It will be useful to make the design SQL-aware. Namely, we want explicit support and optimization of various SQL operations. For example, it would be great to support efficient and verifiable table joins. The naive way is to iterate through all the possible values on the left column and perform wildcard lookup on the right table. In the worst case, this generates $O(N^2)$ lookup proofs and $O(N)$ completeness proofs.

Supporting SQL sub-clauses is easier. The server can generate a temporary table and its corresponding digest. However, depending on the sub-query's semantic, the client circuit might have to compute the table's digest from scratch because it should not trust the server. For example, SQL allows creating a table out of nowhere by using **SELECT**. It remains to see how we could efficiently address this scenario.

Besides, the SQL language also supports many functions, from arithmetic operations like **ABS** to aggregate operations like **MAX**. Supporting them poses a great challenge to the authenticated data structure design.

14.4 NoSQL and Other Database Variants

New DBMS types are emerging with the increasing demand for simple, polymorphic, but massive data management. NoSQL DBMSs differ from conventional relational databases by not using relational data structures. There are many types of NoSQL databases, including document databases, key-value stores, wide-column databases,

and graph databases.

Key-value stores are optimized for dictionary operations (insert, delete, and find) and are more efficient than relational databases for these operations. The current design already supports key-value stores. Our wildcard accumulator naturally supports wide-column databases because it provides interfaces to search based on any row attribute and it is friendly to sparse columns.

Graph databases are optimized to efficiently traverse vertices through the edges connecting them and perform computation on the properties of vertices and/or edges. Unfortunately, due to the irregular structure of real-world graphs, memory bandwidth and latency are the major performance bottlenecks in graph databases. We can explore ways to hide this bottleneck in our verifiable graph database by using processors to help with verification when they are blocked on memory accesses.

It remains to see how we can adapt it to other DBMS variants.

14.5 Extension to Heterogeneous Environments

We can optimize our database for environments that contain heterogeneous resources, such as multi-core CPUs, GPUs, and domain-specific accelerators. For example, we can explore offloading compute-intensive cryptographic tasks to GPUs or domain-specific accelerators so that the CPUs are freer to execute the arbitrary transactional logic. We could also explore fault-tolerance techniques to ensure that the verifiable computation is able to complete even if some of the resources fail. For example, if the GPU or domain-specific accelerator fails, the CPU must be able to detect this and perform the cryptographic computations by itself. This is to enhance availability and power efficiency, as the server has a considerable workload, and starting everything over upon failure incurs wasted power.

14.6 Server Replication and Proactive Security

In Chapter 1, we mentioned a design alternative, where the client outsources the database to several servers and check whether the results from the servers agree. With a deterministic concurrency control algorithm like Deterministic Reservation (c.f. Chapter 7), when the servers are honest, the results should be the same. However, if we assume independence between server corruption events, the soundness probability decreases exponentially with the number of independent servers. Waiting for servers' responses increases the latency. For simplicity, assume the latency of a single server follows a uniform distribution $U(0, l)$ and the servers' latencies are independent, the expected latency of the last one of k servers is $k \cdot l/(k+1)$. However, this increase is bounded by l. Besides, this scheme does not use cryptography building blocks. Therefore, we expect it has a throughput comparable with no-verification outsourcing. We defer this alternative design to future work.

14.7 Conclusion

To conclude, this thesis presents Litmus, a comprehensive design towards a database management system that provides verifiable transaction correctness and semantic properties.

We believe that database security is more than just data privacy.

Data privacy is only effective for honest but curious attackers who still follow the software instructions to perform the computation and data hosting. In contrast, an active attacker breaching the operating system can control the server arbitrarily. Besides, these attackers are usually good at hiding themselves, leaving poor users to accept wrong data and results unconsciously. They pose a more significant security threat than an honest but curious attacker. A cryptographically verifiable database management system provides security from the bottom up. Moreover, the security Litmus provides is independent of the platform, the implementation, the cloud provider, or the weather. All you need to trust is cryptography (or rather,

mathematics). In some sense, a verifiable database management system provides similar functionalities to a blockchain system. Both of them provide trusted computation over data. However, blockchains offer trust by having the computation history publicly auditable, often at the cost of massive replication of resources (e.g., computing power for Proof-of-Work systems). Also, blockchains usually come with certain non-mathematical trust assumptions (e.g., more than two-thirds of the participants are honest for asynchronous BFT-based protocols; more than half of the computing power is honest for Bitcoin). Meanwhile, verifiable DBMSs are (comparatively) energy efficient without resource replication, and the trust assumptions are cleaner. However, a single-machine verifiable DBMS is a single point of failure, and therefore it does not offer the same availability as blockchain systems.

Recent enthusiasm for blockchain technologies has driven a wave of applied cryptography constructions. The stereotype of general-purpose cryptography being computationally expensive and impractical no longer holds. The advancement of cryptographic tools is enabling more and more secure applications. This thesis serves as an exploratory attempt to construct fully-fledged, secure, and practical database management systems. We anticipate more secure and more practical constructions to rise in the community.

Appendix A

Proofs

This chapter presents the proofs of the theorems in the thesis. For the readers' convenience, we duplicate the theorems alongside the proofs.

A.1 Authenticated Dictionary Properties

In this section we briefly prove the correctness and soundness of the authenticated dictionary scheme.

Theorem. *The authenticated dictionary scheme in Section 5.1.3 is correct and weakly key-binding.*

Proof. We prove the correctness and soundness as below:

- The correctness is straightforward. For key lookup correctness, we can see that given $\pi = \mathsf{ProveLookup}(\mathsf{prk}, D, K) = g^{\prod_{(k,v)\in D\setminus K}[H(k,v)]}$, we have $\pi^{\prod_{(k,v)\in(K,V)}[H(k,v)]} = (g^{\prod_{(k,v)\in D\setminus K}[H(k,v)]})^{\prod_{(k,v)\in(K,V)}[H(k,v)]} = g^{\prod_{(k,v)\in D}[H(k,v)]} = d$. Therefore, $\mathsf{VerLookup}(\mathsf{vrk}, d, K, V, \pi) = 1$.

- Similarly, for key non-membership correctness, we can see that $\pi = (A, B)$ s.t. $A \cdot S + B \cdot \prod_{k\in K} \mathsf{Sample}(\lambda, 0, k) = 1$. Raise g to both LHS and RHS we get $g^{A\cdot S + B\cdot\prod_{k\in K}\mathsf{Sample}(\lambda,0,k)} = g \Rightarrow (g^S)^A \cdot (g^{\cdot\prod_{k\in K}\mathsf{Sample}(\lambda,0,k)})^B = g \Rightarrow d^A \cdot (g^{\prod_{k\in K}\mathsf{Sample}(\lambda,0,k)})^B = g \Rightarrow \mathsf{VerNoKey}(\mathsf{vrk}, d, K, \pi) = 1$.

- For lookup soundness, we prove by breaking the Strong RSA assumption if the adversary violates the probability condition. Specifically, for lookup soundness, the adversary outputs with non-negligible probability $(d, K, K', V, V', \pi, \pi')$ that $\mathsf{VerLookup}(\mathsf{vrk}, d, K, V, \pi) = 1$ and $\mathsf{VerLookup}(\mathsf{vrk}, d, K', V', \pi') = 1$. Denote $H(D) = \prod_{(k,v)\in D}[H(k,v)]$ and $H(K,V) = \prod_{(k,v)\in(K,V)} H(k,v)$. Note that, since d is computed honestly, we have $d = g^{H(D)}$. As there exists $k \in K \cap K'$ s.t. $V(k) \neq V'(k)$, either at least one of $V(k)$ or $V'(k)$ is different from $D(k)$, or k is not in D. WLOG, let's assume $V(k) \neq D(k)$. $\mathsf{VerLookup}(\mathsf{vrk}, d, K, V, \pi) = 1 \Rightarrow \pi^{H(K,V)} = d$. Due to the construction of Prime Categorization, when k is not in D, we have $\mathsf{Sample}(\lambda, 0, k) \nmid H(D) \Rightarrow H(k, V(k)) \nmid H(D)$. when $V(k) \neq D(k)$, we have $\mathsf{Sample}(\lambda, 2, h(k, V(k))) \nmid H(D) \Rightarrow H(k, V(k)) \nmid H(D)$. So, in both cases, $H(k, V(k)) \nmid H(D)$. Denote $h = \gcd(H(k, V(k)), H(D)) < H(k, V(k))$. There exists $(A, B) = \mathsf{Bézout}(H(k, V(k))/h, H(D)/h)$ s.t. $A\frac{H(k,V(k))}{h} + B\frac{H(D)}{h} = 1 \Rightarrow \pi^{A\cdot H(k,V(k))/h} \cdot d^{B/h} = g \Rightarrow \pi = g^{\frac{h - B\cdot H(D)}{A\cdot H(k,V(k))}}$ This means the adversary is able to compute $g^{1/p}$ for $p = \frac{A\cdot H(k,V(k))}{h - B\cdot H(D)}$. Outputting (π, p) suffices to break the strong RSA assumption.

- For key non-membership soundness, we also prove by breaking the Strong RSA assumption. Namely, $\mathsf{VerLookup}(\mathsf{vrk}, d, K, V, \pi) = 1 \Rightarrow \pi^{H(K,V)} = d$, and $\mathsf{VerNoKey}(\mathsf{vrk}, d, K', \pi') = 1 \Rightarrow A \cdot H(D) + B \cdot H(K', V') = 1$. Given that $k \in K'$, $\mathsf{Sample}(\lambda, 0, k) | H(K', V')$. Therefore, k cannot be in D, otherwise we will have $\mathsf{Sample}(\lambda, 0, k) | H(D)$ and $\mathsf{Sample}(\lambda, 0, k) | \gcd(H(K', V'), H(D))$, which is conflicting with the existence of $\pi' = (A, B)$. Now the situation is the same as the lookup soundness proof, since $k \notin D$ yet the adversary is able to provide π s.t. $\mathsf{VerLookup}(\mathsf{vrk}, d, K, V, \pi) = 1$. Finally, we can break the Strong RSA assumption.

□

A.2 Proof of RSA Wildcard Accumulator Properties

Theorem. *The RSA-based wildcard accumulator is correct.*

Proof. For any $S \subseteq \mathcal{X}$ and any $\boldsymbol{f} \in \mathcal{F}$, we have $\mathsf{acc} = g^{\prod_{\boldsymbol{x} \in S} \mathsf{m}(\boldsymbol{x})}$, and:

- If $\mathsf{match}(\boldsymbol{f}, S) = \mathsf{yes}$, we have $\tilde{\rho} = \tilde{\pi} = \mathsf{Acc.GenWildcardMemWit}(\boldsymbol{f}, \mathsf{acc}, \mathsf{aux}) = g^{\prod_{\boldsymbol{x}' \in \mathsf{aux}} \mathsf{m}(\boldsymbol{x}')/\mathsf{m}(\boldsymbol{f})}$. Therefore, $\left(g^{\prod_{\boldsymbol{x}' \in \mathsf{aux}} \mathsf{m}(\boldsymbol{x}')/\mathsf{m}(\boldsymbol{f})}\right)^{\mathsf{m}(\boldsymbol{f})} = \mathsf{acc} \iff \tilde{\pi}^{\mathsf{m}(\boldsymbol{f})} = \mathsf{acc} \iff \mathsf{Acc.VerifyWildcard}(\boldsymbol{f}, \mathsf{yes}, \tilde{\rho}, \mathsf{acc}) = \mathsf{yes}$.

- If $\mathsf{match}(\boldsymbol{f}, S) = \mathsf{yes}$, we have $\tilde{\rho} = \tilde{\phi} = \mathsf{Acc.GenWildcardNonMemWit}(\boldsymbol{f}, \mathsf{acc}, \mathsf{aux}) = \mathsf{Bézout}(\mathsf{m}(\boldsymbol{f}), \prod_{\boldsymbol{x}' \in \mathsf{aux}} \mathsf{m}(\boldsymbol{x}')) := (\tilde{A}, \tilde{B})$, where

$$\mathsf{m}(\boldsymbol{f}) \cdot \tilde{A} + \prod_{\boldsymbol{x}' \in \mathsf{aux}} \mathsf{m}(\boldsymbol{x}') \cdot \tilde{B} = 1.$$

  Therefore, $\mathsf{m}(\boldsymbol{f}) \cdot \tilde{A} + \prod_{\boldsymbol{x}' \in \mathsf{aux}} \mathsf{m}(\boldsymbol{x}') \cdot \tilde{B} = 1 \Rightarrow g^{\mathsf{m}(\boldsymbol{f}) \cdot \tilde{A}} \cdot \left(g^{\prod_{\boldsymbol{x}' \in \mathsf{aux}} \mathsf{m}(\boldsymbol{x}')}\right)^{\tilde{B}} = g \iff g^{\tilde{A}\mathsf{m}(\boldsymbol{f})} \cdot \mathsf{acc}^{\tilde{B}} = g \iff \mathsf{Acc.VerifyWildcard}(\boldsymbol{f}, \mathsf{no}, \tilde{\rho}, \mathsf{acc}) = \mathsf{yes}$.

□

Theorem. *The RSA-based wildcard accumulator is weak key binding.*

Proof. Suppose there exists an adversary A, s.t.,

$$\Pr\left[\mathsf{Acc.VerifyWildcard}(\boldsymbol{f}, \neg\mathsf{flag}, \tilde{\rho}, \mathsf{acc}) = \mathsf{yes}. : \begin{array}{c} \tilde{\rho}, S, \boldsymbol{f} \leftarrow A(1^\lambda), \\ \mathsf{flag} \leftarrow \mathsf{match}(\boldsymbol{f}, S), \\ \mathsf{acc}, \mathsf{aux} \leftarrow \mathsf{Acc.Accumulate}(1^\lambda, S) \end{array}\right] \geq \frac{1}{\mathsf{poly}(\lambda)},$$

We can use A to break the RSA assumption. Given an RSA challenge $(a, \mathbb{G})$, we use a as the group generator g and call A with the group parameter. The adversary A returns the tuple $\tilde{\rho}, S, \boldsymbol{f}$.

- If $\mathsf{match}(\boldsymbol{f}, S) = \mathsf{yes}$, we have $\mathsf{flag} = \mathsf{no}$ and $\tilde{\rho} = (\tilde{A}, \tilde{B})$. Therefore,

$$\mathsf{Acc.VerifyWildcard}(\boldsymbol{f}, \mathsf{flag}, \tilde{\rho}, \mathsf{acc}) = \mathsf{yes} \tag{A.1}$$

$$\iff \mathsf{Acc.VerifyWildcard}(\boldsymbol{f}, \mathsf{no}, \tilde{\rho}, \mathsf{acc}) = \mathsf{yes} \tag{A.2}$$

$$\iff g^{\mathsf{m}(\boldsymbol{f})\cdot\tilde{A}} \cdot \left(g^{\prod_{\boldsymbol{x}'\in\mathsf{aux}} \mathsf{m}(\boldsymbol{x}')}\right)^{\tilde{B}} = g. \tag{A.3}$$

  By the definition, $\mathsf{m}(\boldsymbol{f}) = \prod_{I\subseteq\mathsf{SPI}(\boldsymbol{f})} \mathsf{Sample}(\lambda, \sum_{i\in I} 2^{i-1}, f^{(I)})$, and $\mathsf{m}(\boldsymbol{x}) = \prod_{I\subseteq\mathsf{SPI}(\boldsymbol{x})} \mathsf{Sample}(\lambda, \sum_{i\in I} 2^{i-1}, x^{(I)})$ for all $\boldsymbol{x} \in S$. Because $\mathsf{match}(\boldsymbol{f}, S) = \mathsf{yes}$, there exists $\boldsymbol{x}_0 \in S$ that $\mathsf{match}(\boldsymbol{f}, \boldsymbol{x}_0) = \mathsf{yes} \Rightarrow \forall i \in [l], x_0^{(i)} = f^{(i)} \vee f^{(i)} = \star \Rightarrow \mathsf{SPI}(\boldsymbol{f}) \subseteq \mathsf{SPI}(\boldsymbol{x}_0)$. We have $\Rightarrow \mathsf{m}(\boldsymbol{f}) | \mathsf{m}(\boldsymbol{x}_0)$. W.L.O.G., assume $\mathsf{m}(\boldsymbol{x}_0) = k \cdot \mathsf{m}(\boldsymbol{f})$. Now, we know Equation A.3 can be rewritten into

$$\left(g^{\tilde{A}+k\cdot\tilde{B}\cdot\prod_{\boldsymbol{x}'\neq\boldsymbol{x}_0\in\mathsf{aux}} \mathsf{m}(\boldsymbol{x}')}\right)^{\mathsf{m}(\boldsymbol{f})} = g.$$

  Namely, we can break the RSA assumption by outputting $(g^{\tilde{A}+k\cdot\tilde{B}\cdot\prod_{\boldsymbol{x}'\neq\boldsymbol{x}_0\in\mathsf{aux}} \mathsf{m}(\boldsymbol{x}')}, \mathsf{m}(\boldsymbol{f}))$.

- If $\mathsf{match}(\boldsymbol{f}, S) = \mathsf{no}$, we have $\mathsf{flag} = \mathsf{yes}$ and $\tilde{\rho} = \tilde{\pi}$ s.t. $(\tilde{\pi})^{\mathsf{m}(\boldsymbol{f})} = \mathsf{acc}$. Because $\mathsf{match}(\boldsymbol{f}, S) = \mathsf{no}$, for all $\boldsymbol{x} \in S$, we have $\mathsf{match}(\boldsymbol{f}, \boldsymbol{x}) = \mathsf{no}$. Suppose $gcd(\mathsf{m}(\boldsymbol{f}), \prod_{\boldsymbol{x}\in S} \mathsf{m}(\boldsymbol{x})) = d$. Now, we prove a lemma about d:

  Lemma 1. *$m(\boldsymbol{x})/d \neq 1$ with high probability.*

  Proof. If $\mathsf{m}(\boldsymbol{f})/d = 1$, it means $\mathsf{m}(\boldsymbol{f}) | \prod_{\boldsymbol{x}\in S} \mathsf{m}(\boldsymbol{x})$. So, $\forall I \subseteq \mathsf{SPI}(\boldsymbol{f})$, $\mathsf{Sample}(\boldsymbol{\lambda}, \sum_{\boldsymbol{i}\in\boldsymbol{I}} \boldsymbol{2^{i-1}}, \boldsymbol{f^{(I)}}) | \prod_{\boldsymbol{x}\in\boldsymbol{S}} \mathsf{m}(\boldsymbol{x})$. In particular, the prime number $PrimeSample(\lambda, \sum_{i\in\mathsf{SPI}(\boldsymbol{f})} 2^{i-1}, f^{(\mathsf{SPI}(\boldsymbol{f}))})$ (denoted as p) divides $\prod_{\boldsymbol{x}\in S} \mathsf{m}(\boldsymbol{x})$. There must exist a $\boldsymbol{x}_0$ s.t. $p | \mathsf{m}(\boldsymbol{x}_0)$. Because the prime number categories are disjoint and the sampler is collision resistant, we know $f^{(\mathsf{SPI}(\boldsymbol{f}))} = x_0^{(\mathsf{SPI}(\boldsymbol{f}))}$ with high probability. In this case, $\sum_{i\in\mathsf{SPI}(\boldsymbol{f})} |\mathcal{X}|^i f^{(i)} = \sum_{i\in\mathsf{SPI}(\boldsymbol{f})} |\mathcal{X}|^i x_0^{(i)} \Rightarrow f^{(i)} = x_0^{(i)}$ for all $i \in \mathsf{SPI}(\boldsymbol{f})$. This implies $\mathsf{match}(\boldsymbol{f}, \boldsymbol{x}_0) = \mathsf{yes}$, which contradicts the condition $\mathsf{match}(\boldsymbol{f}, S) = \mathsf{no}$. □

  Therefore, $\mathsf{m}(\boldsymbol{f})/d \geq \min \mathbb{P} > 3$. By Bézout, there exists $\tilde{A}$ and $\tilde{B}$ s.t. $\mathsf{m}(\boldsymbol{f}) \cdot$

$\tilde{A} + \prod_{\boldsymbol{x}' \in \mathsf{aux}} \mathsf{m}(\boldsymbol{x}') \cdot \tilde{B} = d$. This implies $g^{\mathsf{m}(\boldsymbol{f}) \cdot \tilde{A}} \cdot \left(g^{\prod_{\boldsymbol{x}' \in \mathsf{aux}} \mathsf{m}(\boldsymbol{x}')}\right)^{\tilde{B}} = g^d \iff g^{\tilde{A}\mathsf{m}(\boldsymbol{f})} \cdot \mathsf{acc}^{\tilde{B}} = g^d \iff g^{\tilde{A}\mathsf{m}(\boldsymbol{f})} \cdot \left(\tilde{\pi}^{\mathsf{m}(\boldsymbol{f})}\right)^{\tilde{B}} = g^d \iff \left(g^{\tilde{A}} \tilde{\pi}^{\tilde{B}}\right)^{\frac{\mathsf{m}(\boldsymbol{f})}{d} \cdot d} = g^d$.

Namely, we can break the RSA assumption by outputting $(g^{\tilde{A}} \tilde{\pi}^{\tilde{B}}, \frac{\mathsf{m}(\boldsymbol{f})}{d})$.

Finally, in both cases, we can break the strong RSA assumption with overwhelming probability. □

Theorem. *The RSA-based wildcard accumulator is witness private.*

Proof. We construct the simulator as below:

- When receiving the trapdoor $\tau = \phi(N)$ from the challenger (where N is the RSA modulus), store τ.
- Randomly sample $S' \subseteq \mathcal{X}$.
- Compute $(\mathsf{acc}, \mathsf{aux}) \leftarrow \mathsf{Acc.Accumulate}(1^\lambda, S')$ and send acc to A.
- When receiving a filter $\boldsymbol{f}$ from the adversary A, get $\mathsf{flag} \leftarrow D(\boldsymbol{f})$.
- If $\mathsf{flag} = \mathsf{yes}$:
  - Compute $(A, B) \leftarrow \mathsf{Bézout}(\mathsf{m}(\boldsymbol{f}), \tau)$.
  - Respond with $\tilde{\rho}' \leftarrow \mathsf{acc}^A$.
- If $\mathsf{flag} = \mathsf{no}$:
  - Let $t \leftarrow \gcd(\mathsf{m}(\boldsymbol{f}), \prod_{\boldsymbol{x} \in S'} \mathsf{m}(\boldsymbol{x}))$ and compute $(\alpha, \beta) \leftarrow \mathsf{Bézout}(t, \tau)$.
  - Compute $(A, B) \leftarrow \mathsf{Bézout}(\mathsf{m}(\boldsymbol{f}), \prod_{\boldsymbol{x} \in S'} \mathsf{m}(\boldsymbol{x}))$.
  - Respond with $\tilde{\rho}' \leftarrow (A \cdot \alpha, B \cdot \alpha)$.

The responses from the simulator follow the same distribution as the ones from the real world, and the witnesses will pass the verification. □

A.3 Proof of TXN20-based Wildcard Accumulator Properties

We first prove Theorem 5.

Theorem. *Given an instance of TXN20-based authenticated dictionary* $D = (c, S) = (\prod_i \left(S^{1/e_i}\right)^{v_i}, g^{\prod e_i}) := (g^e, g^{\prod e_i})$. *We have* $\prod v_i \neq 0$ *if and only if*

$$gcd(e, \prod_i e_i) = 1.$$

Proof. We rewrite e in the form of

$$e = \sum_i v_i \cdot e_1 e_2 \dots e_{i-1} e_{i+1} \dots e_{|K|}.$$

The statement contains two directions:

- $\prod v_i \neq 0 \rightarrow \gcd(e, \prod_i e_i) = 1$: This is straightforward. Since $\prod v_i \neq 0$, $v_i \neq 0 \forall i$. Notice that $\forall i, \gcd(e, e_i) = \gcd(v_i \cdot e_1 e_2 \dots e_{i-1} e_{i+1} \dots e_{|K|}, e_i) = 1$ as e_i does not divide v_i.
- $\gcd(e, \prod_i e_i) = 1 \rightarrow \prod v_i \neq 0$: Suppose the contrary, if $\prod v_i = 0$, there exists i_0 s.t. $v_{i_0} = 0$. Since for all $i \neq i_0$, we have e_{i_0} divides the i-th term $v_i \cdot e_1 e_2 \dots e_{i-1} e_{i+1} \dots e_{|K|}$. Therefore, $\gcd(e, \prod_i e_i) = e_{i_0}$.

□

Then, we present the proof of Theorem 6.

Theorem (Strongly Binding)**.** *The non-membership proof of value domain is strongly binding. Namely, for any efficient adversary* $\mathcal{A}$*, it cannot, with non-negligible probability, produce an authenticated dictionary* $D = (S, c, w)$*, a value* v*, a lookup proof* $\pi = (S_i, \Lambda_i, y)$ *on the key pair* (k_i, v)*, and a non-membership proof* $\phi = (A, B)$ *of the value* v*, such that both* π *and* ϕ *are valid, i.e.,*

$$S_i^{e_i} = S, S_i^v \cdot \Lambda^{e_i} = c, (c/w^v)^A \cdot S^B = g.$$

Proof. Suppose such an adversary exists. Given an instance of RSA challenge (g, N), we use g as the generator for the RSA group $\mathbb{G} = \mathbb{Z}_N$. Denote $z = c/w^v$ as the commitment to the original authenticated dictionary "minus" a unit authenticated dictionary times v. We first extract B from the PoKE proof.

Since we have

$$z^A \cdot S^B = g \tag{A.4}$$
$$\iff [(S_i^v \Lambda_i^{e_i}) / w^v]^A \cdot (S_i^{e_i})^B = g \tag{A.5}$$
$$\iff [\Lambda_i^{e_i} / (w/S_i)^v]^A \cdot (S_i^{e_i})^B = g \tag{A.6}$$
$$\iff \left[\Lambda_i / \left((w/S_i)^{1/e_i}\right)^v\right]^A \cdot S_i^B = g^{1/e_i} \tag{A.7}$$
$$\iff [\Lambda_i / y^v]^A \cdot S_i^B = g^{1/e_i}. \tag{A.8}$$

We can represent the e_i-th root of g with Λ_i, y, A and B. This breaks the Strong RSA assumption. □

Theorem. *The TXN20-based wildcard accumulator is correct.*

Proof. For any $S \subseteq \mathcal{X}$ and any $\boldsymbol{f} \in \mathcal{F}$,

- If $\mathsf{match}(\boldsymbol{f}, S) = \mathsf{yes}$, we have $\tilde{\rho} = \tilde{\pi} = \mathsf{Acc.GenWildcardMemWit}(\boldsymbol{f}, \mathsf{acc}, \mathsf{aux}) = (\{\Lambda_i\}_{i \in \mathsf{SPI}(\boldsymbol{f})}; S_{\boldsymbol{x}})$, where $S_{\boldsymbol{x}} = S^{1/\mathsf{cfhash}(\boldsymbol{x})}$ and $\Lambda_i = \prod_{\boldsymbol{x}' \neq \boldsymbol{x}} \left(S_{\boldsymbol{x}}^{1/\mathsf{cfhash}(\boldsymbol{x}')}\right)^{\boldsymbol{x}'^{(i)}}$. Therefore, $(S_{\boldsymbol{x}})^{\mathsf{cfhash}(\boldsymbol{x})} = (S^{1/\mathsf{cfhash}(\boldsymbol{x})})^{\mathsf{cfhash}(\boldsymbol{x})} = S$ and

$$(S_{\boldsymbol{x}})^{\boldsymbol{x}^{(i)}} (\Lambda_i)^{\mathsf{cfhash}(\boldsymbol{x})} \tag{A.9}$$
$$= (S^{1/\mathsf{cfhash}(\boldsymbol{x})})^{\boldsymbol{x}^{(i)}} \left(\prod_{\boldsymbol{x}' \neq \boldsymbol{x}} \left(S_{\boldsymbol{x}}^{1/\mathsf{cfhash}(\boldsymbol{x}')}\right)^{\boldsymbol{x}'^{(i)}}\right)^{\mathsf{cfhash}(\boldsymbol{x})} \tag{A.10}$$
$$= (S^{1/\mathsf{cfhash}(\boldsymbol{x})})^{\boldsymbol{x}^{(i)}} \left(\prod_{\boldsymbol{x}' \neq \boldsymbol{x}} (S_{\boldsymbol{x}})^{\boldsymbol{x}'^{(i)}}\right) \tag{A.11}$$
$$= \prod_{\boldsymbol{x}} (S_{\boldsymbol{x}})^{\boldsymbol{x}^{(i)}} = d_i, \forall i \in \mathsf{SPI}(\boldsymbol{f}) \tag{A.12}$$

  Equivalently, $\mathsf{Acc.VerifyWildcard}(\boldsymbol{f}, \mathsf{yes}, \tilde{\rho}, \mathsf{acc}) = \mathsf{yes}$. When using random linear combinations, we know

$$\prod_{i\in\mathsf{SPI}(\boldsymbol{f})}\left[(S_{\boldsymbol{x}})^{\boldsymbol{x}^{(i)}}\,(\Lambda_i)^{\mathsf{cfhash}(\boldsymbol{x})}\right]^{r_i} \tag{A.13}$$

$$=(S_{\boldsymbol{x}})^{\sum_{i\in\mathsf{SPI}(\boldsymbol{f})} r_i\cdot\boldsymbol{x}^{(i)}}\left(\prod_{i\in\mathsf{SPI}(\boldsymbol{f})}\Lambda_i\right)^{\mathsf{cfhash}(\boldsymbol{x})} \tag{A.14}$$

$$=(S_{\boldsymbol{x}})^{\bar{\boldsymbol{x}}}\,(\Lambda)^{\mathsf{cfhash}(\boldsymbol{x})}=\prod d_i^{r_i}=d, \tag{A.15}$$

where $\bar{\boldsymbol{x}}=\sum_{i\in\mathsf{SPI}(\boldsymbol{f})} r_i\cdot\boldsymbol{x}^{(i)}$ is the weighted sum of the supporting dimensions of $\boldsymbol{x}$.

- If $\mathsf{match}(\boldsymbol{f},S)=\mathsf{no}$, we have $\tilde{\rho}=\tilde{\phi}=(\tilde{A},\tilde{B})$ s.t. $A\cdot e+B\cdot\prod e_i=1$, where e is the exponent of (d/w^v) and w is the unit dictionary, i.e., $w=\prod_{\boldsymbol{x}} S^{1/\mathsf{cfhash}(\boldsymbol{x})}$. Note that

$$\left(\frac{d}{w^v}\right)^{\tilde{A}}\cdot S^{\tilde{B}} \tag{A.16}$$

$$=g^{\tilde{A}e+B\prod e_i}=g \tag{A.17}$$

In both cases, Acc.VerifyWildcard evaluates to yes. □

Before we talk about the strong binding property, we present a lemma first.

Lemma 2. *Any inconsistent filter-flag pair set C must contain at least two elements. For any single filter-flag pair $(\boldsymbol{f},\mathsf{flag})$, there exists a set S that makes .*

Proof. Suppose otherwise, $|C|=1$. Denote the only element in C as $(\boldsymbol{f}_0,\mathsf{flag})$. Consider the following two cases:

- $\mathsf{flag}=\mathsf{yes}$: Take an arbitrary element $x_0\in\mathcal{X}$. If we replace all the $\star$ in $\boldsymbol{f}_0$ with x_0, we get an element $\boldsymbol{x}_0$ s.t. $\mathsf{match}(\boldsymbol{f}_0,\boldsymbol{x}_0)=\mathsf{yes}$. We can construct a set $S_0=\{\boldsymbol{x}_0\}$ and $\mathsf{match}(\boldsymbol{f}_0,S_0)$.

- $\mathsf{flag}=\mathsf{no}$: We can construct a set $S_0=\emptyset$ and $\mathsf{match}(\boldsymbol{f}_0,S_0)=\mathsf{no}$.

In both cases, it contradicts the assumption C is inconsistent. Therefore, $|C| \geq 2$. For any single filter-flag pair $(\boldsymbol{f}, \mathsf{flag})$, we have constructed a set S that $\mathsf{match}(\boldsymbol{f}, S) = \mathsf{flag}$. □

Theorem. *The TXN20-based wildcard accumulator is strong key binding.*

Proof. Because C is inconsistent, from Lemma 2 we know $|C| \geq 2$. Take a filter-flag pair $(\boldsymbol{f}_0, \mathsf{flag}_0)$ from C, there exists a set $\mathcal{S}$ such that $\mathsf{match}(\boldsymbol{f}_0, \mathcal{S}) = \mathsf{flag}$. Because C is inconsistent, there exists a filter-flag pair $(\boldsymbol{f}, \mathsf{flag}) \in C$ s.t. $\mathsf{match}(\boldsymbol{f}, \mathcal{S}) \neq \mathsf{flag}$. Now we discuss different cases:

- If $\mathsf{flag} = \mathsf{yes}$, denote $\tilde{\rho} = (\Lambda_i | i \in \mathsf{SPI}(\boldsymbol{f}); S_{\boldsymbol{x}}; \mathsf{cfhash}(\boldsymbol{x}))$.
  - If $\mathsf{cfhash}(\boldsymbol{x}) | \prod_{\boldsymbol{x}' \in \mathcal{S}} \mathsf{cfhash}(\boldsymbol{x}')$, we know there exists $\boldsymbol{x}_0 \in \mathcal{S}$ s.t. $\mathsf{cfhash}(\boldsymbol{x}) = \mathsf{cfhash}(\boldsymbol{x}_0)$ since $\mathsf{cfhash}(\cdot)$ are prime numbers. By the collision resistant property, we know $\boldsymbol{x} = \boldsymbol{x}_0$ with overwhelming probability. Because $\mathsf{Acc.VerifyWildcard}(\boldsymbol{f}, \mathsf{yes}, \tilde{\rho}, \mathsf{acc}) = \mathsf{yes}$, we know $d_i = (S_{\boldsymbol{x}})^{\boldsymbol{f}^{(i)}} (\Lambda_i)^{\mathsf{cfhash}(\boldsymbol{x})}$ holds for $i \in \mathsf{SPI}(\boldsymbol{f})$. As $\boldsymbol{x} = \boldsymbol{x}_0$, we have

$$d_i = (S_{\boldsymbol{x}_0})^{\boldsymbol{f}^{(i)}} (\Lambda_i)^{\mathsf{cfhash}(\boldsymbol{x}_0)}, \forall i \in \mathsf{SPI}(\boldsymbol{f}).$$

    Since $\mathsf{match}(\boldsymbol{f}, \mathcal{S}) = \mathsf{no}$, we know there exists $i_0 \in \mathsf{SPI}(\boldsymbol{f})$ s.t. $\boldsymbol{x}^{(i_0)} \neq \boldsymbol{f}^{(i_0)}$. Note that the tuple $(\Lambda_{i_0}; S_{\boldsymbol{x}_0})$ form a lookup proof of the TXN20 dictionary. We can construct an adversary to break its strongly-binding soundness.
  - If $\mathsf{cfhash}(\boldsymbol{x})$ does not divide $\prod_{\boldsymbol{x}' \in \mathcal{S}} \mathsf{cfhash}(\boldsymbol{x}')$, because $\mathsf{cfhash}(\cdot)$ outputs prime numbers, we know $\gcd(\mathsf{cfhash}(\boldsymbol{x}), \prod_{\boldsymbol{x}' \in \mathcal{S}} \mathsf{cfhash}(\boldsymbol{x}')) = 1$. Therefore, there exist A and B s.t. $A \cdot \mathsf{cfhash}(\boldsymbol{x}) + B \cdot \prod_{\boldsymbol{x}' \in \mathcal{S}} \mathsf{cfhash}(\boldsymbol{x}') = 1$. Suppose there exists an adversary A that breaks the strong binding property in the TXN20-based wildcard accumulator. Now, we construct an adversary to break the strong RSA assumption. We use the challenge as the group generator of the authenticated dictionary scheme and call the adversary A to get the set C, $\mathsf{cfhash}(\boldsymbol{x})$ and $S_{\boldsymbol{x}}$. Now, $g^{A \cdot \mathsf{cfhash}(\boldsymbol{x}) + B \cdot \prod_{\boldsymbol{x}' \in S} \mathsf{cfhash}(\boldsymbol{x}')} = g \iff \left(g^A \cdot g^{\frac{B}{\mathsf{cfhash}(\boldsymbol{x})} \cdot \prod_{\boldsymbol{x}' \in \mathcal{S}} \mathsf{cfhash}(\boldsymbol{x}')}\right)^{\mathsf{cfhash}(\boldsymbol{x})} = g$. Because $S_{\boldsymbol{x}}^{\mathsf{cfhash}(\boldsymbol{x})} = S =$

$g^{\prod_{\boldsymbol{x}' \in \mathcal{S}} \mathsf{cfhash}(\boldsymbol{x}')}$, we know $\left(g^A \cdot S_{\boldsymbol{x}}^B\right)^{\mathsf{cfhash}(\boldsymbol{x})} = g$. In other words, we can return $(g^A \cdot S_{\boldsymbol{x}}^B, \mathsf{cfhash}(\boldsymbol{x}))$ to break the strong RSA assumption.

- If $\mathsf{flag} = \mathsf{no}$: We decompose $\tilde{\rho}$ into the tuple $(A, b := S^B)$. Because $\mathsf{Acc.VerifyWildcard}(\boldsymbol{f}, \mathsf{no}, \tilde{\rho}, \mathsf{acc}) = \mathsf{yes}$, we know $\left(\frac{d}{w^v}\right)^A \cdot S^B = g$. We know $\mathsf{match}(\boldsymbol{f}, \mathcal{S}) = \mathsf{yes}$. It means there exists $\boldsymbol{x}_0 \in \mathcal{S}$ s.t. $\forall i \in \mathsf{SPI}(\boldsymbol{f}), \boldsymbol{x}_0^{(i)} = \boldsymbol{f}^{(i)} \Rightarrow \sum_{i \in \mathsf{SPI}(\boldsymbol{f})} r_i \cdot \boldsymbol{x}_0^{(i)} = \sum_{i \in \mathsf{SPI}(\boldsymbol{f})} r_i \cdot \boldsymbol{f}^{(i)} = v$.

  We can produce a wildcard membership witness of $\boldsymbol{x}_0$ by computing $(\Lambda_i | i \in \mathsf{SPI}(\boldsymbol{f}); S_{\boldsymbol{x}_0}; y) = \mathsf{Acc.GenWildcardMemWit}(\boldsymbol{f}, \mathsf{acc}, \mathcal{S})$ and generate the linearly combined membership witness $\Lambda = \prod \Lambda_i^{r_i}$. At the same time, Λ is a wildcard membership witness of $v = \sum_{i \in \mathsf{SPI}(\boldsymbol{f})} r_i \cdot \boldsymbol{f}^{(i)}$ of the random linear combination TXN20 dictionary $d = \prod d_i^{r_i}$.

  From Theorem 6 we know that, with both the non-membership witness (A, b) and the membership witness $(\Lambda, S_{\boldsymbol{x}_0}, y)$ of the same filter $\boldsymbol{f}$, we can break the strong RSA assumption.

□

Theorem. *The TXN20-based wildcard accumulator is witness private.*

Proof. We construct the simulator as below:

- When receiving the trapdoor $\tau = \phi(N)$ from the challenger, store τ.
- Randomly sample $S' \subseteq \mathcal{X}$.
- Compute $(\mathsf{acc} = (d_1, \ldots, d_l; S), \mathsf{aux}) \leftarrow \mathsf{Acc.Accumulate}(1^\lambda, S')$ and send acc to A.
- When receiving a filter $\boldsymbol{f}$ from the adversary A, get $\mathsf{flag} \leftarrow D(\boldsymbol{f})$.
- Compute $w \leftarrow \prod_{\boldsymbol{x}' \in S'} g^{\sigma / \mathsf{cfhash}(\boldsymbol{x}')}$, where $\sigma = \prod_{\boldsymbol{x}' \in S'} \mathsf{cfhash}(\boldsymbol{x}')$.
- If $\mathsf{flag} = \mathsf{yes}$:
  - Sample $\boldsymbol{x}$ uniformly randomly and calculate $e \leftarrow \mathsf{cfhash}(\boldsymbol{x})$.

- Compute $(A, B) \leftarrow \mathsf{Bézout}(e, \tau)$ and let $S_{\boldsymbol{x}} \leftarrow S^A$.
- Compute $\Lambda_i \leftarrow \left(\frac{d_i}{S_{\boldsymbol{x}}^{\boldsymbol{f}^{(i)}}}\right)^A$ for $i \in \mathsf{SPI}(\boldsymbol{f})$.
- Compute $y = (w/S_{\boldsymbol{x}})^A$.
- Respond with $\tilde{\rho}' = (\{\Lambda_i\}; S_{\boldsymbol{x}}; y)$ or $\tilde{\rho}' = (\Lambda := \prod_{i \in \mathsf{SPI}(\boldsymbol{f})} (\Lambda_i)^{r_i}; S_{\boldsymbol{x}}; y)$ if using random linear combinations.

- If $\mathsf{flag} = \mathsf{no}$:
  - Compute the random linear combinations

$$d := \prod d_i^{r_i}, v := \sum_{i \in \mathsf{SPI}(\boldsymbol{f})} r_i \cdot \boldsymbol{f}^{(i)}.$$

  - Denote ε_i as the base-g logarithm of d_i and compute

$$\varepsilon_i = \prod \varepsilon_i = \sum_{\boldsymbol{x}' \in S'} \left[\boldsymbol{x}'^{(i)} \cdot \sigma / \mathsf{cfhash}(\boldsymbol{x}')\right],$$

    where $\sigma = \prod_{\boldsymbol{x}' \in S'} \mathsf{cfhash}(\boldsymbol{x}')$ is the logarithm of S.
  - Denote ε as the base-g logarithm of d and compute $\varepsilon = \sum \varepsilon_i \cdot r_i$.
  - Compute the base-g logarithm of w,

$$\varepsilon_w = \prod \varepsilon_i = \sum_{\boldsymbol{x}' \in S'} \left[\sigma / \mathsf{cfhash}(\boldsymbol{x}')\right].$$

  - Compute $t \leftarrow \gcd(\varepsilon - v \cdot \varepsilon_w, \sigma)$ and $(\alpha, \beta) \leftarrow \mathsf{Bézout}(t, \tau)$.
  - Compute $(A, B) \leftarrow \mathsf{Bézout}(\varepsilon - v \cdot \varepsilon_w, \sigma)$.
  - Respond with $\tilde{\rho}' = (A \cdot \alpha, B \cdot \alpha)$.

The responses from the simulator follow the same distribution as the ones from the real world, and the witnesses will pass the verification. □

A.4 Proof of Serializability Soundness

For the readers' convenience, we recall the definition of serializability soundness here:

Definition (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 $\mathcal{T} = \{T_1, \ldots, T_n\}$ and an initial database D_0 that*

$$\Pr[\mathsf{Verify}(\mathcal{T}, \delta_0, \mathcal{A}(D_0, \mathcal{T})) = \mathsf{yes}]$$

is non-negligible (where $\delta_0 = \mathsf{Digest}(D_0)$), the extractor $\mathcal{E}^{(\mathcal{A})}$ outputs databases $\{D_i\}_{i\in\{1,\ldots,n\}}$ and a permutation σ on $\{1, \ldots, n\}$ such that the following quantity is negligibly *close to 1 in the (implicit) security parameter:*

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

Before we discuss the soundness, we explain the scheme tuple (Digest, Execute, Verify). The Digest function refers to the authenticated data structure used in the memory integrity. It takes in a database state and outputs a digest corresponding to this state. The Execute function refers to the database execution process and the proving process. The Verify function refers to the VC verification process plus the circuit well-formedness check and return value *AllCommit* check.

The adversary $\mathcal{A}$ takes in the initial database D_0 and the transaction set $\mathcal{T}$. It outputs the final digest δ', the transaction outputs $\mathcal{V}$, the VC proof π, and the auxiliary information. In case of Litmus, the auxiliary information contains the circuit C.

Now, we prove the soundness.

Theorem. *Litmus provides serializability soundness.*

Proof. Since Groth16 is an argument of knowledge, there exists an extractor $\mathcal{E}_{vc}$ that can efficiently uncover the witness from the adversary $\mathcal{A}$.

We construct the extractor $\mathcal{E}^{(\mathcal{A})}$ as below:

Since $\Pr[\mathsf{Verify}(\mathcal{T}, \delta_0, \mathcal{A}(D_0, \mathcal{T})) = \mathsf{yes}] = 1/\mathsf{poly}$, $\mathcal{E}$ repeats calling $\mathcal{A}(D_0, \mathcal{T})$ until $\mathsf{Verify}(\mathcal{T}, \delta_0, \mathcal{A}(D_0, \mathcal{T})) = \mathsf{yes}$. $\mathcal{E}^{(\mathcal{A})}$ rewinds $\mathcal{E}_{vc}$ to uncover the extra inputs: the read values and the corresponding memory integrity proofs.

Then, $\mathcal{E}^{(\mathcal{A})}$ extracts the inputs of $\mathcal{A}$ to get the initial database D_0 and the circuit C. By the well-formedness check, C is a chain of transactions with memory integrity checkers plugged in. From C, the extractor learns the permutation σ by matching each of the transaction logic parts in C with $\mathcal{T}$.

It replays the transactions $\mathcal{T}$ in the order of σ and gets the database states D_i for $i \in \{1, 2, \ldots, n\}$. Naturally, we have $(D_i, v_i) \leftarrow T_{\sigma(i)}(D_{i-1})$. We mark these transaction executions as the *ideal* execution.

Now, we only need to show that D_n makes $\mathsf{Digest}(D_n) = \delta'$. Denote the digest after running $T_{\sigma(i)}$ as δ_i.

By the soundness of the authenticated dictionary scheme, for every read value v of address k in $T_{\sigma(i)}$, $D_{i-1}[k] = v$ holds with an overwhelming probability. Since the transactions are deterministic, the write set caused by the transactions $T_{\sigma(i)}$ are the same as that in the ideal execution. As the circuit updates the digest honestly, we have the following:

$$\Pr\left[\mathsf{Digest}(D_i) = \delta_i | \mathsf{Digest}(D_{i-1}) = \delta_i\right] \geq 1 - \varepsilon_{AD} \cdot k,$$

where k is the maximum number of values per transaction reads.

We can get that

$$\Pr\left[\mathsf{Digest}(D_n) = \delta_n\right] \tag{A.18}$$

$$= \Pr\left[\mathsf{Digest}(D_n) = \delta_n | \mathsf{Digest}(D_0) = \delta_0\right] \tag{A.19}$$

$$\geq (1 - \varepsilon_{AD})^n = 1 - \mathsf{negl} \tag{A.20}$$

□

Appendix B

Aggregation with PoKCR

This appendix presents tricks to aggregate the linear-sized wildcard membership witness into single element. The relation $\mathcal{R}_{PoKCR}$ is defined as

$$\mathcal{R}_{PoKCR} = \{(\alpha_i, x_i) : w_i | w_i^{x_i} = \alpha_i \wedge \gcd(\{x_i\}) = 1\},$$

where w_i's are the hidden witnesses.

Suppose we have a wildcard membership witness $\tilde{\pi} = (\Lambda_i | i \in \mathsf{SPI}(\boldsymbol{f}); S_{\boldsymbol{x}}, \mathsf{cfhash}(\boldsymbol{x}), y)$ regarding a filter $\boldsymbol{f}$. For the readers' convenience, we recall the verification process:

- Check whether $(S_{\boldsymbol{x}})^{\mathsf{cfhash}(\boldsymbol{x})}$.
- Check whether $d_i = (S_{\boldsymbol{x}})^{\boldsymbol{f}^{(i)}} (\Lambda_i)^{\mathsf{cfhash}(\boldsymbol{x})}$ for all $i \in \mathsf{SPI}(\boldsymbol{f})$.

We re-write the second condition as

$$\frac{d_i}{(\Lambda_i)^{\mathsf{cfhash}(\boldsymbol{x})}} = (S_{\boldsymbol{x}})^{\boldsymbol{f}^{(i)}}, \forall i \in \mathsf{SPI}(\boldsymbol{f}). \tag{B.1}$$

Let $w_i = \frac{d_i}{(\Lambda_i)^{\mathsf{cfhash}(\boldsymbol{x})}}$, $x_i = \boldsymbol{f}^{(i)}$, and $\alpha_i = S_{\boldsymbol{x}}$. If we make the value space $\mathcal{X}$ a set of prime numbers, $\boldsymbol{f}^{(i)}$'s are co-prime. We can use the PoKCR protocol to prove Equation B.1 with a single element:

$$w := \prod w_i = \prod \frac{d_i}{(\Lambda_i)^{\mathsf{cfhash}(\boldsymbol{x})}}.$$

The verifier computes $x^* := \prod x_i = \prod \boldsymbol{f}^{(i)}$ and $z = \prod \alpha_i^{x^*/x_i}$. To verify, it checks whether:

$$w^{x^*} = z.$$

Please refer to BBF18 [33] for detailed discussion on the correctness and soundness.

Appendix C

Further Discussion on Hydra

C.1 Subcircuit Protocol Guarantees

Here, we provide the complete proof behind the validity of the subcircuit protocol as described.

Completeness

The completeness is straightforward from the completeness of sumcheck and polynomial commitments, as well as the protocol description.

Soundness

In the scope of the individual subcircuits, the soundness is guaranteed by the GKR protocol. With regards to connecting the input and output layers of adjacent subcircuits, the extractability of polynomial commitments in [49] for bounded polynomial-time $\mathcal{P}$ and $\mathcal{V}$ guarantees its soundness. A formal proof follows.

Let $\mathbb{F}$ be a prime field. Let C be a layered arithmetic circuit, given an *input* and a claimed *output*. C has depth d, with circuit layers labeled from the output layer 1 to the input layer d. C is split into k equal-depth subcircuits denoted as c_i for all $1 \leq i \leq k$. The $subinput_i$ of c_i (input of subcircuit c_i) directly corresponds with the $suboutput_{i-1}$ of c_{i-1} (output of subcircuit c_{i-1}). For each subcircuit c_i, let $\widetilde{W}_{in_i}$

denote the multilinear extension of the input layer, and $\widetilde{W}_{out_i}$ denote the multilinear extension of the output layer.

Suppose that $C(input) \neq output$. Assume there exists an adversarial $\mathcal{P}^*$ such that

$$\Pr[\langle \mathcal{P}^*, \mathcal{V} \rangle = accept] = p.$$

Let A be the event in which $\langle \mathcal{P}^*, \mathcal{V} \rangle = accept$. For all $1 \leq i \leq k$, let G_i be the event where $c_i(subinput_i) \neq suboutput_i$ and let P_i be the event where $\widetilde{W}_{in_i} \neq \widetilde{W}_{out_i}$. We observe that

$$\begin{aligned} p &= \Pr[A] \\ &\leq \Pr[\exists i \in [k] \text{ s.t. } A \wedge (G_i \vee P_i)] \\ &\leq \sum_{i=1}^{k} \Pr[A \wedge (G_i \vee P_i)] \\ &\leq \sum_{i=1}^{k} \big(\Pr[A \wedge G_i] + \Pr[A \wedge P_i] \big). \end{aligned} \tag{C.1}$$

The soundness of the GKR protocol implies that

$$\begin{aligned} &\sum_{i=1}^{k} \big(\Pr[A \wedge G_i] \\ &\leq \sum_{i=1}^{k} \frac{s_i \cdot \frac{d}{k}}{|\mathbb{F}|} \\ &\leq O\big(\frac{s_i \cdot d}{|\mathbb{F}|}\big). \end{aligned} \tag{C.2}$$

From the computational extractability of the polynomial commitment scheme, we see

that

$$\begin{aligned}
&\sum_{i=1}^{k} \big(\Pr[A \wedge P_i] \\
&\leq \sum_{i=1}^{k} \lambda \\
&\leq O\big(k(\lambda)\big). \qquad \text{(C.3)}
\end{aligned}$$

where λ is the security parameter used in [49] of the commitment. Finally, by the union bound we obtain

$$\begin{aligned}
p &= \Pr[A] \\
&\leq \Pr[\exists i \in [k] \text{ s.t. } A \wedge (G_i \vee P_i)] \\
&\leq \sum_{i=1}^{k} \Pr[A \wedge (G_i \vee P_i)] \\
&\leq \sum_{i=1}^{k} \big(\Pr[A \wedge G_i] + \Pr[A \wedge P_i] \big) \\
&\leq O\big(\frac{s_i \cdot d}{|\mathbb{F}|} + k(\lambda)\big). \qquad \text{(C.4)}
\end{aligned}$$

C.2 Hydra Protocol Guarantees

Here, we provide the complete proof behind the validity of the Hydra protocol as described.

Completeness

The completeness is straightforward from the completeness of sumcheck and polynomial commitments, and the protocol description.

Soundness

In order to guarantee the security of the Hydra protocol, we must first guarantee the security of the sumcheck's coin tosses. As stated earlier, if at any point f or

its evaluation points are compromised, a malicious prover can completely bypass the soundness of the sumcheck protocol as it knows the coin tosses ahead of time. Thus, we first prove that in the perspective of $\mathcal{P}$, all coin tosses are completely random and probabilistically unpredictable.

Let $n = s_i$ and $m = s_i \cdot deg(f) + 1$. For the sake of simplicity, assume $deg(f) = 1$ (f is linear in each of its dimensions). At each layer, $\mathcal{P}$ views a system of polynomial equations as follows:

$$\begin{array}{ccc} x_1y_1 + z_1 = c_{1,1} & ... & x_1y_m + z_1 = c_{1,m} \\ ... & ... & ... \\ x_ny_1 + z_n = c_{n,1} & ... & x_ny_m + z_n = c_{n,m}. \end{array}$$

where x_i, z_i are variables for $1 \leq i \leq n$, y_i is a variable for $1 \leq i \leq m$, and $c_{i,j}$ is a constant for $1 \leq i \leq n$ and $1 \leq j \leq m$. At first glance, this seems like an overdetermined system of polynomials, as the total number of variables is $2n+m$ and the total number of equations is $n \cdot m$. However, we establish that because of the structure of such equations, this is not the case.

We can form a reduction by linear combination of such a system by subtracting $x_iy_j + z_i = c_{i,j}$ from $x_iy_1 + z_i = c_{i,1}$ so that we obtain

$$\begin{array}{ccc} x_1(y_1 - y_2) = c_{1,1} - c_{1,2} & ... & x_1(y_1 - y_m) = c_{1,1} - c_{1,m} \\ ... & ... & ... \\ x_n(y_1 - y_2) = c_{n,1} - c_{n,2} & ... & x_n(y_1 - y_m) = c_{n,1} - c_{n,m}. \end{array}$$

Factoring out common x_i will reduce this system to dependent ratios in the form of

$$\begin{array}{ccc} \frac{y_1-y_2}{y_1-y_3} = \frac{c_{1,1}-c_{1,2}}{c_{1,1}-c_{1,3}} = & ... & = \frac{c_{n,1}-c_{n,2}}{c_{n,1}-c_{n,3}} \\ ... & ... & ... \\ \frac{y_1-y_2}{y_1-y_m} = \frac{c_{1,1}-c_{1,2}}{c_{1,1}-c_{1,m}} & ... & = \frac{c_{n,1}-c_{n,2}}{c_{n,1}-c_{n,m}} \end{array}$$

Note that the chain of equalities must hold by construction: $\mathcal{V}$ generated the system of polynomials such that it is valid. Therefore, in order for a correct solution to exist,

the ratio of the constants shown must be equivalent and consistent.

We now demonstrate how a solution of the reduced system yields a valid solution to the original system. Given that the above reduction holds, we see that there must exist an $x_i \in \mathbb{F}$ for all $1 \leq i \leq n$ such that

$$\begin{array}{ccc} \frac{x_1(y_1-y_2)=c_{1,1}-c_{1,2}}{x_1(y_1-y_3)=c_{1,1}-c_{1,3}} & \cdots & \frac{x_n(y_1-y_2)=c_{n,1}-c_{n,2}}{x_n(y_1-y_3)=c_{n,1}-c_{n,3}} \\ \cdots & \cdots & \cdots \\ \frac{x_1(y_1-y_2)=c_{1,1}-c_{1,2}}{x_1(y_1-y_m)=c_{1,1}-c_{1,m}} & \cdots & \frac{x_n(y_1-y_2)=c_{n,1}-c_{n,2}}{x_n(y_1-y_m)=c_{n,1}-c_{n,m}} \end{array}.$$

generalizing this for $1 \leq i \leq n$ and $1 \leq j \leq m$ we get

$$x_i(y_1 - y_j) = c_{i,1} - c_{i,j}$$

which expands to

$$x_i y_1 - x_i y_j = c_{i,1} - c_{i,j}$$

Corresponding to the original system of equations, we want to find z_i such that

$$x_i y_1 + z_i = c_{i,1}$$
$$x_i y_j + z_i = c_{i,j}$$

Notice that

$$x_i y_1 - x_i y_j = c_{i,1} - c_{i,j} \implies c_{i,1} - x_i y_1 = c_{i,j} - x_i y_j.$$

Thus, if we set z_i equal to $x_i y_1 - c_{i,1} = x_i y_j - c_{i,j}$, we obtain

$$x_i y_1 + (c_{i,1} - x_i y_1) = c_{i,1}$$
$$x_i y_j + (c_{i,j} - x_i y_j) = c_{i,j}.$$

We can now see by the transitive property of equality that this is indeed a valid solution for $x_i y_j + z_i = c_{i,j}$ across all $1 \leq i \leq n$ and $1 \leq j \leq m$. It is clear how y_1 and y_2 can be the two degrees of freedom in this system. Any y_1 and y_2 fixed in the space of $\mathbb{F}$ will result in a valid system. The remaining $y_3, ..., y_m$ are easily derivable,

and once the y_i values are known for all $1 \leq i \leq m$, plugging them back into the original system will quickly yield x_i and z_i for all $1 \leq i \leq m$. Thus, the probability of $\mathcal{P}$ bypassing soundness when f is linear in each of its dimensions is $\frac{1}{|\mathbb{F}|^2}$. With this in mind, we now proceed with the remainder of the proof.

Suppose that $C(\textit{input}) \neq \textit{output}$. Assume there exists an adversarial $\mathcal{P}^*$ such that

$$\Pr[\langle \mathcal{P}^*, \mathcal{V} \rangle = \textit{accept}] = p.$$

Recall that the Hydra protocol takes $s_i \cdot \textit{deg}(f) + 1$ claims on $\widetilde{W}_i$ for every $1 \leq i \leq d$ layers, which reduces to $2(s_{i+1} \cdot \textit{deg}(f) + 1)$ claims on $\widetilde{W}_{i+1}$. Namely, across all $i \in [d]$, $s_i \cdot \textit{deg}(f) + 1$ claims that $\widetilde{W}_i(r_{i,j}) = R_{i,j}$ are reduced to $s_{i+1} \cdot \textit{deg}(f) + 1$ verifications of $\widetilde{W}_{i+1}(a_{i,j}) = A_{i,j}$ and $s_{i+1} \cdot \textit{deg}(f) + 1$ verifications of $\widetilde{W}_{i+1}(b_{i,j}) = B_{i,j}$.

As we assume the layers in C obey the same logspace, s_i is the equivalent across all $1 \leq i \leq d$. Note that $r_{i,j}$, $a_{i,j}$, and $b_{i,j}$ are all points on f, however, because neither f nor the locations where f is evaluated at are revealed, in the perspective of $\mathcal{P}$ the random coin tosses of the sumchecks are in fact completely random and probabilistically unpredictable.

Let A be the event in which $\langle \mathcal{P}^*, \mathcal{V} \rangle = \textit{accept}$. For all $1 \leq i \leq d$, let T_i be the event in which indeed all $\widetilde{W}_i(r_{i,j}) = R_{i,j}$ for layer i across all $1 \leq j \leq s_i \cdot \textit{deg}(f) + 1$. We observe that

$$\begin{aligned}
p &= \Pr[A] \\
&= \Pr[A \wedge \neg(T_1) \wedge T_d] \\
&\leq \Pr[\exists i \in [d] \text{ s.t. } A \wedge \neg(T_{i-1}) \wedge T_i] \\
&\leq \sum_{i=1}^{d} \Pr[A \wedge \neg(T_{i-1}) \wedge T_i]. \qquad \text{(C.5)}
\end{aligned}$$

Let E_i be the event in which indeed all $\widetilde{W}_{i+1}(a_{i,j}) = A_{i,j}$ and $\widetilde{W}_{i+1}(b_{i,j}) = B_{i,j}$ for

$i \in [d]$. We can see that

$$\begin{aligned}
&\sum_{i=1}^{d} \Pr[A \wedge \neg(T_{i-1}) \wedge T_i] \\
=&\sum_{i=1}^{d} (\Pr[A \wedge \neg(T_{i-1}) \wedge T_i \wedge E_i] \\
&+ \Pr[A \wedge \neg(T_{i-1}) \wedge T_i \wedge \neg(E_i)]).
\end{aligned} \tag{C.6}$$

First, the soundness property of the sumcheck protocol across $s_i \cdot deg(f) + 1$ instances for d layers implies that

$$\begin{aligned}
&\sum_{i=1}^{d} \Pr[A \wedge \neg(T_{i-1}) \wedge T_i \wedge E_i] \\
&\leq \sum_{i=1}^{d} \Pr[A \wedge \neg(T_{i-1}) \wedge E_i] \\
&\leq \sum_{i=1}^{d} \frac{s_i(s_i \cdot deg(f) + 1)}{|\mathbb{F}|} + \frac{1}{|\mathbb{F}|^2} \\
&\leq d(\frac{s_i(s_i \cdot deg(f) + 1)}{|\mathbb{F}|} + \frac{1}{|\mathbb{F}|^2}) \\
&\leq O\big(\frac{d \cdot s_i^2 \cdot deg(f)}{|\mathbb{F}|}\big).
\end{aligned} \tag{C.7}$$

Second, note that a ground truth T_i will result in the correct claims of the query points where $\mathcal{V}$ interpolates on to obtain the polynomial $h_i(x)$, which subsequently is indeed equal to $\widetilde{W}_i(f(x))$. If that is the case, there exists no situation where $\mathcal{V}$ accepts a false E_i event where the claimed values for the challenge points are not consistent. In regard to the soundness of the polynomial commitment, [49] shows that the extractability of their construction guarantees that it is computationally sound. Thus, given a correct $h_i(x)$, $\mathcal{V}$ will always reject $\neg(E_i)$ with probability negligibly

close to 1 owing to the polynomial commitment. Therefore,

$$\begin{aligned}
&\sum_{i=1}^{d} \Pr[A \wedge \neg(T_{i-1}) \wedge T_i \wedge \neg(E_i)] \\
&\leq \sum_{i=1}^{d} \Pr[A \wedge T_i \wedge \neg(E_i)] \\
&\leq O\big(d(\lambda)\big) \qquad \text{(C.8)}
\end{aligned}$$

where λ is the security parameter used in [49] of the commitment for polynomial-time $\mathcal{P}$. Finally, by the union bound we obtain

$$\begin{aligned}
p &= \Pr[A] \\
&\leq \sum_{i=1}^{d} \Pr[A \wedge \neg(T_{i-1}) \wedge T_i] \\
&\leq \sum_{i=1}^{d} \big(\Pr[A \wedge \neg(T_{i-1}) \wedge T_i \wedge E_i] \\
&+ \Pr[A \wedge \neg(T_{i-1}) \wedge T_i \wedge \neg(E_i)]\big) \\
&\leq O\big(\frac{d \cdot s_i^2 \cdot deg(f)}{|\mathbb{F}|}) + O(d(\lambda))\big) \\
&\leq O\big(d(\frac{s_i^2 \cdot deg(f)}{|\mathbb{F}|} + \lambda)\big). \qquad \text{(C.9)}
\end{aligned}$$

Bibliography

[1] Gretchen. https://github.com/aphyr/gretchen.

[2] RSA factoring challenge. `https://en.wikipedia.org/wiki/RSA_Factoring_Challenge`. Accessed: 2022-07-25.

[3] Scott Ames, Carmit Hazay, Yuval Ishai, and Muthuramakrishnan Venkitasubramaniam. Ligero: Lightweight sublinear arguments without a trusted setup. In *ACM CCS*, 2017.

[4] A. Anagnostopoulos, M.T. Goodrich, and R. Tamassia. Persistent Authenticated Dictionaries and Their Applications. In *4th International Conference on Information Security (ISC)*, 2001.

[5] Arkworks. Poly-commit rust library. `https://github.com/arkworks-rs/poly-commit`.

[6] arkworks contributors. `arkworks` zksnark ecosystem, 2022.

[7] Sanjeev Arora, Carsten Lund, Rajeev Motwani, Madhu Sudan, and Mario Szegedy. Proof verification and the hardness of approximation problems. *Journal of the ACM (JACM)*, 45(3):501–555, 1998.

[8] Amazon AWS. AWS cloud database leak: California voter data exposed, held for ransom - MSSP alert, 2017.

[9] Amazon AWS. State & local government, 2020.

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

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

[12] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *ACM CCS*, 1993.

[13] Eli Ben-Sasson, Iddo Bentov, Alessandro Chiesa, Ariel Gabizon, Daniel Genkin, Matan Hamilis, Evgenya Pergament, Michael Riabzev, Mark Silberstein, Eran Tromer, and Madars Virza. Computational integrity with a public random string from quasi-linear PCPs. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 551–579, 2017.

[14] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. *Cryptology ePrint Archive*, 2018.

[15] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. Cryptology ePrint Archive, Report 2018/046, 2018. `https://eprint.iacr.org/`.

[16] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable zero knowledge with no trusted setup. In *Annual international cryptology conference*, pages 701–732. Springer, 2019.

[17] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, and Eran Tromer. On the concrete-efficiency threshold of probabilistically-checkable proofs. In *Electronic Colloquium on Computational Complexity (ECCC)*, volume 19, page 45, 2012.

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

[19] Eli Ben-Sasson, Alessandro Chiesa, Michael Riabzev, Nicholas Spooner, Madars Virza, and Nicholas P. Ward. Aurora: Transparent succinct arguments for r1cs. Cryptology ePrint Archive, Report 2018/828, 2018. `https://eprint.iacr.org/`.

[20] Eli Ben-Sasson, Alessandro Chiesa, and Nicholas Spooner. Interactive oracle proofs. In *Theory of Cryptography Conference*, pages 31–60. Springer, 2016.

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

[22] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct non-interactive zero knowledge for a von neumann architecture. In *USENIX Security*, 2014.

[23] Eli Ben-Sasson, Oded Goldreich, Prahladh Harsha, Madhu Sudan, and Salil Vadhan. Short PCPs verifiable in polylogarithmic time. In *20th Annual IEEE Conference on Computational Complexity (CCC'05)*, pages 120–134. IEEE, 2005.

[24] Eli Ben-Sasson and Madhu Sudan. Short PCPs with polylog query complexity. *SIAM Journal on Computing*, 38(2):551–607, 2008.

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

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

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

[28] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Julian Shun. Internally deterministic parallel algorithms can be fast. In *PPoPP*, pages 181–192, 2012.

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

[30] Robert L. Bocchino, Vikram S. Adve, Sarita V. Adve, and Marc Snir. Parallel programming must be deterministic by default. In *Usenix HotPar*, 2009.

[31] D. Boneh, E. Kushilevitz, R. Ostrovsky, and W. E. Skeith. Public key encryption that allows pir queries. In *CRYPTO*, volume 4622 of *LNCS*, pages pp.50–67, 2007.

[32] Dan Boneh, Joseph Bonneau, Benedikt Bünz, and Ben Fisch. Verifiable delay functions. In *IACR CRYPTO*, 2018.

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

[34] Jonathan Bootle, Andrea Cerulli, Pyrros Chaidos, Jens Groth, and Christophe Petit. Efficient zero-knowledge arguments for arithmetic circuits in the discrete log setting. In *Annual International Conference on Advances in Cryptology (EUROCRYPT)*, page 327–357, 2016.

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

[36] Thomas Brewster. 191 million US voter registration records leaked in mystery database, 2015.

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

[38] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 315–334, 2018.

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

[40] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *IEEE Symposium on Security and Privacy*, 2018.

[41] Jan Camenisch, Markulf Kohlweiss, and Claudio Soriente. An accumulator based on bilinear maps and efficient revocation for anonymous credentials. In *International workshop on public key cryptography*, pages 481–500. Springer, 2009.

[42] Jan Camenisch and Anna Lysyanskaya. Dynamic accumulators and application to efficient revocation of anonymous credentials. In *Annual international cryptology conference*, pages 61–76. Springer, 2002.

[43] Matteo Campanelli, Dario Fiore, Nicola Greco, Dimitris Kolonelos, and Luca Nizzardo. Incrementally aggregatable vector commitments and applications to verifiable decentralized storage. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 3–35. Springer, 2020.

[44] Matteo Campanelli, Dario Fiore, Nicola Greco, Dimitris Kolonelos, and Luca Nizzardo. Vector commitment techniques and applications to verifiable decentralized storage. *IACR Cryptol. ePrint Arch.*, 2020:149, 2020.

[45] Matteo Campanelli, Rosario Gennaro, Steven Goldfeder, and Luca Nizzardo. Zero-knowledge contingent payments revisited: Attacks and payments for services. In *ACM CCS*, 2017.

[46] Dario Catalano and Dario Fiore. Vector commitments and their applications. In *International Workshop on Public Key Cryptography*, pages 55–72. Springer, 2013.

[47] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-mat: A recursive model for graph mining. In *SDM*, pages 442–446, 2004.

[48] Thomas Chen, Hui Lu, Teeramet Kunpittaya, and Alan Luo. A review of zk-snarks, 2022.

[49] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Noah Vesely, and Nicolas Ward. Marlin: Preprocessing zksnarks with universal and updatable srs. In *IACR EUROCRYPT*, 2020.

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

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

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

[53] Graham Cormode, Michael Mitzenmacher, and Justin Thaler. Practical verified computation with streaming interactive proofs. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, pages 90–112, 2012.

[54] Graham Cormode, Michael Mitzenmacher, and Justin Thaler. Practical verified computation with streaming interactive proofs. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, 2012.

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

[56] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, 1998.

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

[58] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudré-Mauroux. Oltp-bench: An extensible testbed for benchmarking relational databases. *PVLDB*, 7(4):277–288, 2013.

[59] Saba Eskandarian and Matei Zaharia. Oblidb: Oblivious query processing for secure databases. *arXiv preprint arXiv:1710.00458*, 2017.

[60] Kousha Etessami and Patrice Godefroid. A model of transactional programming. Technical Report MSR-TR-2008-19, January 2008.

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

[62] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Proceedings on Advances in Cryptography*, 1987.

[63] Benjamin Fuller, Mayank Varia, Arkady Yerukhimovich, Emily Shen, Ariel Hamlin, Vijay Gadepally, Richard Shay, John Darby Mitchell, and Robert K Cunningham. Sok: Cryptographically protected database search. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 172–191. IEEE, 2017.

[64] Ariel Gabizon, Zachary J Williamson, and Oana Ciobotaru. Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. *IACR Cryptol. ePrint Arch.*, 2019:953, 2019.

[65] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct NIZKs without PCPs. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 626–645, 2013.

[66] Craig Gentry and Daniel Wichs. Separating succinct non-interactive arguments from all falsifiable assumptions. In *ACM Symposium on Theory of Computing*, 2011.

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

[68] Esha Ghosh, Olga Ohrimenko, Dimitrios Papadopoulos, Roberto Tamassia, and Nikos Triandopoulos. Zero-knowledge accumulators and set operations. *Cryptology ePrint Archive*, 2015.

[69] Irene Giacomelli, Jesper Madsen, and Claudio Orlandi. Zkboo: Faster zero-knowledge for boolean circuits. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 1069–1083. USENIX Association, 2016.

[70] Shafi Goldwasser and Yael Tauman Kalai. On the (in)security of the fiat-shamir paradigm. In *IEEE Symposium on Foundations of Computer Science*, 2003.

[71] Shafi Goldwasser, Yael Tauman Kalai, and Guy N. Rothblum. Delegating computation: Interactive proofs for muggles. In *ACM Symposium on Theory of Computing*, 2008.

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

[73] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof-systems. In *ACM Symposium on Theory of Computing*, 1985.

[74] Jim Gray. Why do computers stop and what can be done about it? In *Symposium on Reliability in Distributed Software and Database Systems*, pages 3–12, 1986.

[75] Jens Groth. Linear algebra with sub-linear zero-knowledge arguments. In *Annual International Cryptology Conference*, pages 192–208, Berlin, Heidelberg, August 2009. Springer.

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

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

[78] Christian Hammer, Julian Dolby, Mandana Vaziri, and Frank Tip. *Dynamic Detection of Atomic-Set-Serializability Violations*, page 231–240. Association for Computing Machinery, New York, NY, USA, 2008.

[79] Chuntao Hong, Dong Zhou, Mao Yang, Carbo Kuo, Lintao Zhang, and Lidong Zhou. Kuafu: Closing the parallelism gap in database replication. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 1186–1195. IEEE, 2013.

[80] iden3. Circom circuit compiler, 2022.

[81] Yuval Ishai, Eyal Kushilevitz, and Rafail Ostrovsky. Efficient arguments without short PCPs. In *Twenty-Second Annual IEEE Conference on Computational Complexity (CCC'07)*, pages 278–291, 2007.

[82] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Zero-knowledge from secure multiparty computation. In *Proceedings of the Thirty-Ninth Annual ACM Symposium on Theory of Computing*, STOC '07, page 21–30, New York, NY, USA, 2007. Association for Computing Machinery.

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

[84] Marek Jawurek, Florian Kerschbaum, and Claudio Orlandi. Zero-knowledge using garbled circuits: How to prove non-algebraic statements efficiently. In *ACM SIGSAC Conference on Computer & Communications Security*, pages 955–966, 2013.

[85] Jepsen. Analyses, 2020.

[86] Yael Kalai, Omer Paneth, and Lisa Yang. On publicly verifiable delegation from standard assumptions. Cryptology ePrint Archive, Report 2018/776, 2018. `https://eprint.iacr.org/`.

[87] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. H-Store: a high-performance, distributed main memory transaction processing system. *Proc. VLDB Endow.*, 1(2):1496–1499, 2008.

[88] Jonathan Katz, Rafail Ostrovsky, and Adam Smith. Round efficiency of multi-party computation with a dishonest majority. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 578–595, 2003.

[89] Tarandeep Kaur and Inderveer Chana. Energy efficiency techniques in cloud computing: A survey and taxonomy. *ACM Comput. Surv.*, 48(2), oct 2015.

[90] Issa M Khalil, Abdallah Khreishah, and Muhammad Azeem. Cloud computing security: A survey. *Computers*, 3(1):1–35, 2014.

[91] Joe Kilian. A note on efficient zero-knowledge proofs and arguments. In *Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*, pages 723–732, 1992.

[92] Kyle Kingsbury. Black-Box Serializability Verification, 2019.

[93] Kyle Kingsbury and Peter Alvaro. Elle: Inferring isolation anomalies from experimental observations. *Proc. VLDB Endow.*, 14(3):268–280, November 2020.

[94] W Kohler, A Shah, and F Raab. Overview of tpc benchmark c: The order-entry benchmark. *Transaction Processing Performance Council, Technical Report*, 1991.

[95] Ahmed Kosba, Charalampos Papamanthou, and Elaine Shi. xjsnark: A framework for efficient verifiable computation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 944–961. IEEE, 2018.

[96] Simeon Krastnikov, Florian Kerschbaum, and Douglas Stebila. Efficient oblivious database joins. *arXiv preprint arXiv:2003.09481*, 2020.

[97] SCIPR Lab. libsnark: a c++ library for zksnark proofs, 2020.

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

[99] Charles E. Leiserson. The Cilk++ concurrency platform. *Journal of Supercomputing*, 51(3):244–257, 2010.

[100] Charles E. Leiserson, Tao B. Schardl, and Jim Sukha. Deterministic parallel random-number generation for dynamic-multithreading platforms. In *PPoPP*, 2012.

[101] Jiangtao Li, Ninghui Li, and Rui Xue. Universal accumulators with efficient nonmembership proofs. In *International Conference on Applied Cryptography and Network Security*, pages 253–269. Springer, 2007.

[102] Linode. Recovering from a system compromise — Linode, 2022.

[103] Carsten Lund, Lance Fortnow, Howard Karloff, and Noam Nisan. Algebraic methods for interactive proof systems. *Journal of the ACM (JACM)*, 39(4):859–868, 1992.

[104] Carsten Lund, Lance Fortnow, Howard Karloff, and Noam Nisan. Algebraic methods for interactive proof systems. *J. ACM*, 1992.

[105] Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Toward a verified relational database management system. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 237–248, 2010.

[106] Toni Mastelic, Ariel Oleksiak, Holger Claussen, Ivona Brandic, Jean-Marc Pierson, and Athanasios V. Vasilakos. Cloud computing: Survey on energy efficiency. *ACM Comput. Surv.*, 47(2), dec 2014.

[107] Alfred Menezes, Palash Sarkar, and Shashank Singh. Challenges with assessing the impact of nfs advances on the security of pairing-based cryptography. In *International Conference on Cryptology in Malaysia*, pages 83–108. Springer, 2016.

[108] Ralph C Merkle. A digital signature based on a conventional encryption function. In *Conference on the Theory and Application of Cryptographic Techniques*, pages 369–378, 1987.

[109] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, mar 1992.

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

[111] Benjamin Mood, Debayan Gupta, Henry Carter, Kevin R.B. Butler, and Patrick Traynor. Frigate: A validated, extensible, and efficient compiler and interpreter for secure computation. In *IEEE European Symposium on Security and Privacy*, 2016.

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

[113] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. Deterministic galois: On-demand, portable and parameterless. In *ASPLOS*, 2014.

[114] Bernardo Nicoletti. *Cloud computing in financial services*. Springer, 2013.

[115] Alex Ozdemir, Riad Wahby, Barry Whitehat, and Dan Boneh. Scaling verifiable computation using efficient set accumulators. In *USENIX Security Symposium (USENIX Security 20)*, pages 2075–2092. USENIX Association, August 2020.

[116] Alex Ozdemir, Riad Wahby, Barry Whitehat, and Dan Boneh. Scaling verifiable computation using efficient set accumulators. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2075–2092, 2020.

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

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

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

[120] Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. CryptDB: Protecting Confidentiality with Encrypted Query Processing. `http://people.csail.mit.edu/nickolai/papers/raluca-cryptdb.pdf`, 2011.

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

[122] Kaushik Ravichandran, Ada Gavrilovska, and Santosh Pande. Destm: harnessing determinism in stms for application development. In *PACT*, pages 213–224, 2014.

[123] Noga Ron-Zewi and Ron D Rothblum. Proving as fast as computing: Succinct arguments with constant prover overhead. *Cryptology ePrint Archive*, 2021.

[124] Tom Schubert. High-level formal verification of next-generation microprocessors. In *Proceedings 2003. Design Automation Conference (IEEE Cat. No. 03CH37451)*, pages 1–6. IEEE, 2003.

[125] J. T. Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *J. ACM*, 1980.

[126] Srinath Setty. Spartan: Efficient and general-purpose zksnarks without trusted setup. In *Annual International Cryptology Conference*, 2020.

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

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

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

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

[131] Adi Shamir. Ip = pspace. *J. ACM*, 39(4):869–877, October 1992.

[132] Adi Shamir. Ip = pspace. *J. ACM*, 1992.

[133] Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. Brief announcement: The Problem Based Benchmark Suite. In *SPAA*, pages 68–70, 2012.

[134] Julian Shun, Yan Gu, Guy Blelloch, Jeremy Fineman, and Phillip Gibbons. Sequential random permutation, list contraction and tree contraction are highly parallel. In *SODA*, pages 431–448, 2015.

[135] Abraham Silberschatz, Henry F Korth, Shashank Sudarshan, et al. *Database system concepts*, volume 5. McGraw-Hill New York, 2002.

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

[137] Emin Gün Sirer. NoSQL meets bitcoin and brings down two exchanges: The story of Flexcoin and Poloniex, 2014.

[138] Espresso Systems. `Jellyfish` cryptographic library, 2022.

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

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

[141] Cheng Tan, Changgeng Zhao, Shuai Mu, and Michael Walfish. Making cloud key-value databases verifiably serializable. *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

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

[143] Justin Thaler. Time-optimal interactive proofs for circuit evaluation. In *IACR CRYPTO*, 2013.

[144] Justin Thaler, Mike Roberts, Michael Mitzenmacher, and Hanspeter Pfister. Verifiable computation with massively parallel interactive proofs. In *HotCloud*, 2012.

[145] Justin Thaler, Mike Roberts, Michael Mitzenmacher, and Hanspeter Pfister. Verifiable computation with massively parallel interactive proofs. In *USENIX HotCloud*, 2012.

[146] Alexander Thomson and Daniel J Abadi. The case for determinism in database systems. *VLDB*, 2010.

[147] Alexander Thomson and Daniel J Abadi. Building deterministic transaction processing systems without deterministic thread scheduling. In *WoDet*, 2011.

[148] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J Abadi. Calvin: fast distributed transactions for partitioned database systems. In *SIGMOD*, pages 1–12, 2012.

[149] Sergei Tikhomirov. Ethereum: state of knowledge and research perspectives. In *International Symposium on Foundations and Practice of Security*, pages 206–221. Springer, 2017.

[150] Alin Tomescu, Vivek Bhupatiraju, Dimitrios Papadopoulos, Charalampos Papamanthou, Nikos Triandopoulos, and Srinivas Devadas. Transparency logs via append-only authenticated dictionaries. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1299–1316, 2019.

[151] Alin Tomescu, Yu Xia, and Zachary Newman. Authenticated dictionaries with cross-incremental proof (dis) aggregation. *IACR Cryptol. ePrint Arch.*, 2020:1239, 2020.

[152] Mahdi Torabzadehkashi, Siavash Rezaei, Vladimir Alves, and Nader Bagherzadeh. Compstor: An in-storage computation platform for scalable distributed processing. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1260–1267, 2018.

[153] Tiago M Vale, João A Silva, Ricardo J Dias, and João M Lourenço. Pot: Deterministic transactional execution. *ACM TACO*, 2016.

[154] Victor Vu, Srinath Setty, Andrew J Blumberg, and Michael Walfish. A hybrid architecture for interactive verifiable computation. In *2013 IEEE Symposium on Security and Privacy*, pages 223–237, 2013.

[155] Riad S Wahby, Max Howald, Siddharth Garg, Abhi Shelat, and Michael Walfish. Verifiable asics. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 759–778, 2016.

[156] Riad S Wahby, Ye Ji, Andrew J Blumberg, Abhi Shelat, Justin Thaler, Michael Walfish, and Thomas Wies. Full accounting for verifiable outsourcing. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 2071–2086, 2017.

[157] Riad S. Wahby, Ye Ji, Andrew J. Blumberg, abhi shelat, Justin Thaler, Michael Walfish, and Thomas Wies. Full accounting for verifiable outsourcing. In *ACM CCS*, 2017.

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

[159] Riad S. Wahby, Ioana Tzialla, ahbi shelat, Justin Thaler, and Michael Walfish. Doubly-efficient zksnarks without trusted setup. In *IEEE Symposium on Security and Privacy*, 2018.

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

[161] Haixin Wang, Cheng Xu, Ce Zhang, and Jianliang Xu. vchain: A blockchain system ensuring query integrity. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 2693–2696, 2020.

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

[163] Darrell M West. *Saving money through cloud computing*. Governance Studies at Brookings, 2010.

[164] Yu Xia. Litmusdb preliminary build, 2022.

[165] Yu Xia, Xiangyao Yu, Matthew Butrovich, Andrew Pavlo, and Srinivas Devadas. Litmus: Towards a practical database management system with verifiable acid properties and transaction correctness. In *SIGMOD*, 2022.

[166] Yu Xia, Xiangyao Yu, William Moses, Julian Shun, and Srinivas Devadas. Litm: a lightweight deterministic software transactional memory system. In *Proceedings of the 10th International Workshop on Programming Models and Applications for Multicores and Manycores*, pages 1–10, 2019.

[167] Yu Xia, Xiangyao Yu, William Moses, Julian Shun, and Srinivas Devadas. 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)*, page 1–10, 2019.

[168] Yu Xia, Xiangyao Yu, Andrew Pavlo, and Srinivas Devadas. Taurus: lightweight parallel logging for in-memory database management systems. *Proceedings of the VLDB Endowment*, 14(2):189–201, 2020.

[169] Tiancheng Xie, Jiaheng Zhang, Yupeng Zhang, Charalampos Papamanthou, and Dawn Song. Libra: Succient zero-knowledge proofs with optimal prover computation. In *IACR CRYPTO*, 2019.

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

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

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

[173] Andrew C. Yao. Protocols for secure computations. In *23rd Annual Symposium on Foundations of Computer Science (sfcs 1982)*, pages 160–164. IEEE, 1982.

[174] Jing Yao, Yifeng Zheng, Yu Guo, and Cong Wang. Sok: A systematic study of attacks in efficient encrypted cloud data search. In *Proceedings of the 8th International Workshop on Security in Blockchain and Cloud Computing*, pages 14–20, 2020.

[175] Zuoxia Yu, Man Ho Au, Rupeng Yang, Junzuo Lai, and Qiuliang Xu. Lattice-based universal accumulator with nonmembership arguments. In *Australasian Conference on Information Security and Privacy*, pages 502–519. Springer, 2018.

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

[177] William Zhang and Yu Xia. Hydra: Pipelineable interactive arguments of knowledge for verifiable neural networks. In *2021 Third IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA)*, pages 1–10. IEEE, 2021.

[178] William Zhang and Yu Xia. Hydra: Succinct fully pipelineable interactive arguments of knowledge. Cryptology ePrint Archive, Report 2021/641, 2021. `https://eprint.iacr.org/2021/641`.

[179] Yupeng Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou. vsql: Verifying arbitrary sql queries over dynamic outsourced databases. In *IEEE Symposium on Security and Privacy*, 2017.

[180] Yupeng Zhang, D. Genkin, J. Katz, D. Papadopoulos, and C. Papamanthou. vram: Faster verifiable ram with program-independent preprocessing. In *IEEE Symposium on Security and Privacy*, 2018.

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

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

[183] Wenting Zheng, Stephen Tu, Eddie Kohler, and Barbara Liskov. Fast databases with fast durability and recovery through multicore parallelism. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 465–477, 2014.