How SMPC Query Execution can be sped up through Efficient and Flexible Intermediate Result Size Trimming

Long Gu
Systems Group, TU Darmstadt, Germany
long.gu@tu-darmstadt.de

Shaza Zeitouni
Systems Group, TU Darmstadt, Germany
shaza.zeitouni@tu-darmstadt.de

Carsten Binnig
Systems Group, TU Darmstadt, Germany
carsten.binnig@cs.tu-darmstadt.de

Zsolt István
Systems Group, TU Darmstadt, Germany
zsolt.istvan@tu-darmstadt.de

ABSTRACT

There is growing interest in Secure Collaborative Analytics, but fully oblivious query execution in Secure Multi-Party Computation (MPC) settings is prohibitively expensive. Recent related works proposed different approaches to trimming the size of intermediate results between oblivious query operators, resulting in significant speedups at the cost of some controlled information leakage. In Reflex, we generalize these ideas into a flexible and efficient trimming method for the output of the oblivious operators, that we call Resizer. Resizers can be seamlessly integrated between MPC-based query operators. This allows for precisely controlling the security/performance trade-off on a per-operator and per-query basis. Our method has the potential to accelerate the performance of current oblivious query execution by up to 200 times compared to fully oblivious query execution, and by approximately 7 times compared to existing approaches with the same security guarantees. Our work lays down the foundation for a future MPC query planner that can pick different performance and security targets when composing physical plans.

This demonstration showcases the benefits of Reflex. More precisely, it focuses on the integration of our proposed resizers into the oblivious query plan, significantly enhancing performance. Conference attendees will have the opportunity to observe the efficient trimming of intermediate results and, additionally, they will be able to configure the oblivious execution settings, ranging from fully oblivious to fully revealed. This hands-on experience will highlight the benefits of our proposal in various obliviousness scenarios.

PVLDB Reference Format:
Long Gu, Shaza Zeitouni, Carsten Binnig, and Zsolt István. How SMPC Query Execution can be sped up through Efficient and Flexible Intermediate Result Size Trimming. PVLDB, 18(12): 5311-5314, 2025.
doi:10.14778/3750601.3750659

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/DataManagementLab/reflex-smpc-analytics.

Proceedings of the VLDB Endowment, Vol. 18, No. 12 ISSN 2150-8097.
doi:10.14778/3750601.3750659

1 INTRODUCTION

Many global companies need to analyze internal data for decision-making, but transferring plain text data across jurisdictions can be risky or prohibited by regulations like GDPR [6]. This has led to growing efforts [2, 9, 10] to combine analytical data processing with secure Multi-Party Computation (MPC) for stronger security and privacy guarantees. MPC allows multiple parties to jointly compute a function without revealing more than the final result.

MPC computations are significantly more expensive than plaintext operations [2]. For example, under the Secret Sharing approach [1], each arithmetic operation is expressed as a series of logical circuits evaluated through costly distributed protocols on encrypted data. Additionally, MPC algorithms must execute obliviously to avoid leaking information.

This requires redesigning SQL operators to ensure no information about their execution, input, or intermediate results is revealed. Intermediate result sizes cannot be disclosed unless they are the final output. Consequently, analytical queries often face prohibitive overhead due to data ballooning in the operator tree. For instance, an MPC SQL Filter operator must return its entire input table with a secret shared column indicating selected rows, unlike plaintext processing. Similarly, Joins must return a secret shared result in the size of the Cartesian Product of the inputs [9].

To reduce the overheads of oblivious SQL operators, a compelling performance argument is made [3, 5] by allowing parties to learn some information about the output size of intermediate operator results in a SQL query to enhance the query performance. The main idea is that part of the rows kept only for achieving obliviousness can be removed, and the ever-increasing output size can be reduced. Representative examples from related work focus on Join operators and propose sorting the intermediate output rows and trimming out a number of rows from the bottom of the intermediate output table [3]. However, this approach can impact the accuracy of the results. Nevertheless, this controlled information leakage brings significant query speedups that, depending on the actual operator selectivity, can reach several orders of magnitude.

There are two main limitations in related work. First, even though performance improvements through intermediate size trimming have been demonstrated, this has been done mainly in the context of Join operators. To our knowledge, no configurable mechanism has been proposed that plugs into any oblivious operator and can balance performance and security trade-offs. Second, a unified way to quantify the security of different intermediate result size trimming approaches is lacking. Some related work aims to provide

Differential Private guarantees, but as we later show, these are not a direct match to what oblivious query execution would require.

Similar to how databases offer a controlled trade-off between performance (parallelism) and correctness (consistency levels), we believe that, in the future, MPC-based databases should offer a way to choose dynamically, on a per-query basis, how much information leakage of intermediate result sizes is acceptable in exchange for better performance. In our group, we are building Reflex[7], a system that will make it possible to have such a flexible query planner for Secure Analytics on shared data.

This paper focuses on the Resizer operator of Reflex (see Section 2 for details), which optimizes the performance-security balance in secure collaborative analytics. Resizer enhances efficiency by dynamically reducing the output size of oblivious operators and is designed for seamless integration into MPC-based query plans. With linear computational complexity, Reflex[7] delivers significant performance gains over existing methods with minimal effort. Through the interactive interface (see Section 3), we demonstrate that our new Resizer, with controlled information leakage, significantly enhances efficiency and flexibility, adapting to various real-world scenarios.

2 REFLEX OVERVIEW

In Reflex[1], we present a highly adaptable mechanism designed to reduce the size of oblivious operator outputs below their maximum oblivious size while preserving result accuracy. This is accomplished through the implementation of diverse noise generation and addition strategies, enabling optimal balancing of the trade-off between security and performance based on the tolerated level of information leakage.

To achieve this, we introduce the Resizer operator, denoted as ρ, which can be strategically placed within a query plan between the first and last operators. The Resizer minimizes the output size of the preceding oblivious operator by trimming randomly-selected filler rows and operates in an oblivious manner, ensuring that no information about the actual output of the preceding operator is disclosed, except for the size of the trimmed noisy output.

To illustrate the operation of the Resizer, consider the following SQL query:

```sql
SELECT DISTINCT d.patient_id FROM Diagnoses d
JOIN Medications m ON d.patient_id = m.patient_id
WHERE m.medication = 'aspirin' AND d.icd9  = 414
AND d.timestamp <= m.timestamp
```

The corresponding query plan and Reflex-enabled query plan are depicted in Figure 1a and Figure 1b, respectively. In the Reflex-enabled query plan, two Resizer operators are inserted to reduce the input size of the Join operator to $S_1 \leq N_1$ and $S_2 \leq N_2$, where N_1 and N_2 represent the oblivious output sizes of the preceding Filter operators. This optimization decreases the computational overhead of the Join operator by limiting it to processing $S_1 \times S_2$ tuples instead of $N_1 \times N_2$, thereby enhancing efficiency while ensuring that information leakage remains within the acceptable limits.

[1]For space reasons, we are presenting Reflex here in a high-level fashion. More details about its implementation and complexity can be found in the following Arxiv paper [7]

2.1 System and Adversary Model

We consider a system model in which *computing nodes* execute secure collaborative analytics protocols on datasets $\mathcal{D}$ provided by *data owners*. A *data analyst* submits a query Q to the computing nodes and subsequently collects the final result R.

The translation of a SQL query into a Reflex-enabled query plan does not rely on any secret information, as we assume that no data-dependent optimizations are performed for now. Furthermore, the placement and preconfiguration of Resizer operators within the query plan do not depend on secret data. However, during the generation of the final Reflex-enabled query plan, data owners can provide input to fine-tune the preconfiguration of Resizer operators, thereby minimizing information leakage. Additionally, the data analyst can specify performance constraints to be considered during this process. Consequently, this translation step can be executed by one of the computing nodes. If data-dependent optimization steps are required for the query plan, the translation should be performed either by a trusted system broker or through an oblivious protocol among the computing nodes.

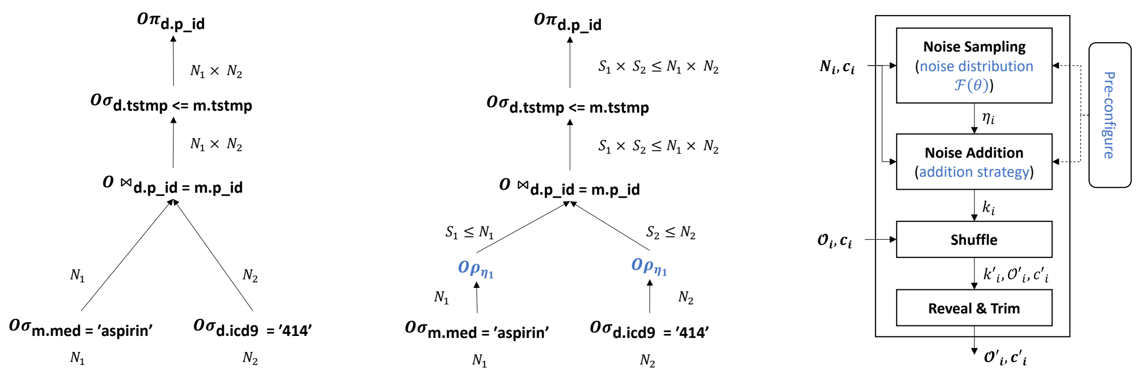
Adversary model. We consider a semi-honest adversary model where the computing nodes adhere to the protocol honestly but may attempt to infer information about the input datasets or the intermediate results of Q's operators. At most, one computing node may be corrupted, and computing nodes are assumed not to collude. All parties, including data owners, computing nodes, and the data analyst are assumed to have knowledge of the database schema and the queries to be executed.

2.2 The Resizer Operator

As Shown in Figure 1c, the Resizer takes as input the oblivious output tuples $O_i = O_i(\mathcal{D})$, the oblivious output size $N_i = |O_i|$, and the column c_i, which indicates whether a tuple is part of the operator's true output. As output, it returns the shuffled oblivious output O'_i and the shuffled column c'_i, indicating true output tuples, both trimmed to size S_i. Note that i refers to the index of the operator in the query plan. Our Resizer has the following components:

Noise Sampling. The noise η_i is sampled from a pre-configured noise distribution $\mathcal{F}(\theta)$. η_i plays a pivotal role in determining the number of filler tuples to be kept along with the actual/true output tuples of the preceding operator. To satisfy the requirement $S_i = T_i + \eta_i \leq N_i$ (where S_i is the trimmed noisy output size), the noise budget η_i is ideally sampled from the range $[0, N_i - T_i]$, where T_i represents the count of true tuples. However, since T_i is not available during the query planning phase, the noise distribution can be configured practically to sample from $[0, N_i]$ or $[0, \infty]$, ensuring the probability of $\eta_i > N_i - T_i$ is negligible. At runtime, η_i can be adjusted to $min(N_i - T_i, \eta_i)$, ensuring the constraints are met.

Noise Addition takes as input the noise budget η_i, the oblivious output size N_i of the preceding SQL operator O_i, and the column c_i, which marks true output tuples. The output of this step is an additional column k_i, indicating both true and filler tuples. We propose two noise addition strategies: 1) Sequential noise addition, which follows a deterministic approach by adding a predefined noise budget η_i to the true output size T_i. Tuples in the oblivious output O_i are processed sequentially, ensuring that a true tuple is always retained in the final output. A filler tuple is kept by setting its



$O\pi$d.p_id
$N_1 \times N_2$
$O\sigma$d.tstmp <= m.tstmp
$N_1 \times N_2$
$O \bowtie$d.p_id = m.p_id
N_1
N_2
$O\sigma$m.med = 'aspirin'
$O\sigma$d.icd9 = '414'
N_1
N_2
$O\pi$d.p_id
$S_1 \times S_2 \le N_1 \times N_2$
$O\sigma$d.tstmp <= m.tstmp
$S_1 \times S_2 \le N_1 \times N_2$
$O \bowtie$d.p_id = m.p_id
$S_1 \le N_1$
$S_2 \le N_2$
$O\rho_{\eta_1}$
$O\rho_{\eta_1}$
N_1
N_2
$O\sigma$m.med = 'aspirin'
$O\sigma$d.icd9 = '414'
N_1
N_2
N_i, c_i
Noise Sampling (noise distribution $\mathcal{F}(\theta)$)
η_i
Noise Addition (addition strategy)
Pre-configure
k_i
O_i, c_i
Shuffle
k'_i, O'_i, c'_i
Reveal & Trim
O'_i, c'_i

(a) Fully Oblivious Query Plan **(b) Oblivious Query Plan with Resizer (ρ)** **(c) Resizer Operator (ρ)**

Figure 1: Oblivious Query Plans. O indicates oblivious operators, N_i is input/output sizes, and S_i is re-sized output after the Resizer ($S_i < N_i$). Resizer inputs: oblivious output O_i, oblivious output size N_i, true output column c_i of operator O_i. Resizer outputs: trimmed and shuffled output O'_i and c'_i indicating true tuples.

corresponding k_i bit to 1, only if the noise budget is not exhausted. 2) Parallel noise addition, where the addition of η_i filler tuples is performed stochastically. A coin is flipped $N_i - T_i$ times, with each flip determining whether a filler tuple is kept. This strategy offers a notable advantage due to its high parallelizability, as the coin flips can be executed independently.

Shuffling. To prevent linkage attacks [4], oblivious output O_i, c_i and k_i tuples are shuffled, after adding the filler tuples (i.e., column k_i has been created) and before trimming the intermediate results. This ensures that no adversary can link secret shares by observing the outputs of different oblivious operators or by repeating the same or similar queries from multiple rounds. The shuffled oblivious output and additional columns are indicated as O'_i, c'_i and k'_i.

Trimming and Revealing After shuffling, column k'_i is revealed, indicating which rows in the oblivious output O'_i should be retained or discarded before proceeding to the next oblivious database operations.

3 DEMO SCENARIO: REFLEX IN ACTION

3.1 Web Interface

Part $\boxed{A}$: **Overall Comparison.** This section compares performance and information leakage metrics, demonstrating the benefits of Reflex's Resizer. Our new information leakage metric quantifies how many observations of trimmed intermediate result sizes an attacker would need in order to recover the true intermediate result size with high probability. Users will understand how Reflex enhances oblivious query execution, improving efficiency and flexibility in balancing security and performance.

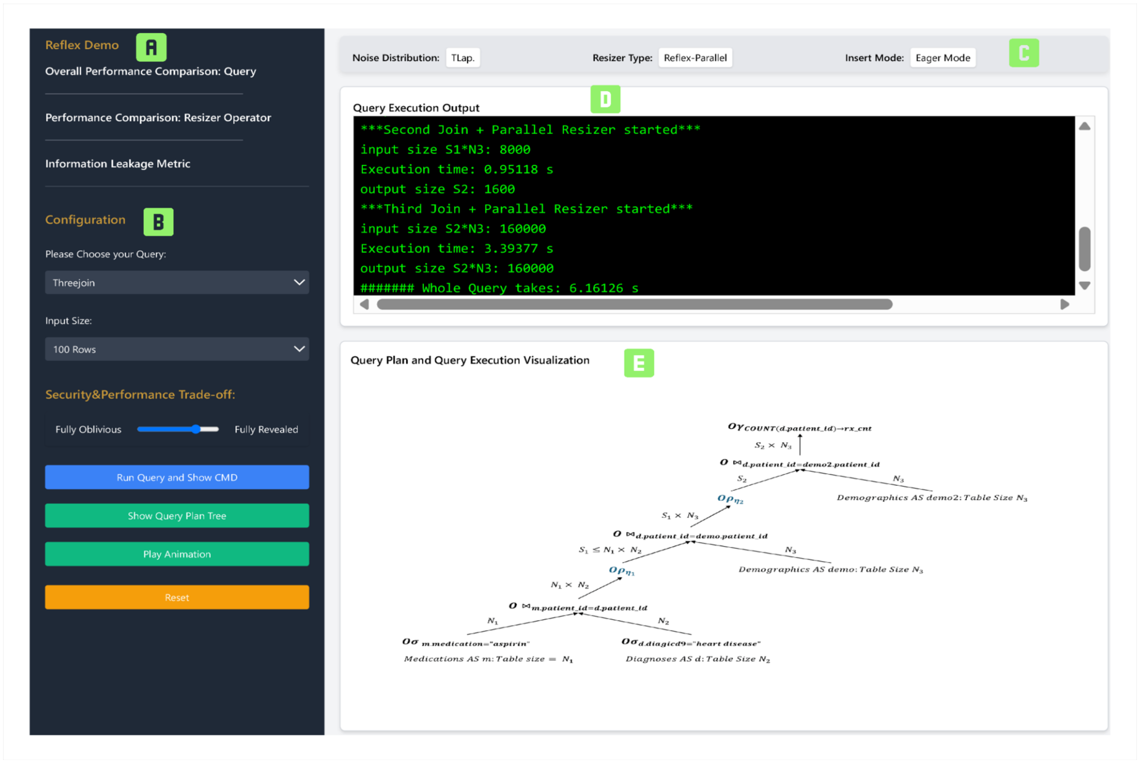
Part $\boxed{B}$: **Query Configuration.** On the sidebar, users can select the specific query they wish to run. A slider bar is included to allow users to explore the spectrum of reflex options, ranging from fully oblivious to fully revealed. By dragging the slider bar, users can observe the trimming parameters will be automatically configured and how changes in the security settings in $\boxed{C}$ affect the overall performance of the system. This interactive feature helps users explore the trade-offs between maintaining security and achieving optimal performance. By adjusting the slider bar configuration, users can select from five distinct execution modes for each query. These modes include "Fully Oblivious", "Sorting-based Resizer", "Reflex sequential Resizer", "Reflex parallel Resizer", and "Fully Revealed".

Part $\boxed{D}$: **Query Generation and Execution.** Behind part $\boxed{D}$, the backend system efficiently generates a .mpc file using parameters received from the part $\boxed{B}$. This file is then compiled into an MPC program using the pre-configured settings shown in $\boxed{C}$. The program subsequently executes the query locally and printing out the execution time of each operator according to the query plan.

Part $\boxed{E}$: **Query Plan and Execution Visualization.** Below the CMD line window, there is a detailed visualization of the query plan and execution process for the selected query. This section provides a step-by-step breakdown of how data is processed using the chosen resizing method, highlighting the placement of Resizers and the trimmed size of intermediate results. The visualization employs a traditional query plan tree to illustrate the execution order of the oblivious operator, including the Resizer. Additionally, users can press the "Show Animation" button in part $\boxed{B}$ to check the execution process for each of the different execution modes.

3.2 Implementation Details.

We integrate Reflex's Resizer into the MP-SPDZ framework [8], a secure MPC framework, to enable foundational oblivious operators for privacy-preserving query execution. While our implementation does not yet support fully automated translation of plaintext SQL queries to oblivious executions, it provides core primitives for secure computation within MP-SPDZ. We employ Replicated Secret Sharing (RSS) with three parties as the foundational secret-sharing scheme for Reflex, balancing performance and security while aligning with MP-SPDZ's architecture. For demonstration, the system is deployed on a local node with an Intel Xeon Gold 5220 CPU (2.20GHz) and 256GB RAM to ensure stable execution independent of cluster connectivity. This local setup mirrors the relative


Reflex Demo
A
Overall Performance Comparison: Query
Performance Comparison: Resizer Operator
Information Leakage Metric
Configuration
B
Please Choose your Query:
Input Size:
Security&Performance Trade-off:
Fully Oblivious
Fully Revealed
Run Query and Show CMD
Show Query Plan Tree
Play Animation
Reset
C
Query Execution Output
D
Query Plan and Query Execution Visualization
E
$O\gamma_{COUNT(d.patient\_id)\rightarrow rx\_cnt}$
$S_2 \times N_3$
$O \bowtie_{d.patient\_id=demo2.patient\_id}$
S_2
N_3
$O\rho_{\eta_2}$
$Demographics\ AS\ demo2: Table\ Size\ N_3$
$S_1 \times N_3$
$O \bowtie_{d.patient\_id=demo.patient\_id}$
$S_1 \leq N_1 \times N_2$
N_3
$O\rho_{\eta_1}$
$Demographics\ AS\ demo: Table\ Size\ N_3$
$N_1 \times N_2$
$O \bowtie_{m.patient\_id=d.patient\_id}$
N_1
N_2
$O\sigma_{m.medication=\text{"aspirin"}}$
$Medications\ AS\ m: Table\ size\ =\ N_1$
$O\sigma_{d.diagicd9=\text{"heart disease"}}$
$Diagnoses\ AS\ d: Table\ Size\ N_2$

Please Choose your Query: Threejoin

Input Size: 100 Rows

Noise Distribution: TLap. Resizer Type: Reflex-Parallel Insert Mode: Eager Mode

```
***Second Join + Parallel Resizer started***
input size S1*N3: 8000
Execution time: 0.95118 s
output size S2: 1600
***Third Join + Parallel Resizer started***
input size S2*N3: 160000
Execution time: 3.39377 s
output size S2*N3: 160000
####### Whole Query takes: 6.16126 s
```

Figure 2: The Reflex Demo: $\boxed{A}$ Performance Comparison and Information Leakage Metric $\boxed{B}$ Query Configuration and Execution $\boxed{C}$ Auto-configured trimming parameters $\boxed{D}$ Query Output $\boxed{E}$ Query Plan and Query Execution Visualization

performance trends observed in distributed environments, though absolute metrics may differ. All performance data are pre-measured across three configurations, which will be visualized in graphical formats for audience reference. These measurements empirically validate the framework's scalability and consistent performance across local and distributed environments.

4 CONCLUSION

This demonstration introduces Reflex, an efficient and flexible method for trimming oblivious intermediate results during MPC-based query execution. The core is the Resizer operator, which reduces the size of intermediate results without sorting or modifying upstream operators. The Resizer's runtime scales linearly with row count and logarithmically with column count, supporting future query optimizers focused on both security and performance. Reflex allows for flexible noise addition strategies, enabling privacy-preserving database systems to choose noise distributions based on compile-time and run-time factors. This flexibility impacts security and query performance, but Reflex can use pre-configured parameters to balance these aspects based on given conditions. Evaluations show that Reflex significantly reduces runtime, often outperforming existing methods by avoiding sorting algorithms and allowing parallel execution. Although the queries were hand-compiled for now, future optimizers would be further improved by compiling SQL directly into query plans with oblivious operators and resizers.

REFERENCES

[1] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. 2016. High-throughput semi-honest secure three-party computation with an honest majority. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 805–817.

[2] Johes Bater, Gregory Elliott, Craig Eggen, Satyender Goel, Abel Kho, and Jennie Rogers. 2016. SMCQL: Secure Querying for Federated Databases.

[3] Johes Bater, Xi He, William Ehrich, Ashwin Machanavajjhala, and Jennie Rogers. 2018. Shrinkwrap: efficient sql query processing in differentially private data federations. *Proceedings of the VLDB Endowment* 12, 3 (2018).

[4] Saba Eskandarian and Dan Boneh. 2021. Clarion: Anonymous communication from multiparty shuffling protocols. *Cryptology ePrint Archive* (2021).

[5] Wenjing Fang, Shunde Cao, Guojin Hua, Junming Ma, Yongqiang Yu, Qunshan Huang, Jun Feng, Jin Tan, Xiaopeng Zan, Pu Duan, et al. 2024. SecretFlow-SCQL: A Secure Collaborative Query pLatform. (2024).

[6] General Data Protection Regulation GDPR. 2016. General data protection regulation. *Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC* (2016).

[7] Long Gu, Shaza Zeitouni, Carsten Binnig, and Zsolt István. 2025. Reflex: Speeding Up SMPC Query Execution through Efficient and Flexible Intermediate Result Size Trimming. arXiv:2503.20932 [cs.DB] https://arxiv.org/abs/2503.20932

[8] Marcel Keller. 2020. MP-SPDZ: A versatile framework for multi-party computation. In *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*. 1575–1590.

[9] John Liagouris, Vasiliki Kalavri, Muhammad Faisal, and Mayank Varia. 2023. {SECRECY}: Secure collaborative analytics in untrusted clouds. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 1031–1056.

[10] Nikolaj Volgushev, Malte Schwarzkopf, Ben Getchell, Mayank Varia, Andrei Lapets, and Azer Bestavros. 2019. Conclave: secure multi-party computation on big data. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–18.