PoneglyphDB: Efficient Non-interactive Zero-Knowledge Proofs for Arbitrary SQL-Query Verification

Binbin Gu
University of California, Irvine
binbing@uci.edu

Juncheng Fang
University of California, Irvine
junchf1@uci.edu

Faisal Nawab
University of California, Irvine
nawabf@uci.edu

Abstract

In database applications involving sensitive data, the dual imperatives of data confidentiality and provable (verifiable) query processing are important. This paper introduces PoneglyphDB, a database system that leverages non-interactive zero-knowledge proofs (ZKP) to support both confidentiality and provability.

In this paper, we propose efficient ZKP designs for basic operations in SQL query processing. PoneglyphDB's circuits are optimized for efficiency using PLONKish-based circuits and low-order polynomial constraints. We evaluate PoneglyphDB with the TPC-H benchmark, and our results show it efficiently achieves both confidentiality and provability, outperforming current state-of-the-art ZKP methods.

CCS Concepts

• **Do Not Use This Code → Generate the Correct Terms for Your Paper**; *Generate the Correct Terms for Your Paper*; Generate the Correct Terms for Your Paper; Generate the Correct Terms for Your Paper.

Keywords

Do, Not, Us, This, Code, Put, the, Correct, Terms, for, Your, Paper

ACM Reference Format:
Binbin Gu, Juncheng Fang, and Faisal Nawab. 2018. PoneglyphDB: Efficient Non-interactive Zero-Knowledge Proofs for Arbitrary SQL-Query Verification. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (Conference acronym 'XX).* ACM, New York, NY, USA, 14 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

As databases serve as the backbone of diverse applications, the responsibility entrusted to them extends beyond data storage—it encompasses the safeguarding of sensitive information. This is an important consideration, especially for databases that store personal data. Also, in applications with sensitive information such as census data or unemployment statistics, it is important for users to trust that the database owner is using the correct database (i.e., one that is consistent with the claimed database) and that the provided answers accurately reflect the intended computations.

We tackle the problem of providing a database solution to ensure two characteristics: **(1) Confidentiality**: this property means that raw data is only maintained at the host (i.e., service provider) and is not shared or made public to any other node. Other nodes only receive responses to specific queries that they send to the host node and that the host node agrees to process. **(2) Provability:** ensuring the correctness of processing. This means that the results computed by the host node and sent back to users reflect the correct processing of the query on the private database. This requires that the host node provides a proof of the correctness of the query response to the user.

As an example, consider the healthcare sector, where medical research institutions collect patient data for collaborative studies. Institution X may wish to share insights about the data with data consumers Y, Z, and W without disclosing the raw data. Data consumers send queries to X, where queries are processed and their responses sent back to the consumer. X has control on what queries to answer and therefore can control the confidentiality of data [12]. However, because data consumers do not have access to the raw database, they cannot verify the correctness of processing.

In this paper, we build on cryptographic solutions, namely *zero-knowledge proofs (ZKP)* [16, 20], to address the challenges of ensuring both confidentiality and provability in database query processing. ZKP constitutes a powerful cryptographic tool that enables one party, the prover, to convince another party, the verifier, of the correctness of a statement without revealing any specific information about the statement itself. The protocols used in ZKP systems can be made interactive and non-interactive ZKP. Interactive ZKP utlize cryptographic protocols wherein two entities, namely a *prover* and a *verifier*, dynamically exchange messages to establish the validity of a statement without revealing any sensitive information. Previous research has introduced interactive ZKP for verifying SQL queries [27, 41, 43]. The rationale behind using interactive ZKP for SQL query verification (like ZKSQL [27]) is that they enable the prover to engage with the verifier in multiple rounds, incrementally constructing and verifying parts of the proof. This step-by-step interaction often results in smaller circuit sizes, as the prover can break down the computation into manageable parts rather than generating a single large, complex proof upfront. However, the interactive nature introduces challenges related to synchronization and availability between the prover and verifier. Specifically, interactive ZKP are accompanied by two main drawbacks that currently limit their widespread adoption in practical applications: (1) They necessitate interaction between the prover and verifier throughout the proof protocol, imposing requirements for availability and synchronization between the involved parties. This challenge exacerbates failures and timeouts. (2) They lack transferability, as only the specific verifier(s) engaged in the original protocol possess the capability to verify the proof. This excludes the possibility of reusing (and caching) previously computed responses and proofs for multiple verifiers.

On the other hand, non-interactive ZKP is a cryptographic protocol class designed to establish the validity of a statement without the

Conference acronym 'XX, June 03–05, 2018, Woodstock, NY
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXXXXX.XXXXXXX

	Zero-knowledge	Non-interactive	Arbitrary SQL queries
IntegriDB [43]	✗	✗	✗
vSQL [41]	✗	✗	✓
vSQL+ [42]	✓	✗	N/A
ZKSQL [27]	✓	✗	✓
PoneglyphDB	✓	✓	✓

Table 1: Comparing cryptography-based methods for verifiable SQL queries.

necessity for a dynamic exchange of messages between the prover and verifier. Unlike interactive ZKP, non-interactive ZKP empowers a prover to generate a single, self-contained proof that can be subsequently verified by any party possessing the necessary verification key. Non-interactive ZKP, however, can experience significant overheads if the underlying cryptographic circuits are not carefully optimized.

In this paper, we propose a database system named PoneglyphDB, which incorporates non-interactive ZKP and the recursive proof composition technique. PoneglyphDB achieves both confidentiality and provability through non-interactive ZKP. PoneglyphDB differs from traditional databases by incorporating ZKP *circuits*, which are sets of equality constraints over arithmetic expressions designed to mimic the steps of query processing. These circuits enable ZKP frameworks to generate proofs of correctness for computations. In PoneglyphDB, we design circuits to represent basic query operations such as range checks, sorting, group-by, and joins, which are combined to handle more complex queries. Circuit design directly impacts proof generation performance. We optimize PoneglyphDB's circuits using the PLONKish framework [40], considering factors such as circuit size, polynomial degree, and batching construction. Additionally, PoneglyphDB leverages a recursive structure for composing proofs of multiple statements, reducing the overall proof size and computational overhead. This is made possible by advancements in proving systems that utilize recursive proof composition techniques [9, 25].

Table 1 compares PoneglyphDB and prior research on verifiable database systems [27, 41, 43] in terms of three properties: (1) the assurance of zero-knowledge (i.e., confidentiality), (2) non-interactive operability, and (3) applicability to arbitrary SQL queries. The protocols used in vSQL and vSQL+ are originally presented in an interactive manner, however they can be made non-interactive via the Fiat-Shamir heuristic [15].

To the best of our knowledge, PoneglyphDB is the first system that achieves all these desirable properties.

This paper is structured as follows: Section 2 presents background material. Section 3 proposes the general design of PoneglyphDB followed by the detailed design in Section 4. Section 5 presents our experimental evaluations. Section 6 presents related work and Section 7 concludes the paper.

2 Preliminaries

2.1 Zero-Knowledge Proofs

In the area of ZKP, a prover can convincingly demonstrate the truth of a given statement to a verifier without divulging any additional information. Specifically, ZKP empowers a prover $\mathcal{P}$ with a private witness w (a "private witness" refers to a piece of secret information such as a secret input, e.g., database) to validate the truth of a public statement $\mathcal{F}$ (with respect to w) to a verifier $\mathcal{V}$, while preserving the confidentiality of the underlying information w. For instance, suppose a database contains information about employees, and the data owner wants to prove to the verifier that the average salary of the employees is a certain number without disclosing individual salary details. In this scenario, the private witness w is the information about individual salaries, and the public statement $\mathcal{F}$ is the average salary computed by the SQL query.

In scenarios with multiple verifiers, such as healthcare settings where various institutions need to verify query results, interactive protocols can become impractical due to the need for multiple rounds of communication with each verifier. Non-interactive ZKP addresses this issue by allowing the prover to generate a single proof that can be verified by all parties without further interaction. For public-coin interactive protocols [20], the Fiat-Shamir heuristic [15] effectively converts them into non-interactive ones (e.g., vSQL), maintaining efficiency and minimizing computational overhead while supporting asynchronous verification.

2.2 Arithmetization

The use of arithmetic circuits is the most common paradigm for expressing computations within ZKP systems. We introduce the PLONKish arithmetization [40] that we use in PoneglyphDB. We emphasize that the PLONKish circuits used in PoneglyphDB serve primarily as a vehicle to demonstrate the potential of non-interactive ZK proofs within database management systems (DBMSs), rather than representing a state-of-the-art protocol.

PLONKish circuits are defined in terms of a rectangular matrix of values. We refer to rows, columns, and cells of this matrix with the conventional meanings. In the following, we introduce the key definitions of PLONKish circuits that are relevant to our circuit optimization.

1. **Fixed columns.** Fixed columns are fixed by the circuit. The values in the fixed columns are usually constants.
2. **Advice columns.** Advice columns correspond to *witness* values. These are private inputs and intermediate values generated during the circuit computation.
3. **Instance columns.** Instance columns are used for any elements shared between the prover and verifier. In most cases, they are used for public inputs and outputs.
4. **Equality constraints.** Equality constraints specify that two given cells must have equal values.
5. **Polynomial constraints.** For each row in the matrix, the multivariate polynomials over the field $\mathbb{F}$ must be evaluated to zero.

Example 2.1. Figure 1 illustrates the PLONKish circuit for calculating the function $f(x,y,z) = 3*(x+y)*z$. The circuit utilizes three advice columns—advice1, advice2, and advice3—to store the private inputs and intermediate values during computation.

In row 0, the values for x and y are put into the cells in advice1 and advice2, respectively (i.e. $x = a1$ and $y = b1$). A polynomial constraint is introduced to ensure that $c1 = a1 + b1$, calculating $x+y$. Next in row 1, the value of the input z (i.e. $z = a2$) is put into the cell in advice1. Since $a1+b1$ was computed as $c1$, its value is propagated to $b2$ and an equality constraint sets $b2 = c1$. A multiplier gate (with polynomial constraints) then calculates $c2 = a2 * b2$, evaluating $a2 * b2 - c2 = 0$. Finally in row 2, the constant 3 is brought into advice1 at the cell of $a3$ and fixed by the copy (or equality) constraint $a3 = 3$. The previous intermediate result $(a1+b1)*a2 = c2$ is copied into $b3$, with an equality constraint $b3 = c2$. The final output cell $c3$ calculates $a3*b3$, which evaluates to $3*(a1+b1)*a2$.

This result is copied into the public instance column at $o1$, allowing the verifier to read off the final output. Through a series of advice

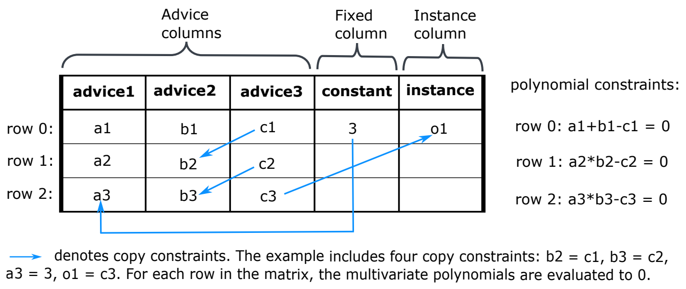


Figure 1: PLONKish circuits illustration for calculating the function $f(x,y,z) = 3*(x+y)*z$.

columns, equality constraints, and polynomial operations, the circuit computes the desired function in a modular fashion while keeping intermediate values private. The verifier only sees the final outputs revealed in the instance column.

To facilitate the modularity of PLONKish circuits, the designs of basic functions are represented as *gates*. A gate is a collection of columns and constraints that together implement a basic operation, such as division and multiplication. In our case, we will define gates for basic operations for query processing such as range checks and sorting. Gates can then be combined to implement more complex functions (or more complex gates).

3 System Overview

3.1 System Model

In PoneglyphDB, the prover $\mathcal{P}$ hosts a copy of a private database $\mathcal{DB}$. The prover receives queries from one or more verifiers, collectively denoted $\mathcal{V}$. The prover does not provide raw access to the data in the database. However, it answers queries sent by the verifiers.

3.2 Workflow Overview

PoneglyphDB operates in five key phases, as illustrated in Figure 2:

(1) Sending SQL Queries: The client, who will eventually assume the role of a verifier, sends SQL queries for execution against a private database, directly to the prover. The prover, holding exclusive access to this private database, is tasked with executing the queries.

(2) Circuit Construction: Upon receipt of the query request, the prover is then responsible for constructing the SQL queries and database commitment into arithmetic circuits. These circuits, delineated by gates and polynomial constraints, performs the computations that need to be proven. The SQL circuit encodes the desired SQL logic to be evaluated. This circuit allows for the computation of different inputs provided by the verifiers during the verification phase.

A database commitment is a cryptographic representation of a database state, enabling proof of properties about the data without revealing the data itself. This allows the prover to include evidence in the proof that the query was indeed processed on $\mathcal{DB}$. We employ the Inner Product Argument (IPA) protocol [8], operating over a 254-bit prime field, to generate this commitment. We choose the IPA protocol for the following reasons: (1) the proving time is typically linear with respect to the circuit size or the degree of the committed polynomial, (2) the proof size and verification time are logarithmic in the circuit size, due to the recursive structure of the inner product proof [9], and (3) it is compatible with PLONKish-based circuit designs.

By leveraging IPA, we can encode both the database commitment and the desired SQL logic within a unified framework [8, 10]. The

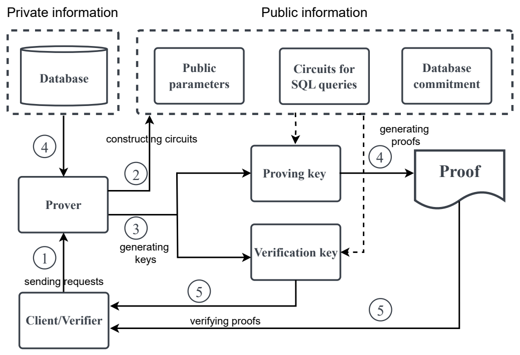


Figure 2: PoneglyphDB Workflow.

public parameters are a shared foundation for both provers and verifiers in proof creation and validation. This process utilizes publicly verifiable randomness and avoids the need for a trusted setup [13]. Initial proof creation uses only public information, forming the basis for subsequent proofs [9].

(3) Key Generation: Leveraging the public parameters and the circuit description, PoneglyphDB generates a proving key. This key is used to generate proofs corresponding to the circuit. Concurrently, a verification key is also produced, grounded in the same public parameters and circuit description. This verification key empowers verifiers to authenticate the proofs associated with this circuit.

(4) Proof Generation: The prover uses the proving key to generate a ZKP validating the correct computation of the SQL query over the private database. Leveraging the previously established database commitment, the prover creates a verifiable link between the committed database, query execution, and result. This process incorporates commitments to relevant database rows and intermediate computation steps, forming a chain of verifiable commitments from the initial database state to the final query output.

(5) Proof Verification: Finally, the verifier employs the public verification key to efficiently validate the ZKP received from the prover. The verification algorithm essentially interpolates the wiring polynomials and checks that all constraints are satisfied.

PoneglyphDB Positioning in the Workflow. In PoneglyphDB, we utilize existing work from the cryptography community to perform the steps of key and proof generation and verification. The main contribution of our work is in the design of SQL circuits that allow for efficient proving and verification given the utilized framework [38].

3.3 Application Framework Disuccsion

Trust Model. In the PoneglyphDB framework, the core parties involved are the database owner, acting as the prover, and the clients, acting as the verifiers. The prover and verifiers do not trust each other as we detail below. There is an additional entity, the auditor, that both the prover and verifier trust (there can be multiple auditors). The following are trust concerns between the parties:

1. **The prover may use a fake database:** The client does not trust the prover to be using the correct database to run the query. For example, a server responsible for running SQL queries might use a tampered or fake database to process a query and provide incorrect results to the verifier.
2. **The prover may process the query incorrectly:** Even if the correct database is used, the client cannot trust the accuracy of the

results returned by the prover. The server could mistakenly or intentionally alter or fabricate the query results.

3. **The verifier may attempt to leak data:** On the other hand, the prover fears that the client might attempt to extract or infer additional information about the underlying database beyond what is allowed by the query. The client could craft a series of queries to uncover sensitive patterns, private data, or proprietary information, breaching the prover's confidentiality.

We now map this trust model to a real-world scenario. In general, this trust model applies to cases when a database owner has sensitive data and wants to enable other entities (clients) to query the database without revealing data beyond what the database owner allows. In the healthcare scenario, a hospital H wants to provide query access to its database of patient data. It does not want to reveal all data, but would allow answering specific types of queries. The clients can be healthcare providers or research institutions that want to utilize H's database for their research. In this scenario, an auditor can be a government or regulatory entity that both H and clients trust. Next, we map the trust model assumptions onto this example:

1. The clients do not trust the hospital H to be using a correct database of patient's data: The hospital H does not reveal the raw database to the clients, and therefore clients cannot attest to the authenticity of the database used to answer queries.
2. The clients do not trust the hospital H to process queries on the database correctly: The hospital H may process the queries on the database incorrectly (e.g., by using approximation techniques to save costs, or by completely fabricating results).
3. The clients may attempt to leak data from query responses: The clients may want to know more information about the database of H than what is provided in the answer to the query.

To address these trust issues, PoneglyphDB introduces the following measures:

(1) Cryptographic Commitment to the Database: PoneglyphDB requires the prover to make an irrevocable cryptographic commitment to the database. This ensures that the prover cannot substitute the real database with a bogus one while still producing valid query results. The commitment is made public and shared in an irrevocable and immutable manner, e.g., by utilizing a decentralized blockchain such as Ethereum. This ensures that the prover will use the same database that corresponds to the database commitment as clients have access to an irrevocable, immutable database commitment that they can compare with the commitment used in the received proof. The database commitment can also be audited by a third-party auditor (e.g., a regulatory or government entity trusted by both the prover and verifiers). The auditor in this case reads the raw database from the database owner, verifies its authenticity, and validates that the database commitment of the authentic database corresponds to the commitment that is shared in the blockchain and accessible by the verifiers.

To prove the correctness of database updates and generate new commitments, a naive method would be to recompute the commitment for the entire database after each update. However, this approach is inefficient for large databases. A more advanced method might involve using a Merkle tree structure [28], where only the affected subtree is updated and recomputed, enabling efficient localized proof generation. Another potential method involves leveraging batch update techniques, where multiple updates are aggregated into a single proof.

(2) Ensuring Query Result Correctness: PoneglyphDB employs a constraint system that encodes SQL queries as circuits, ensuring that the verifier can confirm the query results are derived from a predefined and correct computation process. This process ensures the prover cannot return fabricated results.

(3) Preserving Data Privacy: To ensure the owner's database privacy, PoneglyphDB employs ZKP. These proofs guarantee that the client only receives the query result without extracting or inferring any additional information about the underlying database. The zero-knowledge property protects the database from unwanted data leakage beyond what is revealed from the query response. This zero-knowledge property can be combined with other techniques that prevent other types of leakage of private data. For this reason, it is important to distinguish between the types of leakage that are prevented or allowed by the zero-knowledge property and complement it with other techniques. A ZKP does not reveal information beyond the query response, but it does reveal what is part of the query response (and anything that can be implied by the query response). Therefore, if there is data that a user should not have access to but is part of the query response, then it is leaked to the client. To prevent this type of leakage, ZKP can be combined with techniques such as access control, data masking, query filtering, and policy management to prevent processing queries on data that the user would not have access to [1]. These techniques, typically applied as pre-processing steps, allow the prover to decide whether a query should be processed. For example, if the client's query asks to get raw data that should not be revealed, then the prover would not process the query. Query filtering, in particular, helps prevent leakage by modifying the query plan to exclude sensitive information before execution, ensuring that only authorized data is included in the response. Another type of leakage is to infer information about the database that were not intended to be revealed directly in the query response. This type of leakage can be prevented by incorporating differential privacy techniques as we discuss at the end of Section 3.4.

With these techniques, and reflecting back to the mapping to the real-world scenario above, we observe the following: (1) PoneglyphDB ensures that the hospital H uses an authentic database to process queries. This is by utilizing a trusted third-party auditor that verifies that the publicly shared irrevocable and immutable database commitment corresponds to an authentic database. If a prover attempts to utilize a different database, then the client is able to discover this as the client would match the publicly shared database commitment that is authenticated by an auditor with the database commitment that is used to process the query and received as part of the response. (2) PoneglyphDB ensures that hospital H processes the query correctly (according to the logic of the SQL query) by the ZKP constructions. (3) PoneglyphDB ensures that no information leakage occurs beyond what is revealed by the responses of hospital H. Hospital H can integrate further techniques—such as differential privacy—to ensure that other types of data leakage would not occur (as we describe in Section 3.4).

3.4 Security Model

PoneglyphDB leverages the Halo2 proving system [38], which incorporates well-established cryptographic properties such as completeness, soundness, knowledge soundness, and zero-knowledge.

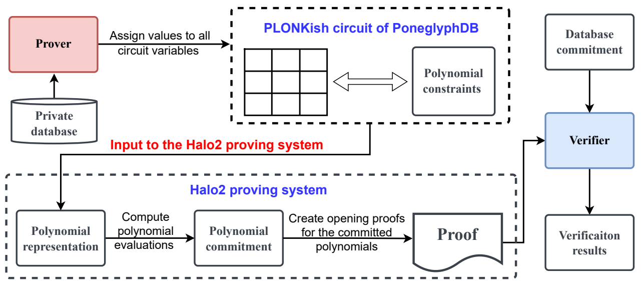

Prover
Assign values to all circuit variables
PLONKish circuit of PoneglyphDB
Polynomial constraints
Database commitment
Private database
Input to the Halo2 proving system
Verifier
Halo2 proving system
Polynomial representation
Compute polynomial evaluations
Polynomial commitment
Create opening proofs for the committed polynomials
Proof
Verificaiton results

Figure 3: Detailed components for generating ZK proofs.

- **Completeness.** If the prover can generate the PLONKish circuit (including the output) of a query, it can always convince the verifier that the PLONKish circuit of the query is true.
- **Soundness.** For any false PLONKish circuit (including any wrong witness, inputs and output), the probability of a dishonest prover successfully convincing an honest verifier is negligible.
- **Knowledge Soundness.** When the verifier is convinced the PLONKish circuit is correct, the prover actually possesses a valid witness.
- **Zero Knowledge.** The verifier learns only the information that can be inferred from the structure of the PLONKish circuit and the output of the query. No additional knowledge about the private witnesses or the database is revealed.

Figure 3 illustrates the detailed components involved in generating ZKP within PoneglyphDB. The system comprises two main parts: the PLONKish circuit and the Halo2 proving system. The PLONKish circuit serves as the input to the Halo2 proving system, which then generates ZKP for the circuit. Our primary contribution lies in the design of the PLONKish circuits, which are tailored to optimize the performance and efficiency of the ZKP generation process.

The PLONKish circuit represents the computation of a SQL query. This circuit is a mathematical representation of the logical operations and constraints involved in the SQL query execution. The prover, who possesses the private database, assigns values to all circuit variables based on the actual data from the database. This step involves mapping the data inputs to the corresponding variables in the circuit, ensuring that the computation is correctly set up for proof generation. The prover must use the database agreed upon with the verifier by utilizing the previously established database commitment.

The *Polynomial representation* component, provided by the Halo2 proving system, translates the circuit into a polynomial form. This component encodes the computation and its constraints as polynomial equations, making them suitable for ZKP.

The *Polynomial commitment* component, also provided by the Halo2 proving system, allows the prover to commit to the polynomial evaluations without revealing the actual polynomials. This ensures that the prover cannot alter the polynomials after the commitment, maintaining the integrity of the proof.

The *Halo2 proving system* takes the committed polynomials and generates opening proofs. These proofs are designed to show that the committed polynomials satisfy the polynomial constraints derived from the PLONKish circuit.

It is important to note that the PLONKish circuits implemented in PoneglyphDB are primarily intended to illustrate the feasibility of using non-interactive ZK proofs within DBMSs, rather than to claim they represent the latest advancements in ZK protocol design.

Guarantees. The correctness and security of PoneglyphDB depend on PLONKish circuits and the Halo2 proving system. we demonstrate the correctness of these circuits for SQL queries in Section 4.

Regarding the security of the Halo2 proving system used in PoneglyphDB , the prover and verifier engage in a non-interactive ZKP protocol utilizing Halo2's polynomial commitment scheme and recursive proof composition. The detailed security analysis of this protocol has been rigorously established in [9, 39]. We present a high-level summary of the correctness of the system below. To analyze the cryptographic protocols employed in Halo2, we utilize the Algebraic Group Model (AGM) [17]. The AGM is used to analyze protocols that rely on discrete logarithm assumptions in prime-order groups, a fundamental aspect of Halo2's design. The AGM evaluates the security of cryptographic protocols by requiring that adversaries must explicitly compute group elements from previously observed elements, emphasizing discrete logarithm-type assumptions in prime-order groups. PoneglyphDB guarantees completeness, soundness, knowledge soundness, and zero-knowledge properties under the AGM, assuming that all parties, including potential adversaries, are computationally bounded to probabilistic polynomial-time (PPT) algorithms. PoneglyphDB ensures that for any PPT adversary, there exists a PPT simulator such that, for any environment with arbitrary auxiliary input, the output distribution of the environment in a real-world execution (where a prover interacts with a verifier) is computationally indistinguishable from the output distribution in an ideal-world execution (where a simulator interacts with the verifier).

Oblivious circuits. The proving logic of the circuits in PoneglyphDB is designed to be oblivious. This means that the execution of the proving algorithm does not depend on the specific values of the private inputs (the witness). Instead, the prover performs the same operations regardless of the actual witness values, which helps ensure that no information about the private inputs is leaked through the proof generation process or the resulting proof itself. For example, when implementing a sorting algorithm, the circuit would compare and swap elements in a fixed pattern regardless of their actual values, ensuring the same operations are performed for any input. Similarly, for conditional statements, both branches of the condition are typically evaluated, and their results are combined using a selector value, rather than following only one branch based on the private condition. To protect the privacy of table cardinalities and intermediate result sizes in join operations, we adopt the method introduced in ZKSQL [27]. This approach involves introducing dummy tuples into our query evaluation process, effectively obscuring true data sizes while maintaining consistent row counts throughout query execution.

3.5 Data Privacy Issues and Limitations

`PoneglyphDB` sends the query results to the clients. Since the proof inherently includes the result of the query, a client could, for example, issue a query like "SELECT * FROM T" to attempt to retrieve all values from T, which threatens data privacy. Beyond the direct data exposure of returned results, sensitive data can also be exposed indirectly. Specifically, when data points or records in the database are correlated, query results may still leak sensitive information about records that are not part of the response. For instance, correlation among records—such as similar attributes shared across groups or statistical dependencies—can allow a client to infer additional, sensitive details beyond the returned query results. In this way, even without direct access to records that are not included in query results, clients can potentially exploit patterns in the query response to gain insights into the broader dataset indirectly. This poses a data privacy risk that needs to be handled carefully.

To address this issue, differential privacy techniques [14, 23] could be employed to ensure that individual data points are not revealed either directly or indirectly. In this work, we do not incorporate differential privacy and leave it as an avenue of future work. We note, however, that methods of incorporating differential privacy to ZKP systems such as PoneglyphDB would lead to additional overhead in the circuit design.

4 Custom Gates

In this section, we present customized gates to represent SQL queries arithmetization in circuits. Our goal is to introduce efficient designs by creating gates that have low-degree polynomials and a smaller number of circuit constraints. We design with low-degree polynomials because ZKP relies on cryptographic primitives where evaluating higher-degree polynomials is computationally expensive.

4.1 Range Check

We first introduce a range check gate as it is involved in many SQL operations like **"filter"**, **"sort"**, **"group by"** and **"join"**.

Consider the range check statement $x \le t$, where x is a private value and t is the public query input. A naive encoding compares x against each possible value using the polynomial equation: $\prod_{i=0}^{t}(x-i)=0$. The degree of this polynomial grows linearly with t, making proof generation and verification computationally infeasible for large t. In this work, we leverage a *lookup table* [18] circuit structure to design the range check gate. The intuition behind using lookup tables lies in the idea of precomputing and storing results for a range of possible inputs. Our work builds upon the widely used Plookup framework [18], which is well-established in the ZKP domain. While there are alternatives, such as Jolt [32], we leave exploration of these methods for future work. Instead of reinventing cryptographic primitives, we focus on solving concrete implementation challenges to enhance efficiency of SQL operations within the PLONK framework. While previous works [27, 41] implement operations like filter, sort, and join using boolean or logic gate frameworks, our approach utilizes arithmetic-based PLONKish circuits. This fundamental difference introduces significant challenges, as arithmetic circuits require different optimization strategies compared to boolean circuits. Adapting SQL operators in this context necessitates a rethinking of their expression and optimization to achieve maximum efficiency.

Definition. The input to a range check gate, which checks if elements are between y_1 and y_m, consists of a column C_{in} containing n elements $\{x_1,x_2,...,x_n\}$ to be range-checked, and a column (lookup table) T containing m sorted values $\{y_1,y_2,...,y_m\}$ representing the valid range, where $y_1 < y_2 < \cdots < y_m$. The output of the range check gate is a column C_{out} containing n elements $\{z_1,z_2,...,z_n\}$, where each z_i indicates whether x_i is within the range defined by the lookup table T. Specifically, $z_i = 1$ if $x_i \in \{y_1,y_2,...,y_m\}$, and $z_i = 0$ otherwise.

Design A: A single range check. We start with a simple case where we only want to prove that a single value x is in a specific range $[0,t]$, hence proving that $0 \le x \le t$. In the first step of constructing the circuit, we create a private array P (stored in an advice column) with the same length of set Q where the first element in P is x and the other elements are any values copied from Q (these values can be duplicates). Then, the prover supplies a permutation of P, denoted by P' where P' is private and stored in an advice column in the circuit. The values in P' are sorted so that duplicate values are row-adjacent to each other.

In the second step, we establish a fixed column to store the set Q, arranging the values in ascending order. This organization ensures that both the prover and the verifier are aware of the values and their corresponding indices within the table. Such knowledge is crucial for determining the size of the lookup table needed.

The prover supplies a permutation of Q, denoted by Q', where Q' is private and stored in an advice column in the PLONKish circuit. The purpose of the permutation Q' is to hide the position of values in Q as the verifier knows all the information about Q. Then, the circuit can compare values in Q' and P' to check whether x (in P') is equal to some value in Q', without revealing which value is that in Q'.

Now, we show how we can check whether the value x in P' is equal to a value in Q'. The values in Q' are arranged in a specific order such that either $P'_i = Q'_i$ or that $P'_i = P'_{i-1}$ where P'_i and Q'_i represent the i-th elements of P' and Q' respectively. Specifically, we enforce that the first values of P and Q are equal, i.e, $P'_i = Q'_i$ with $i=0$. If $P'_i \neq Q'_i$ for $i>0$, we enforce that $P'_i = P'_{i-1}$ meaning that P'_i must be a duplicate of P'_{i-1} in this case. Therefore, these constraints guarantee that every value in P'_i is equal to some value in Q'. Formally, we enforce that either $P'_i = Q'_i$ or that $P'_i = P'_{i-1}$, using the rule:

$$0 = \begin{cases} (P'_i - Q'_i)\cdot(P'_i - P'_{i-1}) & \text{if } 1 \le i \le len(Q')-1, \\ P'_i - Q'_i & \text{if } x = 0. \end{cases} \tag{1}$$

With the above polynomial constraints, the verifier knows that all the values of P' are in Q' without knowing the position information of the values in P' and Q' (therefore it does not know the exact values at each position in P', preserving the privacy of the x value).

Since P' and Q' are permutations of P and Q, the polynomial constraints above ensure that all the values of P (and P') are in Q. To ensure this property, we develop polynomial constraints to ensure that both P' is a permutation of P and Q' is a permutation of Q:

$$\prod_{i=0}^{len(Q)-1} (P_i+\alpha)(Q_i+\beta) = \prod_{i=0}^{len(Q)-1} (P'_i+\alpha)(Q'_i+\beta) \tag{2}$$

where P_i, Q_i, P'_i and Q'_i represent the i–th element in P, Q, P', and Q' respectively, and α and β are randomly chosen parameters. The random values α and β conceal the contents of the columns, ensuring confidentiality during verification and preventing zero products from causing collisions due to poor randomness. To make the circuit formulation efficient, we express it as a recursive function to ensure that each equation maintains a low polynomial degree:

$$\begin{gathered} Z_{i+1} = Z_i \cdot \frac{(P_i+\alpha)(Q_i+\beta)}{(P'_i+\alpha)(Q'_i+\beta)} \\ Z_{\text{len}(Q)} = Z_0 = 1 \end{gathered} \tag{3}$$

Example 4.1. Figure 4 illustrates our proposed lookup table circuit design for proving a range check statement without revealing the specific values of x_1 and x_2. The prover uses a fixed column Q accessible to verifiers, containing values in the range $[0,4)$. An advice column P includes the actual values of x_1 and x_2 with the remaining cells filled arbitrarily from Q. The prover then generates advice columns P' and

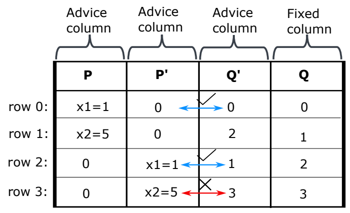


Figure 4: Range check with lookup tables illustration.

Q' as permutations of P and Q, respectively, with P' sorted in ascending order and duplicates adjacent, ensuring Q' starts with the same value as P'. The prover checks that each P'_i is either equal to Q'_i or P'_{i-1}. This polynomial constraint ensures that values are within the range.

Design B: Batching range check. The lookup table technique facilitates verifying whether all elements of an array P belong to the set $Q = \{i \mid 0 \le i \le t, i \in \mathbb{Z}\}$, effectively checking if they fall within the range $[0,t]$. By organizing the elements of P into a single advice column and aligning the set Q within a fixed column, the lookup table approach previously discussed can be applied. For instance, the circuit depicted in Figure 4 can demonstrate whether the elements in the array $[0,2,1,3]$ (column Q') are contained within the range $[0,4)$ (column Q). The complexity of verifying the inclusion of the arrays $[0]$ and $[0,1,2,3]$ within the range $[0,4)$ remains consistent, as it necessitates the application of formulas 1 and 3 for each row.

Design C: Optimizing range checks with bitwise decomposition and lookup tables. To mitigate the scalability concerns associated with range checks, especially when the size of the set Q becomes significantly large (e.g., 2^{64}), we propose an optimization technique that leverages bitwise decomposition in conjunction with lookup tables. This method entails representing integers in a fixed bit-length format, such as 64 bits, and subsequently verifying the integrity of this bitwise representation in relation to the original integer value.

Given an integer N and its representation as a sequence of bits $b_0,b_1,...,b_{k-1}$, where b_0 is the least significant bit and b_{k-1} the most significant, the relationship between N and its bits is described by $N = \sum_{i=0}^{k-1} b_i \cdot 2^i$. We partition the bitwise representation of integer N into smaller segments of 8 bits, referred to as u8 cell. This approach is predicated on the standard binary representation of integers, wherein the integer is segmented into 8-bit blocks. For instance, a 64-bit integer is divided into 8 u8 cells, each encapsulating an 8-bit slice of the integer. Constraints are imposed to ensure this decomposition is accurately executed with the constraints $N = \sum_{i=0}^{7} c_i \cdot 2^{8i}$ for a 64-bit integer N, where c_i represents the 8-bit segment of N at position i. Each u8 cell c_i should have values within the range 0 to 255 (inclusive), corresponding to $2^8 - 1$. To validate each u8 cell, we utilize a fixed-sized lookup table of size 256, which contains integers from 0 to 255. This table allows efficient range checking for each segment of the integer. The advantage of using this lookup table is that it is fixed-sized (256 entries) and can be reused multiple times for each u8 cell check. This ensures that each u8 cell c_i can quickly verify if its value falls within the allowed range of 0 to $2^8 - 1$. By reusing the lookup table across all 8 u8 cells of N, we streamline the validation process and ensure consistent range checks.

Design D: Conditional statements proving. The previous range check method faces limitations when a value falls outside the lookup range, making proof construction difficult. To this end, we introduce an augmented method that seamlessly integrates with lookup tables to facilitate range checks while gracefully handling conditional scenarios. When dealing with data filtering, an upper-bound value, denoted as u, typically exists. Consequently, for $x \ge t$, this also implies $u > x \ge t$. Therefore, proving $u > x \ge t$ is equivalent to proving $0 \le x - t \le u$. To establish $x < t$, it suffices to demonstrate that $x - t < 0$. By adding u to both sides of the inequality, we obtain $0 \le x - t + u < u$. Introducing a binary variable, denoted as $check$, to determine whether $x < t$, we ultimately prove the following statement:

$$0 \le (x-t) + \text{check} \cdot u < \text{u} \tag{4}$$

To accomplish this, the prover configures several supporting columns in the PLONKish circuits. Initially, an advice column is created to output 1 if $x < t$ and 0 otherwise. Additionally, another advice column is established to store the values of $x - t$. Finally, the prover undertakes the task of proving that check$+(x-t)$ falls within the range of $[0,u)$ with the assistance of the lookup tables introduced in Section 4.1.

Note that the values in the **"check"** columns are prover-determined, with no explicit constraints imposed among x, t, and **"check"**. However, if the **"check"** values are inaccurately provided, the proof generation process encounters a failure. Moreover, the values in the **"x"**, **"t"**, and **"x-t"** columns adhere to the constraints $cell_i(x) - cell_i(t) - cell_i(x-t) = 0$ for the initial four rows, where $cell_i(x)$ represents the value in the **"x"** column at row i. These constraints guarantee that the discrepancies between corresponding elements in the **"x"** and **"t"** columns align with the values stored in the **"x-t"** column.

Correctness. We prove the correctness of the range check gate from two aspects. (1) Property 1: Element Inclusion. All the values in P' are in Q'. Assume that there exists one value x in P' that is not in Q'. We will show that this assumption leads to a contradiction. If x is the first value in P', by Equation (1), $0 = P'_0 - Q'_0$, implying $P'_0 = Q'_0$. Thus, $x = P'_0$ contradicts $x \notin Q'$. If x is not the first, for $1 \le i \le \text{len}(Q') - 1$, $0 = (P'_i - Q'_i)(P'_i - P'_{i-1})$ enforces $P'_i = Q'_i$ or $P'_i = P'_{i-1}$. If $P'_i = Q'_i$, $x \in Q'$; if $P'_i = P'_{i-1}$, x duplicates P'_{i-1}, tracing back to $P'_0 = Q'_0$. Thus, x must be in Q', contradicting $x \notin Q'$. Hence, all P' values are in Q', proving Equation (1)'s constraints ensure $P' \subseteq Q'$. (2) Property 2: Permutation Integrity. Assume P' and Q' are not permutations of P and Q. This implies that some P_i or Q_i does not match any corresponding P'_i or Q'_i. Since the products involve symmetric polynomials, if $\{P'_i + \alpha\} \cup \{Q'_i + \beta\}$ are not permutations of $\{P_i + \alpha\} \cup \{Q_i + \beta\}$, the two sides of the equation cannot be equal due to the unique factorization of polynomials. Therefore, by the Fundamental Theorem of Symmetric Polynomials, P' and Q' must be permutations of P and Q.

With the two properties proven above, we can guarantee the correctness of proving $x < t$ if x is not larger than t. The inequality in Equation 4 holds if $check = 1$ and $x < t$, or if $check = 0$ and $x \le t$. Therefore, setting the binary variable check correctly is sufficient to determine whether $x < t$ or not. By transforming x to $(x-t) + \text{check} \cdot u$, we can guarantee that the transformed x is in the range of 0 to u.

Complexity of a Range Check Gate. The ZK proof generation cost depends on the number of constraints. We analyze the complexity of a gate by specifying the required number of each type of constraint. To validate that all elements of an array P reside within a set Q, the number of constraints, as defined in Equations 1 and 3, corresponds to the greater of $|P|$ or $|Q|$, denoted as $\max(|P|,|Q|)$. When employing the bitwise decomposition method for an input set P where $|P| > 256$, the lookup table is padded to match the size of the input by including duplicates of values ranging from 0 to 255. For 64-bit integers decomposed into 8 u8 cells, each cell undergoes a range check utilizing the lookup table. The number of constraints, as described by Equations 1 and 3, is equivalent to $8|P|$. Additionally, the number of constraints required to ensure the correct decomposition of integers into u8 cells, as well as verifying that all segments of the integer fall within the specified ranges, is $|P|$. Finally, to transform a value x with $x' = (x-t) + \text{check} \cdot u$ (see Equation 4), an additional $|P|$ constraints are needed.

4.2 Sort

We detail our approach to proving the correctness of sort operations.

Definition. The input of a sort gate consists of a table D with m columns $C_1, C_2, ..., C_m$, where C_i represents different attributes or values to be sorted. Each column C_i contains n elements $\{x_{i1}, x_{i2}, ..., x_{in}\}$. The output is a table SD with m columns $C'_1, C'_2, ..., C'_m$, where each column C'_i contains the sorted elements $\{y_{i1}, y_{i2}, ..., y_{in}\}$. Here, y_{ij} represents the j-th element in column C'_i of D, arranged in the desired order (e.g. increasing or decreasing).

Design. The first step in proving sort operations entails generating a witness, which includes the result of applying a specific sorting algorithm to the input data. The prover has the flexibility to choose any sorting algorithm, as long as the resulting order is correct. Let D and R represent the input data and the sorted result, respectively. Two essential properties of R must be guaranteed for a successful proof. First, the data in R should match that in D except for the order, leading to a permutation check between R and D. The following constraints, akin to Equations (3), ensure this:

$$\begin{aligned} Z_{i+1} &= Z_i \cdot \frac{R_i + \alpha}{D_i + \alpha} \\ Z_{\text{len}(D)} &= Z_0 = 1 \end{aligned} \tag{5}$$

Here, D_i and R_i represent the i-th element in D and R, and α is a randomly chosen parameter similar to ones in Equation 2.

In sorting mechanisms where multiple attributes are considered, a unified approach is adopted to encapsulate these attributes into a singular composite entity. This is achieved by allocating a consistent bit-length representation for each attribute, specifically employing a 64-bit format for this purpose. Such a fixed bit-length representation is critical in preserving the intrinsic value hierarchy and relative ordering of each attribute during the process of concatenation. In addition, the data in R must align with the sort definition. To verify this, we check that $R_i \le R_{i+1}$ (assuming an ascending order) for i in the range $[0, \text{len}(R)-1)$. This is achieved by proving the transformed statement introduced in Equation 4 with the assistance of lookup tables.

Correctness. We prove the correctness of the two properties introduced in the sort gate. (1) Property 1: Permutation Integrity. R is a permutation of D. The proof follows a similar approach used to prove that P' is a permutation of P. For details, refer to the proof in Section 4.1. (2) Property 2: Sortedness. R is sorted in ascending order. Assume, for contradiction, that R is not sorted in ascending order. This implies there exists at least one pair of indices $i < j$ such that $R_i > R_j$. Since we enforce the constraints $R_i \le R_{i+1}$ for each pair (R_i, R_{i+1}) where $i \in [0, \text{len}(R)-1)$, and the correctness of the range check gate for $R_i \le R_{i+1}$ is proven in Section 4.1, we guarantee that $R_i \le R_{i+1}$ holds for all valid indices i. This contradicts the assumption that there exists at least one pair of indices $i < j$ such that $R_i > R_j$. Therefore, R must indeed be sorted in ascending order to satisfy the defined properties and constraints of the sort operation.

Complexity of a sort gate. Permutation checks between D and R require $|D|$ constraints, as per Equation 5. Additionally, checking $R_i \le R_{i+1}$ for $i \in [0, \text{len}(R)-1)$ requires constraints proportional to $|D|$, similar to the range check gate in Section 4.1.

4.3 Group-by

In this section, we outline our methodology for verifying the correctness of group-by operations.

Definition. The input of a group-by gate consists of a table D with n columns $C_1, C_2, ..., C_n$, where each column represents different attributes of D. The output is a table SD with $n+2$ columns that rearranges the records of D such that records with identical values on the grouping attributes G are placed into the same group-by bin. Alongside the original n columns from D, SD includes two additional columns: (1) start_index: Indicates the starting index of each group-by bin in D. (2) end_index: Indicates the ending index of each group-by bin in D. The start_index and end_index are used for the later aggregation functions such as SUM and others.

Design. Given an input table D and the group-by attributes G, the prover first generates the sorted table SD based on the group-by attributes G. This sorting ensures that records with identical values in G are adjacent in SD. To verify that SD is a sorted version of D, we employ the approach introduced in Section 4.2.

To identify the starting and ending indices of each group-by bin, we check each record in SD on the group-by attributes G. We use a binary value b to indicate whether a record in SD is a starting or ending record, where 1 signifies that the record is either a starting or ending record. The constraint to check whether two values v_1 and v_2 are equal or not with a binary value b is as follows:

$$b = 1 - (v_1 - v_2) \cdot p \tag{6}$$

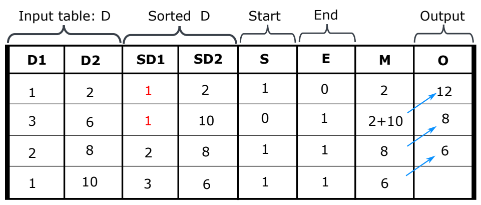
where p is the value provided by the prover. Specifically, $p = 0$ if $v_1 = v_2$ and $p = \frac{1}{v_1 - v_2}$ otherwise. To ensure that the prover provides the correct value of p, we add the following constraint for each pair of v_1 and v_2:

$$b \cdot (v_1 - v_2) = 0 \tag{7}$$

A record is marked as the start of its bin if no preceding adjacent records share identical values in group-by attributes G. Conversely, it is marked as the end of its bin if no subsequent adjacent records share identical values in G.

Correctness. We utilize the sort gate introduced in Section 4.2 to ensure that SD is sorted. The correctness of the sort gate is demonstrated in Section 4.2. Next, we deduce the correctness of the starting and ending indices of each group-by bin. According to Equation 6, if $v_1 = v_2$, then $b = 1$, and Equation 7 holds trivially as $b \cdot (v_1 - v_2) = 0$. If $v_1 \ne v_2$: (1) If $p = \frac{1}{v_1 - v_2}$, then $b = 0$, and Equation 7 holds because $b \cdot (v_1 - v_2) = 0$. (2) If $p = 0$, then $b = 1$. In this case, Equation 7 does not hold as $b \cdot (v_1 - v_2) = (v_1 - v_2) \ne 0$. Consequently, a valid proof cannot be generated because the system detects the inconsistency.

Complexity of a group-by gate. The group-by operation is facilitated through the sorting of associated attributes, followed by a verification of the accuracy of sorting. Consequently, the computational complexity and the number of constraints required for ensuring the correctness of group-by operations are identical to those identified in the analysis of sorting constraints. Additionally, the number of constraints (Equations 6 and 7) needed to identify the starting and ending indices of each group-by bin is $2|D|$ where D is the input set.



Input table: D		Sorted D		Start	End		Output
D1	**D2**	**SD1**	**SD2**	**S**	**E**	**M**	**O**
1	2	1	2	1	0	2	12
3	6	1	10	0	1	2+10	8
2	8	2	8	1	1	8	6
1	10	3	6	1	1	6	

Figure 5: Illustration with the SQL query "SELECT SUM(D2) FROM T GROUP BY D1.".

Example 4.2. Figure 5 illustrates the SQL query "SELECT SUM(D2) FROM **T** GROUP BY D1." The input table has columns **D1** and **D2**. The prover adds sorted columns **SD1** and **SD2**, as well as columns S and **E** for group-by bin indices. An advice column **M** is created where M_i is defined as $M_{i-1}+SD2_i$ if $SD1_i = SD1_{i-1}$, otherwise $M_i = SD2_i$, with $M_0 = SD2_0$. The final output column **O** captures the result by copying only the last record of each group-by bin, as indicated by the **E** column.

4.4 Join

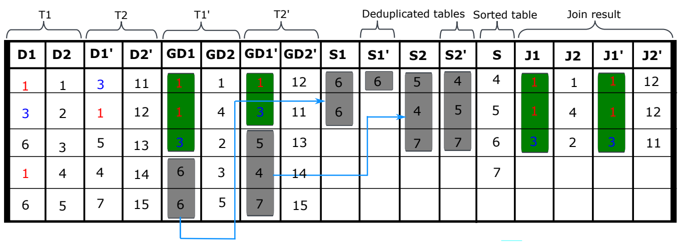
Now, we present our methodology for verifying the correctness of join operations.

Definition. The input of a join gate consists of two tables $T1$ and $T2$, each represented by m columns $C_1^{(1)}, C_2^{(1)}, \ldots, C_m^{(1)}$ for $T1$ and n columns $C_1^{(2)}, C_2^{(2)}, \ldots, C_n^{(2)}$ for $T2$. These columns correspond to different attributes of $T1$ and $T2$, respectively. The joining operation is performed based on equality conditions between specified joining attributes $\mathcal{J}1$ from $T1$ and $\mathcal{J}2$ from $T2$, which may involve any of the columns $C_i^{(1)}$ from $T1$ and $C_j^{(2)}$ from $T2$. The output is a table JD that combines records from $T1$ and $T2$ where the joining condition $T1.\mathcal{J}1 = T2.\mathcal{J}2$ is satisfied. The table JD has $m+n$ columns: (1) The first m columns $C_1^{(1)}, C_2^{(1)}, \ldots, C_m^{(1)}$ are the attributes from $T1$. (2) The next n columns $C_1^{(2)}, C_2^{(2)}, \ldots, C_n^{(2)}$ are the attributes from $T2$.

Design. Consider two private tables $T1$ and $T2$, and let p be the join predicate. To establish the correctness of the join operations, the prover calculates several types of witnesses locally. Initially, the prover creates two new tables $T'1$ and $T'2$ to reorder the records within them. Specifically, each table $T1'$ and $T2'$ is split into two parts: $T1'_p$ and $T2'_p$ contain records contributing to the join predicate, while $T1'_{non-p}$ and $T2'_{non-p}$ contain records that do not contribute to the join predicate. The correctness of this reordering is verified using polynomial constraints, as detailed in Equation 5.

Next, the prover ensures that the records in $T1'$ and $T2'$ indeed contribute to the join predicate. In the context of primary key-foreign key joins (we discuss non-primary key-foreign key joins later), instead of checking the existence of records in one table in another, the prover proves that $T1'_{non-p}$ is disjoint from $T2'_{non-p}$. Traditional straightforward methods involve checking each value in $T1'_{non-p}$ against $T2'_{non-p}$, resulting in a large number of polynomial constraints. To address this, a sorted table S is created to store the unique values in $T1'_{non-p}$ and $T2'_{non-p}$, with a proof of $S_i < S_{i+1}$ for $i \in [0, \text{len}(S)-1)$.

However, duplicates [1] in S pose challenges in determining their origin (from the same or different tables). To resolve this, the prover



T1		T2		T1'		T2'			Deduplicated tables			Sorted table	Join result			
D1	**D2**	**D1'**	**D2'**	**GD1**	**GD2**	**GD1'**	**GD2'**	**S1**	**S1'**	**S2**	**S2'**	**S**	**J1**	**J2**	**J1'**	**J2'**
1	1	3	11	1	1	1	12	6	6	5	4	4	1	1	1	12
3	2	1	12	1	4	3	11	6		4	5	5	1	4	1	12
6	3	5	13	3	2	5	13			7	7	6	3	2	3	11
1	4	4	14	6	3	4	14					7				
6	5	7	15	6	5	7	15									

Figure 6: Illustration with the SQL query "SELECT T1.D1, T2.D2' FROM T1, T2 WHERE T1.D1 = T2.D1' ".

implements a deduplication strategy through the creation of distinct versions, namely $T1^{de}$ and $T2^{de}$. This process ensures that each value in $T1'non-p$ is accounted for in $T1^{de}$ and every value in $T2'non-p$ is accounted for in $T2^{de}$. To establish this deduplication property, we leverage the mechanism of lookup tables we proposed in Section 4.1. The verification of the range check operation, which confirms the existence of each value in a column within a lookup table (i.e. another column), is adapted to ensure the deduplication of $T1^{de}$ and $T2^{de}$. It is noteworthy that the distinction between $T1^{de}$, $T2^{de}$, and a lookup table for range check lies in their roles within the PLONKish circuit configuration. Specifically, $T1^{de}$ and $T2^{de}$ serve as `advice columns`, signifying the privacy of the data they contain, while a lookup table for range check is stored in an `instance column`, designating the public nature of the data within it. This distinction is integral to the overall architecture of the PLONKish circuit configuration. The prover establishes that S is a permutation of $T1^{de} \cup T2^{de}$, ensuring that S matches the records in $T1^{de}$ and $T2^{de}$ except for the order. And $S_i < S_{i+1}$ holds for $\forall i \in [0, len(S)-1)$.

Next, we introduce the method for generating join results from contributing records. Consider the scenario where the join predicate between two tables, $T1$ and $T2$, is defined as $T1.\text{attr1} = T2.\text{attr2}$. In the context of primary key-foreign key joins, the uniqueness of primary keys implies the absence of duplicates, ensuring that each foreign key in $T1$ corresponds to at most one matching row in $T2$. Let us denote $T1'_p$ and $T2'_p$ as the subsets of $T1$ and $T2$, respectively, that are relevant to the join predicate. Within $T1$, attr1 serves as the foreign key, whereas attr2 is the primary key within $T2$. For each record in $T1'_p$, a search is conducted within $T2'_p$ to identify any record that satisfies the join predicate. Upon finding a match, the record from $T2'_p$ is concatenated with the corresponding record from $T1'_p$, forming a combined record. To verify the join results, two key properties must be established: (1) **Equality Verification**: For each concatenated record r, we set the polynomial $r.\text{attr1} - r.\text{attr2} = 0$, where attr1 and attr2 are the attributes from $T1$ and $T2$, respectively, used for the join. (2) **Source Verification**: To ensure all records joined with $T1'_p$ originate exclusively from $T2'_p$, we use lookup tables as described in Section 4.1.

To handle joins without relying on primary key-foreign key relationships, we compute the join result by pairwise comparing records from $T1'_p$ and $T2'_p$ using a similar procedure. However, the method introduced above may not include all records contributing to the join predicate in $T1'$ and $T2'$. To ensure completeness, we additionally prove that $T1'_{non-p}$ is disjoint from $T2'$ and $T2'_{non-p}$ is disjoint from $T1'$ using the same deduplication and sorting mechanism. This ensures that all relevant records are considered in the join operation.

[1] Since SQL operates on multisets, duplicates are not only a concern in join algorithms but also throughout the entire query process, as the result must accurately reflect the correct number of duplicates. In Plookup [18], this can be efficiently managed by encoding the frequency of elements in the multiset into polynomial commitments. Plookup then ensures that the correct number of occurrences for each value is preserved during query execution, maintaining the semantic integrity of SQL operations.

Scalability. When managing a large number of join operations, the database's query engine typically executes these joins sequentially, joining two tables at a time. This results in various execution plans having differing numbers of total and intermediate join results. Optimizations aimed at reducing the number of these total and intermediate join results should be identified and applied prior to the circuit design phase.

Correctness. To ensure the correctness of the join gate, we verify the following properties: (1) Property 1: Permutation Integrity. The sets $T1'_p \cup T1'_{non-p}$ and $T2'_p \cup T2'_{non-p}$ are permutations of the original tables $T1$ and $T2$, respectively. The proof follows a similar approach used to prove that P' is a permutation of P. For details, refer to the proof in Section 4.1. (2) Property 2: Completeness. $T1'$ and $T2'$ include all records that contribute to the join predicate. Assume $T1'$ misses a record x that contributes to the join predicate, i.e., x is in $T1'_{non-p}$. We prove that $T1'_{non-p}$ is disjoint from $T2'$ and $T2'_{non-p}$ using the Element Inclusion and Sortedness properties as detailed in Section 4.2. This implies x does not overlap with $T2'$, indicating that x does not contribute to the join predicate. This contradicts the assumption, proving that $T1'$ includes all relevant records. Similarly, $T2'$ includes all records that contribute to the join predicate. (3) Property 3: Exclusivity. $T1'$ and $T2'$ do not include records that do not contribute to the join predicate. Assume $T1'$ includes a record y that does not satisfy the join condition. During the join process, the concatenated record r involving y would fail the constraint $r.\text{attr1} - r.\text{attr2} = 0$, contradicting the assumption. Therefore, $T1'$ and $T2'$ only include records that contribute to the join predicate. The correctness of the above three properties ensures the correctness of the join gate.

Example 4.3. Figure 6 illustrates the SQL query "SELECT T1.D1, T2.D2 FROM T1, T2 WHERE T1.D1 = T2.D1'." Upon receiving the input tables $T1$ and $T2$, the prover creates new tables $T1'$ and $T2'$. The upper parts contain records contributing to the join $T1.D1 = T2.D1'$. The green area in columns $GD1$ and $GD1'$ shows corresponding values. The prover verifies non-contributing records (gray areas) in $GD1$ and $GD1'$ do not intersect by creating columns $S1$ and $S2$, eliminating duplicates, and ensuring each value in $S1$ and $S2$ exists in their respective sorted columns $S1'$ and $S2'$. A column S is constructed by sorting $S1'$ and $S2'$. The prover checks S is a permutation of $S1' \cup S2'$ and that $S_i < S_{i+1}$ for all i. The join results, as depicted in the last four columns, are derived from the green-highlighted values in $GD1$ and $GD1'$.

Complexity of a join gate. Given two tables $T1$ and $T2$ with the number of records denoted by $T1_{\text{num}}$ and $T2_{\text{num}}$ respectively, we categorize records that contribute to the join predicate as $T1^{\text{join}}$ and $T2^{\text{join}}$, and those that do not contribute as $T1^{\text{disjoin}}$ and $T2^{\text{disjoin}}$. The counts of these records are denoted as $T1_{\text{join\_num}}$, $T2_{\text{join\_num}}$, $T1_{\text{disjoin\_num}}$, and $T2_{\text{disjoin\_num}}$ respectively. We omit the copy constraints in this analysis as they are lightweight. A permutation or range check gate, sized at X, implies that X corresponding constraints are applied across two columns, each populated with X values. The computation of constraints for a join operation encompasses five distinct categories:

- Two permutation check gates with sizes $T1_{\text{num}}$ and $T2_{\text{num}}$.
- Two range check constraints with lookup tables (referred to by Equations 1 and 3) in sizes $T1_{\text{disjoin\_num}}$ and $T2_{\text{disjoin\_num}}$ for the deduplication process of $T1^{\text{disjoin}}$ and $T2^{\text{disjoin}}$.
- One range check constraint with lookup tables (referred to by Equations 1 and 3) in size $T1_{\text{disjoin\_num}} + T2_{\text{disjoin\_num}}$ (in the worst case) for sorting the deduplicated versions of $T1^{\text{disjoin}}$ and $T2^{\text{disjoin}}$.
- The number of equality check constraints (in the form $x - y = 0$) for checking if corresponding values satisfy the join predicate, with a maximum of either $T1_{\text{disjoin}}$ or $T2_{\text{disjoin}}$, for columns such as $J1$ and $J1'$ in the given figure.
- One range check constraint with lookup tables (referred to by Equations 1 and 3) in the size of $\max(T1_{\text{disjoin}}, T2_{\text{disjoin}})$ to ensure all records joined with T_2^{join} originate exclusively from T_2^{join}.

4.5 Aggregation and Other Operations

Since aggregation operations are often applied together with the group-by operation, we describe how to implement them in conjunction with group-by. The SUM gate is implemented by establishing a column that holds intermediate, non-final values for each group-by bin, as described in Figure 5 and Example 4.2. To identify the starting and ending indices of each group-by bin, we can follow the method introduced in the Group-by section 4.3. Once we determine the indices of these boundary records for each group-by bin, we can employ a similar approach to implement the COUNT gate. With the SUM and COUNT values determined for each bin, the AVERAGE gate can be naturally realized through a division gate that processes these values.

Furthermore, the MAX and MIN gates are facilitated by a sorting mechanism. By arranging the values in ascending order, the smallest and largest values, corresponding to the MIN and MAX gates respectively, can be directly identified as the first and last values in the sorted list. Along with these functionalities, we have implemented additional aggregate functions such as Standard Deviation, Variance, and Median. Additionally, we have developed capabilities for string matching and concatenation by validating the equality of sub-strings in two strings using lookup tables.

For projection operation, we use selectors to project the desired columns by setting them to 1 for inclusion and 0 for exclusion. Each selector controls a multiplication gate, multiplying the column by 1 or 0 based on whether it is part of the projection.

The set operations can be implemented using the methods described for the join gate. Set equality is handled by first sorting both tables and then comparing tuples at each index. Set disjointness is checked by sorting both tables and ensuring that any consecutive tuples R_i and R_{i+1} in the sorted list satisfy $R_i \leq R_{i+1}$. Set intersection is achieved as illustrated in Example 4.3 and Figure 6, where the join method is applied to extract common tuples between tables R and S. For set union, tuples that are common to both R and S (found via set equality) are first removed from R, and the remaining tuples are then concatenated to S.

We have covered the most common operations used in SQL queries. Other variations of these operations can be constructed using the methods introduced in this work, as long as they can be represented within the circuit framework.

4.6 Combining Gates

PoneglyphDB processes full SQL queries by combining customized gates for different operators as follows:

1. **Mapping Operations to Gates:** Each SQL query operation, such as sorting or joining, is represented by a corresponding gate. This

gate executes the specific operation, ensuring accurate relationships between inputs and outputs.

2. **Predefined Execution Plan and Assembly:** The SQL query's predefined execution plan outlines the sequence and dependencies of operations, guiding the assembly of gates in sequence. Each gate's output serves as the input for the next, ensuring data flows correctly through the circuit according to the optimized plan.
3. **Combining Gates:** Multiple gates are combined to handle various operations as outlined in the execution plan. The gates are strategically assembled in sequence, with the output of one gate serving as the input for the next.

Since each operator (such as sorting or aggregation) is verified separately, proving each ensures the correctness of the entire query. However, even when all inputs appear in the output, as with sorting, there is a risk of data leakage from intermediate steps like comparisons. This is because intermediate steps, like comparing the relative order of data elements, may reveal unintended patterns or relationships. This is why we implement oblivious circuits: to ensure that no information about intermediate steps, such as comparisons or data order, is exposed during proof generation.

Correctness. Let G_1,G_2,...,G_n be a sequence of gates processing a query. Assume G_1 receives correctly transformed input and operates correctly, yielding correct output O_1. Assume for each G_i (where $1 \leq i \leq n$), the output O_i is correct and serves as input to G_{i+1}. Given each gate's correctness, G_{i+1} operates correctly on O_i to produce O_{i+1}. By induction, the final output O_n produced by the last gate G_n is correct.

5 EXPERIMENTAL RESULTS

We evaluate PoneglyphDB in terms of proving and verification time, memory usage, operator performance, proof size and scalability.

5.1 Experimental Setup

We implement PoneglyphDB's circuits and gates using Halo2, a state-of-the-art ZKP system [38]. The implementation of all Plonkish circuits for the queries is conducted using Rust. Our evaluation of PoneglyphDB focuses on a selected subset of the TPC-H benchmark [35], specifically targeting queries that are representative of common data analytics workloads.

We compare with ZKSQL [27], a state-of-the-art solution for using interactive ZKP in database systems.

The interactivity in ZKSQL involves breaking down the computation into smaller sub-circuits to reduce the complexity of the overall circuit. These sub-circuits are then verified interactively, where the prover demonstrates the correctness of each sub-circuit step-by-step. The interaction ensures that the combined outputs of the sub-circuits correspond to the correct execution of the SQL query, allowing the verifier to check the correctness of the entire query process incrementally. However, this interactivity increases the communication and computational overhead, as each round of interaction requires multiple exchanges between the prover and verifier.

To align with existing research evaluations of ZKP-based databases and to conduct a fair comparison with ZKSQL, we have implemented the six TPC-H queries identified in their evaluation: Q1, Q3, Q5, Q8, Q9, and Q18.

In addition, we compare our system with Libra [36], a state-of-the-art non-interactive ZKP system that leverages the GKR protocol [19], which is also foundational to vSQL [41]. To the best of our knowledge, Libra is the most efficient publicly available system utilizing the GKR protocol. Since the high-level logic for implementing SQL operations in vSQL can be adapted across various ZKP systems, we use Libra's circuit structure to implement SQL operations based on the logic and optimizations introduced in vSQL. While Libra uses a fixed input structure, we adopt an alternative approach to avoid the need for relay gates for inputs that may not be needed immediately. That is, we split the circuit into multiple parts, ensuring that each part includes only the necessary inputs in its input layer.

Table 2: Running time (in seconds) for generating public parameters with different maximal number of rows in Plonkish circuits.

Maximal number of rows	2^{15}	2^{16}	2^{17}	2^{18}
Running time (s)	104	221	410	832

Table 3: Running time (in seconds) of database commitment over data of increasing sizes.

The size of the database	$60k$ *Rows*	$120k$ *Rows*	$240k$ *Rows*
Running time (s)	2.89	5.53	10.94

In our experimentation, we adhere to the same query variables wherever applicable, such as the `orderdate` filter, to maintain consistency and relevance in our results. For query Q9, similar to ZKSQL's approach, we exclude string pattern-matching predicates from our evaluation. We converted all floating point operations to 64-bit integer ones in our experiments similar to ZKSQL.

Our experimental setup quantifies the database scale by the size of the central fact table, `lineitem`, and scales the dimension tables proportionally, as described in the TPC-H benchmark specifications. We report results across three database sizes—60k Rows, 120k Rows, and 240k Rows—with the `lineitem` table containing 60k, 120k, and 240k rows, respectively. These varying sizes provide insights into the scalability and practicality of PoneglyphDB in handling verifiable databases across different volumes of data. Unless otherwise mentioned, our experiments run on 60k Rows.

Our experiments are conducted on Chameleon Cloud [24] using a Skylake node, equipped with two Intel Xeon Skylake CPUs running at 2.60 GHz, 192 GB of RAM, and 10 Gigabit Ethernet connectivity.

5.2 Setup

PoneglyphDB eliminates the need for a trusted setup process. Instead, PoneglyphDB utilizes public parameters. These parameters are essential for both constructing and verifying proofs and are known to all parties involved—the prover and the verifier alike. Importantly, these parameters are not confidential and require no secrecy.

Table 2 details the running time associated with generating these public parameters. It's worth noting that the generation of these parameters is a one-time process; once created, they can be stored and reused indefinitely. The versatility of these parameters allows for their application across various circuits, provided the number of rows in the circuit does not surpass the maximum capacity defined by the public parameters. Consequently, the time spent generating these parameters is not considered part of the cost associated with generating SQL query proofs in this work.

Running time of database commitment. The proof generation for a fixed database commitment can be done once and be reused for SQL queries that are applied on the database. Table 3 shows the running time of committing to the 8 TPC-H tables.

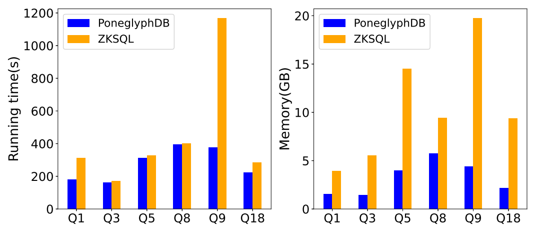


Figure 7: Running time (left figure) and memory usage (right figure) for generating SQL queries proofs.

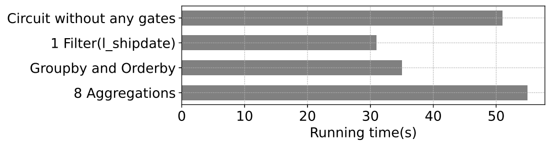


Figure 8: PoneglyphDB's performance breakdown of different proof generation steps for Q1.

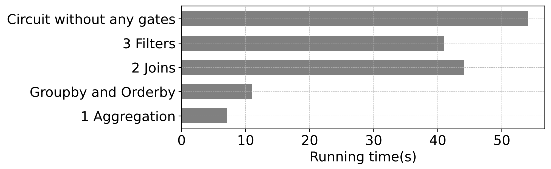


Figure 9: PoneglyphDB's performance breakdown of different proof generation steps for Q3.

5.3 Benchmarking with ZKSQL

We compare the running time of generating proofs for six SQL queries in PoneglyphDB with that of ZKSQL. The results (the left figure of Figure 7), reveal that PoneglyphDB—although a non-interactive ZKP solution—achieves performance that is similar to the interactive ZKP solution ZKSQL for most queries. In fact, PoneglyphDB outperforms ZKSQL significantly—by at least up to 40%—for queries Q1 and Q9. This difference is attributed to the relatively fewer range check (or filtering) and sort operations required in Q1 and Q9. PoneglyphDB utilizes arithmetic circuits for handling range checks and sorting operations. Despite the use of lookup tables to optimize the degrees of polynomial constraints and reduce the circuit size, arithmetic circuits can become more complex than boolean circuits, which ZKSQL employs for filtering and sorting operations, especially as the range of data increases. Nevertheless, PoneglyphDB exhibits enhanced performance in join operations, which necessitate arithmetic expressions to represent polynomial constraints accurately. Figure 10 (right) shows the memory usage for generating proofs for the six SQL queries in PoneglyphDB and ZKSQL. PoneglyphDB uses significantly less memory, ranging from 23% to 60% of ZKSQL's usage.

5.4 Benchmarking with Libra

Since Libra is a non-interactive ZKP system, we benchmarked against Libra in terms of three critical factors: proving time, verification time, and proof size. As shown in Table 4, Libra requires more proving time than PoneglyphDB. In Libra, complex operations such as sorting require a large number of basic gates, with each gate limited to two inputs, which increases both the circuit depth and size. For comparison operations in SQL queries, decimal values are represented using full 64-bit binary representations in Libra. Logical operations on these 64-bit binary numbers necessitate circuits that handle each bit individually, including managing carry bits across the entire bit width. This bitwise processing, along with the overhead of transforming binary values to decimal for subsequent arithmetic operations, results in significantly larger circuits. This increased circuit size leads to longer proving times. In contrast, PoneglyphDB optimizes the handling of decimal values by segmenting them into 8-bit chunks and leveraging lookup tables to efficiently validate and perform operations on each segment. The larger circuit size in Libra not only increases the proving time but also leads to longer verification times and larger proof sizes, as shown in Table 4 for queries Q1, Q3, and Q5.

Table 4: Benchmarking against Libra in terms of proving time, verification time and proof size.

		Proving time (in seconds)	Verification time (in seconds)	Proof size (in kilobytes)
Libra	Q1	812	1.290	435.8
	Q3	997	1.212	411.4
	Q5	1021	1.227	413.9
PoneglyphDB	Q1	180	0.617	8.6
	Q3	161	0.725	24.7
	Q5	313	0.739	29.6

5.5 Operation Performance

To enhance our understanding of PoneglyphDB's performance, we assess the overheads associated with the steps involved in proof generation. Figures 8 and 9 show a breakdown of the execution time to generate proofs for queries Q1 and Q3, respectively. We selected these two queries for our performance evaluation because they encompass a comprehensive range of SQL operations, including multiple aggregations, joins, group-by, sort, and filtering functions.

The proof generation process begins with the construction of a comprehensive circuit, encapsulating all facets of witness generation; this preliminary phase is described as a "circuit without any gates". Following this, the procedure advances to the integration of polynomial constraints—or gates—that correspond to the SQL operations. These operations include filtering, grouping by, ordering, and performing eight aggregations for query Q1, as well as applying three filters, executing two joins, a group-by, an order-by, and an aggregation for query Q3.

The results show that the initial step takes over 50 seconds, attributable to the fixed overheads determined by the chosen public parameter; a larger public parameter size increases this initial step overhead. The significant overhead in proof generation, notably from the aggregations in Q1 and the filters and joins in Q3, can be attributed to the extensive computational resources required. Aggregation operations, for instance, necessitate the collation and computation across sizable datasets to yield a singular summary outcome. This task demands multiple iterations of data processing and polynomial constraints verification within the circuit, thereby amplifying its complexity and extending the duration needed for proof generation. Similarly, filters and joins in Q3 are computationally intensive, as filtering requires checking each record against conditions, while joins involve aligning records from different tables based on join keys.

5.6 Scalability

We evaluate the scalibility of PoneglyphDB by generating proofs with larger workloads. We evaluate with the workload of TPC-H's

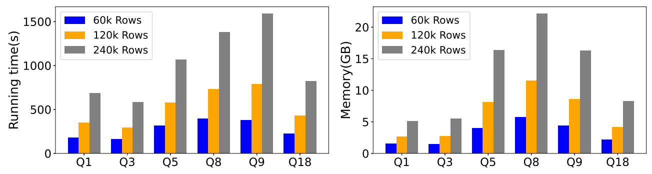


Figure 10: Proof generation time and memory usage over data of increasing sizes.

lineitem table at 120k and 240k rows as the lineitem table dominates the complexity of the SQL queries. As depicted in Figure 10, the running time and memory consumption increases with the increase in the size of the dataset. Specifically, the running times for the six SQL queries—Q1, Q3, Q5, Q8, Q9, and Q18—exhibit a gradual increase as the database size increases from 60k to 240k Rows.

The running time for query Q1 starts at 180 seconds for 60k rows and increases to 683 seconds for the 240k row dataset, indicating a proportional relationship between dataset size and observed performance. This is because the size of our circuits linearly grows with the size of inputs and all the polynomial constraints enforced on the circuits have low degrees. This validates our design goal of maintaining low-degree polynomials in PoneglyphDB's circuits. Memory usage shows a similar pattern, with the memory footprint for query Q1 starting at 1.53 GB for 60k rows and increasing to 5.12 GB for 240k rows.

6 Related Work

There is substantial research into verifiable SQL querying, employing a variety of techniques that ensure the integrity and security of query results. These methods can be broadly categorized into three groups: Authenticated Data Structures (ADS) [34], Trusted Execution Environments (TEE) [31], and Cryptographic Proof Techniques [16].

ADS-based methods use asymmetric cryptography to authenticate data, requiring extra memory to maintain authenticated data structures for SQL query verification [4, 29, 30, 37, 44]. While secure, these methods are generally limited to specific computational tasks.

TEE-based approaches, exemplified by [3, 4, 33, 45, 46], secure SQL query results through computations performed within trusted hardware environments. Basic TEE implementations might expose sensitive data through program traces. Integrating TEE with Oblivious Random Access Machine (ORAM), as in [22], can obscure such traces but at the cost of additional computation time, highlighted in [2, 26].

Cryptographic proof techniques, such as zk-SNARKs [6] and zk-STARKs [5], enable entities to verify the correctness of computations without revealing underlying data. These techniques ensure high levels of security but are resource-intensive, requiring substantial memory to manage numerous intermediate values and considerable time to create proofs.

Prior cryptographic proof systems, such as IntegriDB [43] and vSQL [41], employ cryptographic verifiable computation to validate a wide range of SQL queries. While IntegriDB and vSQL ensure data integrity, they operate in an outsourcing model and do not inherently provide zero-knowledge properties. An extension of vSQL, referred to as vSQL+ [42], introduces ZKP; however, it lacks support for ad-hoc queries and does not thoroughly address practical efficiency or the translation of arbitrary SQL statements into cryptographic protocols necessary for such guarantees. Notably, vSQL and vSQL+ are based on public-coin protocols [20, 21], which can be transformed into non-interactive ZKP systems using the Fiat-Shamir heuristic [15].

ZKSQL [27] reduces the proving cost by dividing the entire circuit into smaller sub-circuits to reduce the size of the overall circuit. This approach supports ad-hoc queries and maintains zero-knowledge properties, but it shares the common limitations of interactive ZKP. ZKSQL is based on designated-verifier protocols [21], where the Fiat-Shamir heuristic cannot generally be applied to transform the protocol into a non-interactive proof.

Our system, PoneglyphDB, generates non-interactive ZKP using recursive proof composition [7, 9, 11, 25]. It enhances proof generation performance by optimizing arithmetic circuits.

7 Conclusion

We introduce PoneglyphDB, a non-interactive ZKP-based database designed for efficient confidentiality and provability. PoneglyphDB optimizes proof generation through recursive methods and tailored designs for SQL queries, showing competitive or superior performance in TPC-H benchmarks compared to existing solutions.

References

[1] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. 2002. Hippocratic databases. In *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*. Elsevier, 143–154.

[2] AKM Mubashwir Alam, Justin Boyce, and Keke Chen. 2023. SGX-MR-Prot: Efficient and Developer-Friendly Access-Pattern Protection in Trusted Execution Environments. In *2023 IEEE 43rd International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 1029–1032.

[3] Arvind Arasu, Ken Eguro, Raghav Kaushik, Donald Kossmann, Pingfan Meng, Vineet Pandey, and Ravi Ramamurthy. 2017. Concerto: A high concurrency key-value store with integrity. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 251–266.

[4] Sumeet Bajaj and Radu Sion. 2013. CorrectDB: SQL engine with practical query authentication. *Proceedings of the VLDB Endowment* 6, 7 (2013), 529–540.

[5] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. 2019. Scalable zero knowledge with no trusted setup. In *Advances in Cryptology–CRYPTO 2019: 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18–22, 2019, Proceedings, Part III 39*. Springer, 701–732.

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

[7] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. 2017. Scalable zero knowledge via cycles of elliptic curves. *Algorithmica* 79, 4 (2017), 1102–1160.

[8] Jonathan Bootle, Andrea Cerulli, Pyrros Chaidos, Jens Groth, and Christophe Petit. 2016. Efficient zero-knowledge arguments for arithmetic circuits in the discrete log setting. In *Advances in Cryptology–EUROCRYPT 2016: 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II 35*. Springer, 327–357.

[9] Sean Bowe, Jack Grigg, and Daira Hopwood. 2019. Recursive proof composition without a trusted setup. *Cryptology ePrint Archive* (2019).

[10] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. 2018. Bulletproofs: Short proofs for confidential transactions and more. In *2018 IEEE symposium on security and privacy (SP)*. IEEE, 315–334.

[11] Benedikt Bünz, Alessandro Chiesa, Pratyush Mishra, and Nicholas Spooner. 2020. Recursive proof composition from accumulation schemes. In *Theory of Cryptography: 18th International Conference, TCC 2020, Durham, NC, USA, November 16–19, 2020, Proceedings, Part II 18*. Springer, 1–18.

[12] Ji-Won Byun and Ninghui Li. 2008. Purpose based access control for privacy protection in relational database systems. *The VLDB Journal* 17 (2008), 603–619.

[13] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Noah Vesely, and Nicholas Ward. 2020. Marlin: Preprocessing zkSNARKs with universal and updatable SRS. In *Advances in Cryptology–EUROCRYPT 2020: 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10–14, 2020, Proceedings, Part I 39*. Springer, 738–768.

[14] Cynthia Dwork. 2006. Differential privacy. In *International colloquium on automata, languages, and programming*. Springer, 1–12.

[15] Amos Fiat and Adi Shamir. 1986. How to prove yourself: Practical solutions to identification and signature problems. In *Conference on the theory and application of cryptographic techniques*. Springer, 186–194.

[16] Uriel Fiege, Amos Fiat, and Adi Shamir. 1987. Zero knowledge proofs of identity. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*. 210–217.

[17] Georg Fuchsbauer, Eike Kiltz, and Julian Loss. 2018. The algebraic group model and its applications. In *Advances in Cryptology–CRYPTO 2018: 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19–23, 2018, Proceedings, Part II 38*. Springer, 33–62.

[18] Ariel Gabizon and Zachary J Williamson. 2020. plookup: A simplified polynomial protocol for lookup tables. *Cryptology ePrint Archive* (2020).

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

[20] S. Goldwasser, S. Micali, and C. Rackoff. 1985. The Knowledge Complexity of Interactive Proof-Systems. In *Proceedings of the 17th Annual ACM Symposium on Theory of Computing, May 6-8, 1985, Providence, Rhode Island, USA*.

[21] Shafi Goldwasser and Michael Sipser. 1986. Private coins versus public coins in interactive proof systems. In *Proceedings of the eighteenth annual ACM symposium on Theory of computing*. 59–68.

[22] Michael T Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. 2012. Privacy-preserving group data access via stateless oblivious RAM simulation. In *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms*. SIAM, 157–167.

[23] Noah Johnson, Joseph P Near, and Dawn Song. 2018. Towards practical differential privacy for SQL queries. *Proceedings of the VLDB Endowment* 11, 5 (2018), 526–539.

[24] Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Colleran, Haryadi S. Gunawi, Cody Hammock, Joe Mambretti, Alexander Barnes, François Halbach, Alex Rocha, and Joe Stubbs. 2020. Lessons Learned from the Chameleon Testbed. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX Association.

[25] Abhiram Kothapalli, Srinath Setty, and Ioanna Tzialla. 2022. Nova: Recursive zero-knowledge arguments from folding schemes. In *Annual International Cryptology Conference*. Springer, 359–388.

[26] Duc V Le, Lizzy Tengana Hurtado, Adil Ahmad, Mohsen Minaei, Byoungyoung Lee, and Aniket Kate. 2020. A tale of two trees: one writes, and other reads. *Proceedings on Privacy Enhancing Technologies* (2020).

[27] Xiling Li, Chenkai Weng, Yongxin Xu, Xiao Wang, and Jennie Rogers. 2023. ZKSQL: Verifiable and Efficient Query Evaluation with Zero-Knowledge Proofs. *Proceedings of the VLDB Endowment* 16, 8 (2023), 1804–1816.

[28] Ralph C. Merkle. 1980. Protocols for Public Key Cryptosystems. In *Proceedings of the 1980 IEEE Symposium on Security and Privacy, Oakland, California, USA, April 14-16, 1980*. IEEE Computer Society, 122–134. https://doi.org/10.1109/SP.1980.10006

[29] Dimitrios Papadopoulos, Stavros Papadopoulos, and Nikos Triandopoulos. 2014. Taking authenticated range queries to arbitrary dimensions. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. 819–830.

[30] Dimitrios Papadopoulos, Charalampos Papamanthou, Roberto Tamassia, and Nikos Triandopoulos. 2015. Practical authenticated pattern matching with optimal proof size. *Proceedings of the VLDB Endowment* 8, 7 (2015), 750–761.

[31] Mohamed Sabt, Mohammed Achemlal, and Abdelmadjid Bouabdallah. 2015. Trusted execution environment: what it is, and what it is not. In *2015 IEEE Trustcom/BigDataSE/Ispa*, Vol. 1. IEEE, 57–64.

[32] Srinath Setty, Justin Thaler, and Riad Wahby. 2024. Unlocking the lookup singularity with Lasso. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 180–209.

[33] Rohit Sinha and Mihai Christodorescu. 2018. Veritasdb: High throughput key-value store with integrity. *Cryptology ePrint Archive* (2018).

[34] Roberto Tamassia. 2003. Authenticated data structures. In *Algorithms-ESA 2003: 11th Annual European Symposium, Budapest, Hungary, September 16-19, 2003. Proceedings 11*. Springer, 2–5.

[35] Transaction Processing Council. 2023. TPC-H Benchmark. http://www.tpc.org/tpch/. Accessed: 2023-xx-xx.

[36] Tiacheng Xie, Jiaheng Zhang, Yupeng Zhang, Charalampos Papamanthou, and Dawn Song. 2019. Libra: Succinct zero-knowledge proofs with optimal prover computation. In *Advances in Cryptology–CRYPTO 2019: 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18–22, 2019, Proceedings, Part III 39*. Springer, 733–764.

[37] Yin Yang, Dimitris Papadias, Stavros Papadopoulos, and Panos Kalnis. 2009. Authenticated join processing in outsourced databases. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. 5–18.

[38] Zcash. [n.d.]. halo2. https://github.com/zcash/halo2

[39] Zcash. [n.d.]. *Halo2 Protocol*. https://zcash.github.io/halo2/design/protocol.html

[40] Zcash. [n.d.]. *PLONKish Arithmetization*. https://zcash.github.io/halo2/concepts/arithmetization.html

[41] Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. 2017. vSQL: Verifying arbitrary SQL queries over dynamic outsourced databases. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 863–880.

[42] Yupeng Zhang, Daniel Genkin, Jonathan Katz, Dimitrios Papadopoulos, and Charalampos Papamanthou. 2017. A zero-knowledge version of vSQL. *Cryptology ePrint Archive* (2017).

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

[44] Qingji Zheng, Shouhuai Xu, and Giuseppe Ateniese. 2012. Efficient query integrity for outsourced dynamic databases. In *Proceedings of the 2012 ACM Workshop on Cloud computing security workshop*. 71–82.

[45] Wenting Zheng, Ankur Dave, Jethro G Beekman, Raluca Ada Popa, Joseph E Gonzalez, and Ion Stoica. 2017. Opaque: An oblivious and encrypted distributed analytics platform. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 283–298.

[46] Wenchao Zhou, Yifan Cai, Yanqing Peng, Sheng Wang, Ke Ma, and Feifei Li. 2021. Veridb: An sgx-based verifiable database. In *Proceedings of the 2021 International Conference on Management of Data*. 2182–2194.