UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

Fakultät für Elektrotechnik, Informatik und Mathematik
Arbeitsgruppe Codes und Kryptographie

Evaluating database systems relying on secure multi-party computation

Bachelor's Thesis

in Partial Fulfillment of the Requirements for the Degree of

Bachelor of Science

by
Niklas Isermann

submitted to:
Prof. Dr. Johannes Blömer
and
Prof. Dr. Juraj Somorovsky

Paderborn, October 18, 2022

Eidesstattliche Versicherung

Nachname: ______________________________ Vorname: ______________________________

Matrikelnr.: _____________ Studiengang: ____

□ Bachelorarbeit □ Masterarbeit

Titel der Arbeit: Evaluating database systems relying on secure multi-party computation

□ Die elektronische Fassung ist der Abschlussarbeit beigefügt.

□ Die elektronische Fassung sende ich an die/den erste/n Prüfenden bzw. habe ich an die/den erste/n Prüfenden gesendet.

Ich versichere hiermit an Eides statt, dass ich die vorliegende Abschlussarbeit (Ausarbeitung inkl. Tabellen, Zeichnungen, etc.) selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate kenntlich gemacht. Die Abschlussarbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen. Die elektronische Fassung entspricht der gedruckten und gebundenen Fassung.

Belehrung

Wer vorsätzlich gegen eine die Täuschung über Prüfungsleistungen betreffende Regelung einer Hochschulprüfungsordnung verstößt, handelt ordnungswidrig. Die Ordnungswidrigkeit kann mit einer Geldbuße von bis zu 50.000,00 € geahndet werden. Zuständige Verwaltungsbehörde für die Verfolgung und Ahndung von Ordnungswidrigkeiten ist die Vizepräsidentin / der Vizepräsident für Wirtschafts- und Personalverwaltung der Universität Paderborn. Im Falle eines mehrfachen oder sonstigen schwerwiegenden Täuschungsversuches kann der Prüfling zudem exmatrikuliert werden. (§ 63 Abs. 5 Hochschulgesetz NRW in der aktuellen Fassung).

Die Universität Paderborn wird ggf. eine elektronische Überprüfung der Abschlussarbeit durchführen, um eine Täuschung festzustellen.

Ich habe die oben genannten Belehrungen gelesen und verstanden und bestätige dieses mit meiner Unterschrift.

Ort: ____________________________________ Datum: _____________

Unterschrift: ____

Datenschutzhinweis

Die o.g. Daten werden aufgrund der geltenden Prüfungsordnung (Paragraph zur Abschlussarbeit) i.V.m. § 63 Abs. 5 Hochschulgesetz NRW erhoben. Auf Grundlage der übermittelten Daten (Name, Vorname, Matrikelnummer, Studiengang, Art und Thema der Abschlussarbeit) wird bei Plagiaten bzw. Täuschung der/die Prüfende und der Prüfungsausschuss Ihres Studienganges über Konsequenzen gemäß Prüfungsordnung i.V.m. Hochschulgesetz NRW entscheiden. Die Daten werden nach Abschluss des Prüfungsverfahrens gelöscht. Eine Weiterleitung der Daten kann an die/den Prüfende/n und den Prüfungsausschuss erfolgen. Falls der Prüfungsausschuss entscheidet, eine Geldbuße zu verhängen, werden die Daten an die Vizepräsidentin für Wirtschafts- und Personalverwaltung weitergeleitet. Verantwortlich für die Verarbeitung im regulären Verfahren ist der Prüfungsausschuss Ihres Studiengangs der Universität Paderborn, für die Verfolgung und Ahndung der Geldbuße ist die Vizepräsidentin für Wirtschafts- und Personalverwaltung.

Contents

1 Introduction

Secure multiparty computation is subfield of modern cryptography that focuses around the question of how multiple parties with private inputs can evaluate a function over these inputs in a secure manner. In the context of multiparty computation secure means that no information about the private inputs is revealed. The first problem that was studied in the context of secure multiparty computation was Yao's "Millionaire's Problem". In Yao's "Millionaire's Problem" there are two millionaires that want to know who of them is wealthier and do not want to disclose any other information about their wealth. The "Millionaire's Problem" was solved by Andrew Yao in 1982 [Yao82] using Garbled Circuits.

Secure Multiparty Computation In the generally secure multiparty computation setting(MPC), there are n parties $P_1, \ldots, P_n$ that want to compute an agreed upon public function F based on their inputs $x_1, \ldots, x_n$. It is desired that no party P_i learns anything about the inputs of the other parties, that is not revealed by his input and the result of the function. In the example of the "Millionaire's Problem" there are two parties P_0, P_1, therefore n equals 2 and the public function is $F = argmax(x_1, x_2)$. In the context of the "Millionaire's Problem" secure means that P_1 and P_1 only learn who of P_1 and P_2 is wealthier, i.e they only learn if $x_1 > x_2$ holds or $x_1 < x_2$ holds, as that cannot be avoided if they want to evaluate the function. There is a variety of different security models for the field of secure multiparty computation that characterise the attacker or adversary and its capabilities. For example, an attacker may be able to corrupt at most one party or possible many parties. Another important security model is the distinction between a passive adversary and an active adversary, an active adversary is allowed to force a corrupted party to change its behaviour, while a passive adversary can only observe it. For the majority of security models there already exist generic solutions that allow to evaluate an arbitrary probabilistic polynomial time function under these models. Such generic protocols like [DW82] and [Yao82] have shown to have insufficient efficiency to be applied in practice. Therefore efforts have been made to develop new, more sophisticated protocols and optimise existing solutions. Archer et al. [ABL$^+$18] and Hastings et al. [HHNZ19] have conducted an evaluation of existing multiparty computation frameworks. Yet, their studies are not exhaustive and are partly concerned with simple toy problems. Therefore, they provide only limited information about the usability of secure multiparty computation in real world applications.

Database Applications One use case for secure multiparty computation are large scale database systems, where multiple companies may own large databases and are willing to cooperate together and compute shared statistics over their combined data. Yet they cannot afford to disclose individual records, as these may contain highly sensitive personal information or highly valuable trade secrets. One such use case is described by Bater et al. [BEE$^+$16], where several hospitals work together to use their patient data for scientific research. Of course, due to legal regulations, the individual patient data must remain strictly confidential. Another example for a practical utilisation of secure multiparty computation is documented by Archer et al. [ABL$^+$18], where in a pilot project, the secure multiparty computation framework Sharemind [BJSV15] was utilized to implement a system that helped the Estonian government detect tax fraud. Since the individual data sets cannot be published, join and aggregate operations are of particular importance. Frameworks that are included in our study are ABY3 [MRR20], CipherCompute[1] , Conclave [VSG$^+$19], Prio+ [AGJ$^+$21] , SMCQL [BEE$^+$16] and VaultDB [RAB$^+$22]. These frameworks are a very heterogeneous group and differ in their capabilities and security assumptions. For example Prio+ does support an arbitrary number of parties, while VaultDB only supports two and ABY3 works exclusively in a three party setting.

1.1 Related Work

Fundamentals of Secure Multiparty Computation As base literature in the field of MPC we have mainly relied on Cramer et al.[CD05] and Lindell [Lin17]. Cramer et al. gives a broad overview of the topic of secure multiparty computation and its theoretical foundations. Lindell provides an introduction into the topic of simulation based proof which is the key to understanding the security definitions of secure multiparty computation.

General Purpose Compilers for Secure Multi-Party Computation Hastings at al. [HHNZ19] did also conduct a survey about the state of secure multiparty computation. Yet there are several significant differences between their approach and ours. While Hastings et al. uses general purpose protocols that can compute arbitrary functions. We evaluate protocols that are specifically designed and optimized for the database context. Their survey was published in 2020, therefore two of the frameworks that are subject of our study, namley ABY3 and VaultDB, have not been published by the time they conducted their survey. Another difference is that Hastings et al. did not collect data to measure the performance of the frameworks but rather focused on usability and documentation aspects.

MP-SPDZ: A Versatile Framework for Multi-Party Computation Keller et al. [HHNZ19] provide with MP-SPDZ an implementation of over 30 different MPC protocols, which enables it to compute arbitrary functions. It is notable that MP-SPDZ supports protocols for a range of different security assumptions, these include passive and active adversaries

[1] https://github.com/Cosmian/CipherCompute

as well as settings in which the adversary may corrupt all but one party alongside with settings where the the adversary can corrupt only a single party. MP-SPDZ comes with a high level programming interface that simplifies its usage. As MP-SPDZ is not optimized for the database operations that are of interest for us it is not included in our study.

1.2 Contribution

In our work we have evaluated existing secure multiparty computation frameworks that are able to perform typical database operations, like joins, union, and selects. We started with a pre-selection of 6 frameworks. First, we implemented different use cases with the frameworks on a trial basis, thereby we placed particular emphasis on the usability of the frameworks. Afterwards we selected the three frameworks that were most promising to yield good results and discarded the rest. We then defined several different use cases that include typical database operations. We implemented the use cases with the selected frameworks and evaluated their performance. We used different technical tools to keep our results as robust and reproducible as possible. Finally, we graphically present and discuss the collected data.

1.3 Structure

In this section we provide an overview over the rest of our work and summarise the content of each remaining chapter.

Preliminary In [Chapter two] we provide an overview over the topic of secure multiparty computation. We explain the general concept and different important security assumption. Furthermore, we explain the database operations that are of importance for our work.

Framework Description [Chapter three] contains a detailed description of the frameworks we used to perform our benchmarks. Furthermore, a comparative discussion of the frameworks can be found in the chapter, where we especially focus on usability aspects. The chapter concludes with a description of the frameworks that did not make it into our final selection.

Benchmarking In [Chapter four], we describe the different metrics for which we collected data. In particular, we list the different tools that helped us to do this and provide an overview how they are used.

Implementation In [Chapter five], we describe our implantation. First, we describe our use cases and motivate their design. Then we describe the concrete implementations with the individual frameworks, including commented excerpts of our source code.

Evaluation In [Chapter six], we evaluate the data we have collected. We highlight special features and provide a graphical representation of the data. Finally we conclude our work with a discussion of our results.

2 Preliminary

In this chapter we establish important vocabulary for our work. At first we provide an overview over the topic of Secure Multiparty Computation. That is followed a description of the semantics of different database operations, that are of importance for our benchmarks.

2.1 Secure Multiparty Computation

In the secure multiparty computation(short MPC) scenario there are n parties $p_0, \ldots, p_{n-1}$. They want to compute an agreed upon functionality $\mathrm{F}(x_0, \ldots, x_{n-1})$. A functionality is a function that is allowed to have internal randomness. Each party p_i holds an input value x_i. The parties keep their input private and do not want to reveal any information about it.The goal of secure multiparty computation is to develop a protocol π that enables them to jointly compute $\mathrm{F}(x_0, \ldots, x_{n-1}.)$. The security goal of "not revealing the inputs" is often formalised through the Real-/Ideal-World Paradigm.

2.1.1 Real World and Ideal World

When modelling security of secure multiparty computation we compare the real world, to a perfect ideal world, where the problem can be solved in a perfect way. In the perfect world, there exists a trusted third party that evaluates the function for the other parties. Because of this, no information is leaked besides the output of the function. In order for a protocol to be secure, it is required that its execution in the real world does effectively reveal no more information then the ideal worlds solution.

Real World In the real world there exists a protocol π which was designed to enable the parties to compute F. All parties execute the protocol together. During the execution they communicate for several rounds. The attacker or adversary has the ability to corrupt one or more of the parties. His capability to influence the corrupted parties is an important parameter and may differ based on different security assumptions. These may range, for example, from a relatively weak adversary that can only read messages, to a very powerful adversary, which can actually influence the behaviour of the parties he corrupts. We explain the adversary models that are of importance for our benchmarks in Section 3.2 in detail. The real world view of this attacker A consists of the inputs of each corrupted party, their obtained messages throughout the protocol and the used randomness. The protocol achies our security goal if the view of A contains no more information beyond what can be deducted from the corrupted parties' input and outputs alone, which we formalise by simulation in the ideal world.

Ideal World In the ideal world the parties can rely on a trusted, incorruptible third party P that aids them. With the aid of P, the parties can evaluate F in two simple steps. In a first round of communication every party sends P its input. P now holds all information needed to compute F. Afterwards, P sends the result to each party in a second round of communication. Like in the real world, in the ideal world there also exists an adversary. Similar to his real world counterpart, he is also able to corrupt one or more parties. Compared to his real world counterpart the ideal world adversary has otherwise very limited abilities. He can only see the input and output of the parties he corrupts. Especially the computation with the aid of P produces no intermediate results that he can observe. Depending on the underlying security assumptions he also may be able to modify the input a corrupted party sen to P in the first round of communication. For security we want to show that a real world attacker, effectively, learns nothing more than such an ideal world adversary with its clearly specified, limited capabilities. This is specified using the simulation paradigm.

Simulation Paradigm Given a real-world adversary A, a secure multiparty computation protocol π, and a functionality F for π to be secure we require the existence of a so called simulator S. S is an ideal world adversary for F that has to indistinguishably simulate views the real world attacker A obtains in a protocol execution. To achieve this, S is only allowed to perform corruptions consistently with A's behaviour. This means that S and A corrupt the same parties and if A is an active adversary, then S is allowed to change the inputs of its corrupted parties. After S has performed its attack, S computes and outputs a real world view. Protocol π is secure against A, if the view S outputs is indistinguishable from a view of A. This means that the real world attacker A cannot learn more than the ideal world attacker S, despite the very limited abilities S has compared to A. Finally, we say π is secure, if, for all A, π is secure against A.

2.1.2 Adversarial Models

There are multiple models and categorizations of adversaries and their capabilities. These distinctions have significant impact on feasibility and difficulty of secure multiparty computation. In the following we will outline the models and assumptions that are of importance for our benchmarks.

Passive Adversary vs Active Adversary A passive adversary cannot force a corrupted party to deviate from the protocol in any way. One could think of a passive adversary as a ”read-only” adversary, as a passive adversary is only able to read the messages his corrupted parties receive or send. An active adversary is allowed to do everything a passive adversary is allowed, in other words, the set of all abilities an active adversary has, is a super set of the set of all abilities an passive adversary has. He has the additional power to force a corrupted party to deviate from the protocol in an arbitrary way. So if for example, the protocol would at some point require that each party choses an integer between 1 and n uniformly at random ,then a passive adversary would have no choice but to choose the integer between 1 and n uniformly at random. On the contrary, an active adversary would be able to force a corrupted party to chose the value 42 or any other value that the adversary considers to be advantageous for him. In the ideal world a passive adversary is bound to forward the real input values. A active adversary can choose to ignore the real input and forward any value instead. A passive adversary is an apt modelling for scenarios in which an attacker manages to take over a party after the execution of the protocol. This is for example conceivable if the attacker takes over a party after the protocol has already been executed but all important information is still in memory. In such a scenario it is impossible for the attacker to influence the behaviour of the corrupted party. In such a scenario, security against a passive adversary guarantees that the attacker cannot gain any valuable information. Furthermore, due to the additional capabilities of an active adversary, protection against such an adversary is often associated with high performance losses and therefore not always desirable in practical applications.

Monolithic Adversary A common assumption is the assumption of a so called monolithic adversary. When assuming a monolithic adversary one assumes that, there is only a single adversary that controls all corrupt parties. For the honest parties a monolithic adversary is a worst-case scenario. A monolithic adversary is more powerful than multiple adversaries that control the same total amount of parties but do not cooperate with each other. A protocol that is secure in the presence of a single adversary that corrupts n parties and is able to coordinate their efforts, will be secure in the presence of up to n adversaries that corrupt n parties total and do not coordinate their efforts. In the following we will assume a monolithic adversary.

General Adversary vs Threshold Adversary One important distinction in modelling adversaries is the distinction between a general adversary and a threshold adversary. In the threshold setting each party is a legitimate target for corruption and the threshold adversary is parametrized by a parameter t, 0<t<n, which denotes the limit of parties it is allowed to corrupt. A common setting for t is t = $\lfloor \frac{n}{2} \rfloor$, which is called the honest majority. For example for n=3 the presence of an honest majority means that it is assumed that the threshold adversary can corrupt at most 1 party. Threshold MPC best fits scenarios that feature a very homogenous group of parties, as it cannot model differences between parties. To model a heterogeneous group of parties on can rely on a general adversary. A general adversary is limited in his choice which party he corrupts by an adversary structure $Z = \{Z_1, \ldots, Z_l\}$. Where Z_i can be any set of parties. The general adversary must only corrupt a set of parties P such that there exists an $X \in Z$ that holds $P \subset X$. This allows for a flexible way to formalise assumptions. If, for example, in a protocol there are two parties that hold a very vital role and one wants to assume that no adversary can corrupt both of these parties, that can be formalised by using a general adversary and defining Z so that no element in Z contains both of these two parties.

Static vs Dynamic Corruptions Another important distinction is the distinction between static and adaptive adversaries. A static adversary is forced to choose which parties he wants to corrupt before the execution of π starts. An adaptive adversary can corrupt a party during the execution of the protocol. This makes the adaptive adversary much more powerful. He can try to identify "weak links" based on the information he gets during the execution of the protocol and then choose corrupt those. Because adaptive adversaries is more powerful then a static adversary, it is harder to achieve a protocol that is secure against such an adversary. Therefore protocols that are secure against adaptive adversaries are typically significantly slower, then protocols that are only secure in the presence of a static adversary.

2.2 Relational Databases

Relational database are not the only popular type of database that can be secured using multiparty computation, as for example Cui at al. [CCLW20] describe an approach to use secure multiparty computation to secure a graph database. Yet relational databases are in the focus of our work as they "continue to be the dominant data management system" [ABL+18]. In this section we describe the semantics of different relational database operations that are important in our use cases.

Select, From, Where The select,from,where operator consists of three parts and takes a single table as input. The `from` part specifies the input table from which the operator selects. The `select` part specifies a subset of the columns of the input table, which are the columns of the output. The `where` part, defines a boolean constraint and each row of the input that does not fulfil the constraint is removed from the output.

Projection The Projection operator takes as input a single table. The table is allowed to have any kind of layout. The output is also a single table. The output table is created by using the input table and removing the specified columns. If two rows differ only in one column that was removed, the resulting duplicate is also removed.

Equi-Join The equi join takes two tables as input and outputs a single table. In a first step the cartesian product of the input tables is formed. The cartesian product of two tables consist of all columns of the first table followed by all columns of the second table and is filled with all possible combinations of row of the two input tables. In a second step a set of equality clauses over the columns is defined and each row of the cartesian product that does not fulfil the equalities is removed. The remaining rows that fulfil these conditions form the final result.

Natural Join The natural join is special case of the equi-join instead of defining a set of equalities that are used for the join, the natural join automatically uses the condition that columns with incidental name need to have identical values otherwise they are removed.

UNION The union operator takes two tables as input and outputs a single table. In order for the two tables to be eligible they need to have the same amount of columns and their columns need to have matching types. This means both tables need to have n columns and forall i, with $1 \leq i \leq n$, the i-th column of the first table needs to have the same type as the i-th column from the second table. The output is a table that has an identical layout as the input tables. The output table contains each row of the first table followed by each row of the second table. Duplicates row are removed. We will sometimes refer to union as row wise concatenation.

Left Join/Right Join A left join is an extension of the equi-join. The left join takes an equi join and adds each row of the left input table that was not included in the original equi-join to the output table . In these rows the columns of the right input table are filled with null values. The right join works accordingly. It takes the output of an equi-join and adds the rows of the right input table and fills the missing columns of the left table with null values.

3 Framework Description

In this chapter we describe the different MPC frameworks that are candidates for our study. First we describe the frameworks for which we have implemented our use cases. We have dedicated one subchapter for each of these frameworks. We begin we a description of the feature they implement. Afterwards we describe the MPC technology they utilise to realise their functionality. We describe framework-specific details like special algorithms or other optimisations. At the end of the chapter we also provide a brief overview over some MPC frameworks that were candidates for our study, but are not part of our final selection.

3.1 Conclave

Conclave [VSG+19] is a secure multiparty computation framework which is optimised to implement relational database operations. Conclave based on Python 3.5 and is able to work with either 2 or 3 parties. Conclave was released in 2019.

3.1.1 Features

Conclave [VSG+19] allows to perform MPC analytics on "big data". Conclave aims to provide a high-level interface that abstracts internal MPC details away from the user. Through this high abstraction level Conclave aims to make MPC more accessible for those who are not experts in this field. Every operation done with Conclave is composable, that means that the output of every query can be the input of another query. This mechanism makes it possible to construct very complex queries out of multiple simple queries. With Conclave one can join tables using the equivalent of an equi-join or an union operator. Conclave also supports a range of aggregate functions. These include sum, mean and standard deviation.

3.1.2 Documentation and Usability

Another important property of a framework is its documentation. Conclave's documentation features three key components. First, Conclave features a quick start guide. The quick start guide functions as entry point for new users and guides them through Conclave's initial setup. Second, Conclave comes with an external documentation that provides a detailed description for the majority of its functions. For each function it explains the expected behaviour as well as how input and output are supposed to be provided and received. Conclave's external documentation archieves to provide an overview of its capabilities to the user. The part of Conclave's documentation where it shines the most are its comments. Within Conclave every important function is fully commented and even functions that are of limited significance for the end-user are partially commented. The presence of such extensive comments simplifies the usage of Conclave, as it allows to look up specify implementation details, that are missing in the external documentation, with limited effort.

3.1.3 Underlying MPC Technology

Conclave utilises existing MPC frameworks for its backend to perform its underlying MPC operations. Therefore Conclave inherits most security guarantees and assumptions from these frameworks. The frameworks of use are Sharemind [BLW08] and Obliv-C [ZE15]. Both of these frameworks are designed to withstand passive a adversary and do not support more then 3 parties. Conclave also assumes a passive adversary and supports up to 3 parties. Since Obliv-C is based on garbled circuits and Sharemind on secret sharing, Conclave uses both and switches between them internally. Conclave interacts with its backend through a generic interface ,therefore it is theoretically feasible to integrate another framework to add support for more than 3 parties. Conclave assumes a passive threshold adversary that corrupts statically and is bound by an honest majority.

3.1.4 Optimizations

MPC techniques are multiple orders of magnitude slower then cleartext processing. It is Conclave's key principle to archive better performance by avoiding the use of unnecessary MPC operations where possible. Instead of exclusively using MPC operations, Conclave evaluates queries with a combination of local cleartext processing and MPC operations. When Conclave compiles a query it applies various optimizations to it, one such optimization is conclave's query rewriting

Query Rewriting If all input data of an operator belongs to one party, this party can evaluate the operator locally without having to rely on expensive secure multiparty computation operations, because it has all data necessary for the computation locally available. In this mechanism lies potential to optimise many queries, as the order of operations decides which operations can be evaluated locally and which can not. For example, if a projection is applied to an input table for a query in a first step and then in a second step a join is computed, Conclave can compute the projection locally. In this particular example, the order of the operations is crucial, because if one would first calculate the join and then apply the projection, one would get an identical result, but would be forced to calculate the projection with an expensive MPC algorithm. Conclave's query rewriting analyses queries and automatically swaps operations so that as few operators as possible have to be calculated with MPC operations and as much as possible locally. Another example would be a query in which two tables are concatenated and then a filter is applied to the concatenation. In this example, one obtains an identical result if one first applies the filter locally to each table and then concatenates the filtered tables with an MPC algorithm. Conclave is able to automatically apply this optimization and thus minimize the required MPC operations.

Trust Annotations and Hybrid Operations Conclave features optional trust annotations that allow for a trade-off between security and performance. With these trust annotations one party can annotate that it does trust another party to learn the values of a specific column. There exists a variety of use cases that fit this mechanism. For example, the sensitivity of data may largely differ between columns. Therefore it may be desirable for a party to reveal some less sensible data in order to speed up the computation. If a party decides to do so , Conclave uses these annotations to apply optimisations that speed up query evaluation. One example of such an optimization are Conclave's hybrid operators. When possible Conclave substitutes expensive MPC operations with cheaper hybrid operations. In a hybrid operation one party is ”promoted” to a selectively-trusted party (short STP). Conclave reveals some input columns to the STP. Such leakage is only possible if the parties did explicitly allow it with the trust annotations. Otherwise it is not possible to apply hybrid operations. With the information the STP obtains, it can evaluate the operator using mainly local computation and only minor MPC based aid from the other parties. Besides the leakage of the input columns to the STP, Conclave upholds it's standard security guarantees for every other column. For these special operations Conclave's security assumptions differ from its normal security assumptions and can be modelled using a general adversary. Conclave's hybrid operations can withstand any adversary that can corrupt a set of parties ,that exclusively contains the STP, or does not contain the STP, and could be withstood by a standard operation.

Sorts and Shuffles Many of Conclave's high-level operators include “sub-protocols” like sorts and shuffles. These sorts and shuffles are MPC operations. As such they are highly expensive operations. Yet not all of these sorts and shuffles are always necessary. If for example ,an operator produces a sorted intermediate result ,like an `order by` operation would do, it is redundant to sort again as part of the next operator. Conclave is able to identify such redundant sorts and shuffles and eliminates them where possible. The ability to skip such expensive MPC operations provides significant performance gains.

3.2 ABY3

ABY3 [MRR20] is a secure multiparty computation framework that is based on C++ and was released in 2020. ABY3 is a mixed framework for machine learning and database operations, in our work we focus exclusively on the database features of ABY3.

3.2.1 Features

ABY3 is a 3-party MPC framework that allows to compute queries on relational database tables. It focuses on computing various SQL-like join operations as efficiently as possible. It features a large range of different join operations. These include, but are not limited to, left join, right join, set union, set minus, and also full joins. Besides joins it is also possible to query a single table with queries that have a comparable semantic to the ”SELECT ... FROM ... WHERE” statement in SQL. For example, a selection like ”select X1 from X where X2 > X1” can be done with relative ease using the implemented features of ABY3. In theory, ABY3 is able to compute any polynomial time function of a table, in practice, the efficiency may differ between functions, and may not always be sufficient. For executing its MPC operations ABY3 relies mainly on secret sharing. One of ABY3’s great strengths is its composability. Each operation done on one or more tables produces as output also a table, which is a valid input for another query. This allows to build larger complex applications out of many small ones, very similar like one would do with a pipes-and-filters architecture, which is best known from the Linux command line interface(CLI).

3.2.2 Documentation and Usability

Documentation Probably the most common form of documentation for any software are comments within the code, in this aspect ABY3’s documentation yields mixed results, as it features classes that are very well commented, while there are important classes that have barely any comments in them. ABY3’s documentation also features a quick start guide that demonstrates some of its core capabilities like setting up three parties and performing basic integer operations. For external documentation, ABY3 comes with a description of how aggregate functions like MAX, SUM, COUNT can be realized when utilizing ABY3. For example, the maximum operator can be evaluated with a recursive algorithm that computes the maximum of the first and second half of the rows. Yet these descriptions focus on high level concepts and many insights that are of great importance for a practical implementation remain unspecified. ABY3’s pre implemented prototype implementations contribute also to its documentation, as they demonstrated how some of its more complex functions are supposed to be utilised. One aspect of documentation in which ABY3 shines is are tests. ABY3 comes with 108 automated tests that significantly simplify the process of installing ABY3 and doing its initial set up, as they allow specific error tracking.

Prototype Implementations ABY3 demonstrates its capability in two prototype applications. One of them illustrates the possibility of ensuring the validity of voter registration records. In the United States, each state maintains its own list of registered voters. Through the highly sensitive nature of these records, coordination between states to ensure their faultlessness is not trivial. For that reason, one person moving from one state to another may often result in being registered in both states, which would allow them to illegitimately cast a vote in both of these states. ABY3 demonstrates that it provides the states with the tools needed to detect such double registration while preserving the confidentiality of the records.

3.2.3 Underlying MPC Technology

ABY3 works within a 3 party setting with an honest majority. This is a conscious decision as the two party and tree party settings each provide their own advantages and disadvantages. The third party allows to deploy more efficient algorithms that could not be deployed in a two-party setting. For example, oblivious permutations can be done in O(n) in a three-party setting instead of O(n log n) in a two-party setting [MRR20] ABY3 guarantees security against a passive threshold adversary. For executing its MPC operations ABY3 relies mainly on secret sharing. In order to archive composability of operations it is important that input and output have the same format. For ABY3 this requirement means that input and output of its operations are secret shared. For secret sharing based techniques having the output in a secret shared from can be archived by omitting the final reconstruction step, that would transform the output into the clear information. Therefore having a secret shared based protocol be composable can be achieved in a natural way. This property gives ABY3 a significant advantage over many other MPC frameworks. Pinkas Et al. [PSZ14] for example present a framework with similar capability and performance as ABY3 that is not composable as its algorithms require the input to be present in cleartext and also cannot extended to be composable in trivial way [MRR20].

3.2.4 Optimizations

ABY3s key feature are its new protocols for joins based on a MPC based cuckoo hash table. With these new protocols it is possible to join n rows with only O(n) overhead.

Secure Join using Cuckoo Hashing In the following paragraph we describe ABY3's join protocol that based is on cuckoo hashing. Here we describe the main concepts and high-level ideas. In particular, we do not describe how to implement the protocol securely. ABY3's version of cuckoo hashing is a special variant of a hash table. The table is based on a vector T that has m slots that can hold items. We address the i-th slot of T with T[i] for any i with 1 $\leq$ i $\leq$ m. For a given keyspace X the hash table utilises two hash functions h_0 , h_1: X $\mapsto$ $\{1, ..., m\}$. A hash function can be practically any function and is used to map the data to its storage location, as the size of the keyspace is normally larger than m, the hash functions cannot be injective. To insert any x $\in$ X into the hash table x, is inserted into T[$h_i(x)$], where i $\in$ $\{1, 2\}$ is decided with a coin toss. If T[$h_i(x)$] already contains an element, the element is removed from the hash table , and afterwards reinserted. This approach yields an important invariant: For any x it always holds that x is located in T[$h_0(x)$] or T[$h_1(x)$]. Therefore cuckoo hash tables have O(1) worst case lookup time, as a lookup only needs to check two possible positions in the table.

One key task for computing any kind of join is identifying which rows have identical join keys. More precisely, for two tables X,Y and key columns X_1, Y_1 and any given i the question, if there exists y such that $X_1[i] = Y_1[y]$, must be answered, where X[i] denotes the i-th row of table X and $X_1[i]$ the i-th entry of the first column of table X. ABY3 implements an algorithm that solves this problem using a secure cuckoo hash table T with two hash functions h_1 and h_2.

1. In a first step each row of Y is inserted into the hash table, such that Y[i] is inserted into $T[h_0(Y_1[i])]$ or $T[h_1(Y_1[i])]$.

2. If $X_1[i]$ has a matching join key $Y_1[y]$, then per definition it holds that $X_1[i] = Y_1[y]$, this implies also $h_0(X_1[i]) = h_0(Y_1[y])$ and $h_1(X_1[i]) = h_1(Y_1[y])$. The matching row $Y[y]$, if it exists, can only be located in $T[h_0(X_1[i])]$ or $T[h_1(X_1[i])]$. Therefore in a second step one is able find by comparing $Y[y]$ to $T[h_0(Y_1[y])]$ and $T[h_1(Y_1[y])]$ in a secure way.

3. Instead of comparing the values directly, T is transformed into two new tables T_1 and T_2. $T[h_b(X[i])]$ is inserted into $T_b[X[i]]$ for b in $\in \{0, 1\}$. After that X[i] must be compared with $T_1[i]$ and $T_2[i]$. At first this seems like extra work, but the consequence is that no hash function has to be evaluated in a secure way.

In the following paragraph we summarize the algorithm in a more intuitive way. First the rows of Y are inserted successively into the hash table. In order to find matches the keys that are used in the hash table are the keys used in the join. To check if a row x from table X has a row in table Y that matches the join criteria, one can compare it, with the two rows in the hash table that are located at the locations where one would insert x. Since hash keys and join keys are identical, every possible match of join keys is indicated by a possible collision in the hash table. This algorithm only needs 2 secure comparisons for each input row, which is significantly better than the n comparisons the naive approach would need.

The key challenge in this algorithm is the construction and usage of a secure cuckoo hash table that does not leak sensitive information. ABY3 implements such a hash table based on an oblivious switching network [MRR20].

3.3 SMCQL

SMCQL [BEE$^+$16] is a secure multiparty computation framework that is based on JAVA 8 and designed to execute database queries that are very similar so SQL. SMCQL currently only supports two parties and was released in 2017.

3.3.1 Features

SMCQL is an MPC based framework for relational database operations that is based on an already existing MPC framework, ObliVM [LWN$^+$15]. With SMCQL one can specify a query and SMCQL automatically generates secure code for evaluating the query. SMCQL realizes a private data network. A private data network is a union of many mutually distrusting databases that can be queried like a single engine that holds all data of every party. From the user's perspective, a private data network functions exactly like one monolithic database. With SMCQL one can specify queries in a semantic very similar to SQL and SMCQL translates these queries into a sequence of MPC operations. Therefore SMCQL allows using MPC without having detailed knowledge of the underlying system. With this approach, SMCQL wants to increase the accessibility of MPC. SMCQL supports a variety of SQL operators, these include selection, projection, aggregation, equi-joins, theta joins, and cross products.

3.3.2 Documentation and Usability

In terms of documentation, a strength of SMCQL are the three showcase applications that are already pre-implemented. They document the entire process from its start to the finished result. They start with pre-generated input and demonstrate how input should be formatted. They also provide a script that loads the input into the SMCQL database system and demonstrates this process. Furthermore, they provide formatted queries and show how to pass them to SMCQL so that SMCQL evaluates them. A disadvantage of SMCQL is the complete lack of external documentation. Thus, for a complete overview of which operations SMCQL supports, we had to search extensively in SMCQL's source code.

3.3.3 Underlying MPC Technology

SMCQL currently works in a two-party setting and provides security against a passive threshold adversary that can corrupt at most one party. It essentially resides in an honest majority setting. The two parties are aided by an honest broker, a neutral third party that plans the execution of the protocol. Besides the honest broker planning the execution of the protocol, he is not involved in its actual execution. For this reason, every MPC based operation always includes only the parties that are providing the data. The honest broker also functions as an access point for the user and receives his query. Once the honest broker receives the query, it parses the query into a directed acyclic graph of operators. Each node in the graph represents one operation and an edge between two notes annotates that the incoming node consumes the output data of the outgoing node. With the operator graph, the honest broker is able to analyse the flow of data through the query and decide which of SMCQL's different optimizations are applicable to each node. A detailed description of these optimization can found in Section 3.3.4. Once all optimizations are planned the honest broker generates secure MPC based code and provides it to the parties. For its secure computations SMCQL uses the already existing ObliVM framework.

3.3.4 Optimizations

SMCQL implements various techniques that speed up query evaluation and help it scale.

Access Control SMCQL features an accesses control system that enables the data owners to adequately model the sensitivity of their data. The accesses control is column based and each column is either public, protected, or private. A public column may always be revealed to any party including the honest broker. A protected column may be revealed if the query is k-anonymous. A query is k-anonymous if, for each queried tuple, it holds that the projection onto it's protected attributes is indistinguishable from at least k-1 other tuples. A private column is under no circumstances revealed to any party. With these accesses control mechanisms, SMCQL is able to speed up query evaluation by applying various optimizations. If, for example, an operator only works with public columns, it can be evaluated without using expensive MPC operations.

Slicing One such optimization is slicing. When SMCQL identifies an operator as sliceable, it partitions the input data into smaller units of computation. The partitioning of the input tuples is done horizontally. Small units of computation are easier to evaluate compared to a large monolithic operator, they allow for less complex secure code and in some cases, the evaluation of the units can be parallelized. Projections and filters are particularly easy to slice, as they can be evaluated working one tuple at a time.

Split Operators Another optimization that helps SMCQL scale are its split operators. A split operator splits the evaluation of an operator that requires MPC into two phases. The first phase of local plaintext computation that is followed by a second phase of MPC computation. The intuition behind this is that the MPC computation in the second phase is cheaper than the evaluation of the entire operator with MPC would be. Most aggregate functions can be spilt, in the first phase each party local aggregates over its own columns, and in the second phase, MPC is used to compute the correct aggregate out of these intermediate aggregates. The evaluation of a count(*) operator, for example, can be split, in the first phase each party locally counts its own input data, and in the second phase, these intermediate results are added up with the help of MPC.

3.4 Discussion of Frameworks

In this section we discuss important properties of the frameworks and compare them. At the end of the section we provide a table that summarizes our discussion. In their goals and general approach, Conclave and SMCQL are very similar. Both Conclave and SMCQL employ an existing MPC frameworks in a new way, with the primary goal to make MPC more accessible. Conclave even describes SMCQL as "the most similar state-of-the-art system" [VSG$^+$19]. ABY3 is new framework, that developed new MPC protocols, with the primary goal to be as much efficient as possible. Conclave, ABY3 and SMCQL all assume a passive adversary that can corrupt at most one party. Therefore their security assumptions and guarantees are quite comparable.

Features When comparing the implemented features of the frameworks, there is no clear best framework. Since ABY3 focuses its efforts on joins, ABY3 provides the largest selection of join operations. In particular, ABY3 is the only framework that supports left and right joins. One category of features that ABY3 completely lacks are pre-implemented aggregate functions. Conclave and SMCQL both support sum aggregation. In addition, SMCQL is the only framework that supports pre-implemented maximums and count operators. Therefore SMCQL is the framework which supports the most aggregate functions. It is notable that the only two features that implemented in every frameworks are the Equi-join and UNION operators.

Composability One notable aspect of each frameworks is their approach towards composability. Both Conclave and ABY3 lay huge emphasis on it and ensure all of their operators are composable. During our work with the frameworks this mechanism has saved us a lot of time and effort, because it significantly simplified the implementation of our more complex use cases. SMCQL does not achieve full composability. One example where SMCQL fails to archive composability are select statements, as the majority of SMCQL's select statements cannot be subject to another select statement afterwards. During our work with SMCQL it's limited composability has proven to be impractical.

Documentation It is not a trivial task to compare documentations objectively. There are hardly any measurable metrics that provide information about the quality of a documentation. However, one such metric is the amount of comments in the source code. We have found that SMCQL has the least amount of comments, in SMCQL there is about one comment for every 20 lines of code. ABY3 contains significantly more comments, the code of ABY3 contains one comment for every 10 lines of code. Conclave contains the most comments, because Conclave contains one comment for every 5 lines of code. This measurement is consistent with the experience we have gained while working with the frameworks. In particular, we have observed Conclave's comments to be significantly more useful than those of ABY3. We also perceived Conclave's quick start guide to be more helpful then the quick start guide of ABY3. We suspect the reason for this is the different approach of the two frameworks towards the quick start guide. Conclave demonstrates how to perform a single complete calculation in its quick start guide. ABY3 demonstrates multiple different small functions.

Experimental Software One shared property of ABY3, Conclave and SMCQL is that their implementations are experimental software that primarily serves as a proof of concept. This manifests especially as the existence of numerous bugs and undesirable behaviour that is part of every framework. Conclave, for example, features a failure detector that is supposed to detect if a party crashed because of an internal error and then abort the computation. This failure detector itself is extremely error-prone and tends to produce a significant amount of false positives, i.e. it aborts the execution of the protocol, despite no party actually having crashed. Such a false positive may, for example, be caused by a party that is fully occupied with a computation and does not respond to messages of the failure detector in time. For some queries this problem is more significant than for others, in extreme cases we observed more then 50% of all executions being aborted without justification.

Installing a secure multiparty computation framework and completing the install setup is often a complex and non-trivial task. To quote Hastings et al. “Installing and running these systems can be challenging” [HHNZ19] During our work we have observed this statement to be true for each of our candidates. Especially for Conclave and SMCQL the instructions are often vague or incomplete. Conclave, for example, requires the installation of JIFF [AIL+19], a JavaScript library, as one of its dependencies. In order for JIFF to be usable for Conclave, one must actively chose to ignore several recommendations during its installation. The task is additionally complicated by the fact that neither Conclave nor SMCQL provide sufficient tests to locate errors in their setup precisely. For installing and during its initial setup, ABY3 provides a script that automates this task. In theory, this approach simplifies the installation a lot. In practise, the automated installation comes with its own issues. If, like in our case, such an installation does not function correctly, fixing its errors is a time consuming task, as it’s lacks documentation, and therefore it is a non-trivial task to identify the root causes of it errors.

Table 3.1: Summary of important properties of the frameworks

	ABY3	Conclave	SMCQL
2 Parties	X	✓	✓
3 Parties	✓	✓	X
Passive Adversary	✓	✓	✓
Active Adversary	X	X	X
Composable	✓	✓	X
Quick Start Guide	✓	✓	X
UNION Operator	✓	✓	✓
Equi-join	✓	✓	✓
Left Join	✓	X	X
Right Join	✓	X	X
Theta Join	X	X	✓
SUM aggregate	X	✓	✓
MIN/MAX aggregate	X	X	✓
Projection	X	✓	✓
Automated Setup	✓	X	X

3.5 Rejected Frameworks

CipherCompute One candidate for our study was CipherCompute[1]. With the Cypher-Compute framework it is possible to solve a huge range of MPC problems using Rust.

[1] https://github.com/Cosmian/CipherCompute

These include SQL operations like joins that are of interest for us. Furthermore, CypherCompute provides a rich documentation, consisting of a full quickstart guide and several well documented example projects. CypherCompute utilises the SCALE-MAMBA [ACC+21] framework for its underlying MPC operations. SCALE-MAMBA itself has evolved out of the SPDZ [DPSZ11] protocol. Unfortunately the early access version of CypherCompute is out of maintenance by the time we are conducting this study. Therefore we have decided to not include CypherCompute in our study.

Prio+ Prio+ [AGJ+21] is the next generation of the Prio [CGB17] framework. Prio+ strives to maintain the same use and security as Prio, while significantly increasing performance compared to its predecessor. Prio Plus allows an arbitrary number of parties to jointly compute aggregated statistics, like SUM, MAX, MIN operators. Prio+ utilises a client server model in which the (potentially many) input parties use a small number of servers to compute the statistics. Prio+ guarantees confidentiality of the input values if at least one server stays honest. Unlike CipherCompute or Conclave, Prio+ is not a framework for developing MPC solutions. Its rather a system for special purposes. This means that the use of Prio+ can not be extended beyond the usecases that have been originally implemented by the authors of Prio+. This leaves Prio+ with a relatively small range of usecases compared to frameworks like ABY3 or Conclave. Therefore we have decided to not include Prio+ in our study.

VaultDB VaultDB [RAB+22] is a secure multiparty computation framework that was released in 2022. VaultDB is secure in the presence of a passive adversary and supports two parties, of which the adversary can corrupt at most one, therefore it resides in a setting with an honest majority. VaultDB is implemented on top of the EMP toolkit EMP toolkit [WMK16] which is an open source multiparty computation framework released in 2016. VaultDB provides an open source implementation which demonstrates its core capabilities. A major shortcoming of VaultDB is the complete lack of documentation, and as a result, a very low usability. Therefore it has been proven to be infeasible for us to implement any none trivial use case within our time and resources. Consequently, it is not included in our study.

4 Benchmarking

For benchmarking efficiency there is often a variety of different metrics that are relevant and require to be measured with great care, as flawed or unclean measuring may deteriorate the value of the results. In this chapter, we describe the different metrics we want to benchmark and the different tools we use to achieve valid results.

4.1 Measuring Runtime

A basic metric of how well a program works, is its runtime. Time is often measured in either wall-clock time or process time. Wall-clock time references, as the name implies, the passing of time on a wall-clock while the program runs. Process time resembles the actual time a CPU was used by the program. If, for example, the process blocks for a longer period of time, its wall-clock and process execution time may differ by a large amount. Typical reasons for a for a process to block would be that it waits for a slow hard drive or for incoming network traffic from another party. One needs to be careful what time is measured and that it is measured precisely. Otherwise one may obtain flawed or unfairly biased results. We measure both wall-clock and process execution time, as this allows us a far a more detailed analysis. In particular, this approach enables us to evaluate the difference between process time and wall-clock time, which indicates how long a process was blocked. In order to do so, tool-aided measuring is required. Measuring the individual frameworks is always trivial. Since the frameworks were implemented in different languages, different tools are available for measuring the individual frameworks.

SMCQL `time`[1] is a command line interface(CLI) command that is designed to benchmark the execution of other CLI commands. It is feasible to wrap the execution of a secure multiparty protocol in a .sh script and measure the performance of the script. Therefore `time` is in theory able to measure any of frameworks. One of `time's` advantages are the many different metrics it is able to measure, these include wall-clock and process time but also average memory requirements and maximum memory requirements. One major disadvantage of `time` is its inability to measure individual parts of a protocol in more detail. For SMCQL `time` perfectly fits our requirements. Therefore we use `time` to benchmark SMCQL

Conclave Because `time` is able to measure shell commands we could also use it measure Conclave. Conclave is based on Python and Python comes with a “batteries included”

[1] https://manpages.debian.org/stretch/time/time.1.en.html

approach, as it features a comprehensive standard library. Therefore plenty of tools, that provide valuable utility for us, are already integrated within python, because of this, there are other tools that are also valid candidates to measure Conclave. Two such candidates are timeit[2] and the python profiler[3], both are python libraries that offer a simple way to measure wall-clock or process execution time. Timeit measures exclusively end-to-end execution time, or in other words, the time the execution takes from one end to another. The python profiler comes with a more detailed analysis that includes information on which functions have been executed, how often they have been executed, and how long it took to execute them. The extra utility provided by the python profiler does not come for free, compared to timeit, it has significant overhead that slows the execution down. Therefore using it would result in an unfair disadvantage for Conclave. One significant disadvantage of timeit, is its inability to measure wall-clock and process time simultaneously. For this reason, in order to measure both of these values, it is required to perform each measurement twice, which doubles the time required and is impractical. Therefore we use the `time` tool with which we also measured SMCQL's execution time.

ABY3 To benchmark ABY3 we use the cryptoTools[4] library, which is integrated into ABY3. The cryptoTools library is a C++ toolbelt that features a variety of tools for building cryptographic protocols. Among these utilities is a benchmarking tool for measuring runtime. With cryptoTools it is possible to measure the execution time of specific parts of the protocol. The croytoTools library provides the features we need and is already "inbuilt" into ABY3, therefore we have decided to use it for measuring ABY3's runtime. For a detailed example of how we utilise this tool benchmarking, see the description of our implementation for the second use case in Section 5.

4.2 Networking

In our standard setup all parties run on the same machine and communicate through localhost. This simulates a practically perfect local area network (LAN) connection with very low latency and high throughput. During the execution of MPC protocol the parties send each other multiple round of communication. The specify quantity of commination can variate by a significant margin between different protocols. Therefore, it is also of interest how well the frameworks works in less ideal conditions. In particular, we also simulate a suboptimal wide area network(WAN) connection with high latency and limited bandwidth.

In order to do so we require a proxy server. Instead of connecting the parties to one another we connect them to the proxy server and the proxy server forwards the incoming messages to the addressed parties. To simulate a slow connection with high latency the proxy server delays incoming messages. Setting up such a proxy server is

[2] https://docs.python.org/3/library/timeit.html

[3] https://docs.python.org/3/library/profile.html

[4] https://github.com/ladnir/cryptoTools

non trivial task, one challenge in particular is mapping the various connections to one another, in a correct way. This task is made more difficult by the fact that the different frameworks handle their connections in various different ways. In ABY3 for example, each party holds one direct connection to every other party. Differently, in Conclave, every party is connected to a Node.js server that forwards the messages. In order to handle these various approaches correctly, an analysis of the communication patterns is required. An additional factor that complicates our task is the fact that different frameworks use various different protocols to communicate. For example Conclave uses among others HTTP,while ABY3 utilities plain TCP. Checking implementation details and source code is a target-oriented approach for such an analysis , another tool that helped us to understand the communication patterns is Wireshark.

Wireshark Wireshark[5] is an open source packet sniffer that allows to capture network traffic and saves it for a detailed analysis. Wireshark supports the analysis of numerous different protocols, among these are plain TCP, HTTP and websockets. Hence Wireshark supports all the protocols that our frameworks relies on, and therefore are of relevance for our work. With Wireshark we have been able to record the communication of our parties and pin down the exact communication patterns. Another utility Wireshark provides for us, is the ability to record communication once our proxy was setup. We also used Wireshark's recordings to confirm that our proxy works correctly once it was setup and used it detect error in our configuration.

Toxiproxy Both ABY3 and SMCQL implement communication between parties based on a plain standard socket. In the case of ABY3 it is the standard C++ socket. For SMCQL it is the standard java socket. Both of these are TCP based and can be proxied with a standard TCP proxy. For this purpose we use Shopify's Toxiproxy[6]. Toxiproxy is a Go framework that allows to simulate different hazardous network conditions. These include a connection that delays its messages to simulate a high latency setup. Once the proxy server is setup it can be configured over the command line interface (CLI) or alternatively over an HTTP interface, Toxiproxy provides multiple different dedicated HTTP clients for this purpose. The clients differ in that they offer an interface in different programming languages. We have chosen to use the provided Ruby client as it is the one that provides the most extensive documentation. A simplified example on how to use Toxiproxy to simulate latency can be found in Listing 4.1. In order to use Toxiproxy, we first set up the proxy so it starts to accept new connections. This is done by calling Toxiproxy populate and specifying to which address the proxy listens to and the address the messages get forwarded to. By default Toxiproxy does not add any network limitations to a connection. In our example we apply two limitations, which simulate a latency of 1000ms. The first limitation is applied to the "upstream" direction. Therefore it affects every message that is send towards the server from the address the server listens to. The second limitation is applied to the "downstream"

[5] https://www.wireshark.org

[6] https://github.com/Shopify/toxiproxy

direction. Therefore it affects every response that comes from the address the server listens to.

```
#First we instantiate a connection between the two parties.
Toxiproxy.populate([
{
 name: "aby3_party2_party1",
 #party 3 sends its messages for party1 to port 50010 therefore the proxy
   must listen to this port
 listen: "127.0.0.1:50010",
 #party 1 listens to port 50001 therefore theproxy must forward to this
   port
 upstream: "127.0.0.1:50001"
}
])
#Then we simulate a latency of 1000ms
toxiproxy−cli add aby3_party2_party1 —type latency —name upstream latency
   =1000 —upstream
toxiproxy−cli add aby3_party2_party1 —type latency —name downstream
   latency=1000 —downstream
```

Listing 4.1: Setting up a proxy that simulates latency between two parties with Toxiporxy

Node-Http-Proxy Conclave's communication is based partially on websockets and partially on plain standard HTTP. Websockets are implemented on top of TCP. In particular, websockets use a single TCP socket for bidirectional communication. Therefore proxying Conclave cannot be done with a simple TCP proxy. Instead we use node-http-proxy a library for proxying HTTP that also supports websockets. With node-http-proxy we are able to delay messages to simulate high latency. It is also possible to measure the amount of data sent and received over a connection. Node-http-proxy is based on JavaScript and relies heavily on an event driven programming paradigm. A simplified example how to use node-http-proxy can be found in Listing 4.2. In this example we first create a proxy server and specify the address it forwards to. In a second step we create a a HTTP server that delays every incoming message for 500 milliseconds and then sends it to the proxy server. Subsequently we demonstrate how Node-Http-Proxy make use of the event driven programming paradigm. In the third step we subscribe to the "proxyReqWs" event. The "proxyReqWs" event is raised by the proxy each time an outgoing websocket message is received. More concrete event is raised before the message is transmitted to its destination and we can submit an anonymous function as event handler. The event handler is executed for each message before the message is transmitted. This for example enables to modify responses or to apply various other practical use cases. In our specific example, we provide a function that waits 500 milliseconds to simulate latency. In the fourth step we subscribe to the "close" event. The "close" event is raised each time a websocket is closed. We use our event handler to save the amount of bytes transmitted though the websocket. This is a showcase example, of how a proxy helps to measure important metrics that would be non trivial to measure otherwise.

```
#create proxy server and specify the adress it forwards to
#with the ws:true parameter we enable support for websockets
var proxy = new httpProxy.createProxyServer({
  target: {
    host: 'localhost',
    port: 9005
  },
  ws:true
});
 # Here we crate a standart HTTP server that delays every incoming
  message for 500ms and then forwars it the proxy server.
var proxyServer = http.createServer(function (req, res) {
  setTimeout(function () {
    proxy.web(req, res);
  }, 500);
}).listen(9000);
# for every outgoing message the proxy emits a proxyReqWs event that we
  react to and delays the message for 500 milliseconds
proxy.on('proxyReqWs', function(){
   setTimeout(function(){ },500)
 });
  # each time a connection is closed the proxy emits a "close" event that
   we react to and save the amount of bytes transmitted through the
  connection
proxy.on('close' ,function (res,socket,head) {
  send  = socket.bytesRead;
  received  = socket.bytesWritten;
});
```

Listing 4.2: Setting up a proxy that simulates latency with node-http-proxy

4.3 Measuring Memory Consumption

Another metric we collect is the consumption of memory. Sometimes it is possible to speed up computations at the cost of their memory consumption, so it is not sufficient to measure only runtime to get a fair measure of efficiency. A well-known example of such a time-memory trade-off are Rainbow Tables [KKJ+13]. Another reason to measure memory usage is the fact that during our survey it often turned out to be a limiting factor for the size of datasets that are feasible to evaluate. For each framework we have measured its maximum memory consumption during an execution. With time we already have a tool that is able to measure the memory consumption of each framework in a sufficient way, therefore we use `time` to do so.

5 Implementation

In this chapter we describe the use cases we have chosen to implement and benchmark. We first describe and motivate our choice of use cases and, afterwards, give reasonable details on their implementation. We implement three use cases that are ordered from the least complex to the most complex.

5.1 Use Case Description

We have decided to implement every use case with Conclave and SMCQL for two parties. As ABY3 does not allow a two party protocol but requires at least three parties, we have implemented everything for ABY3 with three parties. For ABY3 there will always be only two parties that provide input data. The third party will not provide input data, but will assist in the execution of the protocol. We consider this approach justified, as it is the exact approach Conclave and ABY3 use in their evaluation. And if the authors of the frameworks considers such an approach fair, then we do not hold objections against it. We will refer to first party that provides input as Alice and to the second party that provides input as Bob. Furthermore, if a framework is not capable of replicating the semantic of a use case entirely, we do use that functionality available to implement the most similar semantic the framework can provide. For example, use case two features boolean logic, but Conclave does not feature an explicit boolean logic. Therefore we use integers where the integer one represents the value true and the integer zero repents false.

Use Case One For our first use case, we have chosen a single join. A single join may not be a very complex use case, but it is not irrelevant as joins are of great importance for practically every relational database query. It is estimated that over 60 % of privacy-sensitive analytics queries include at least one join [JNS17]. In our first experiment, Alice and Bob each hold one table. Each of these tables consists of 4 columns. The first column serves as the primary key that is used for the join. The other 3 columns are filled with random non-negative integers and simulate user data. We have chosen to use non-negative integers, because the usage of negative integers has resulted in an increased frequency of various bugs and other undesirable behaviour. We are calculating an equi join with the primary key generated in such a way that 50 % of the entries in each table will match the join criteria. We do not generate the primary key randomly, because the size of the output relation depends on it, and we want those sizes to be consistent between individual executions. As result, we will reveal the entire outcome of the join.

The primary utility provided by the use of MPC operations is the fact that the entries that do not match the join criteria are obscured.

```
SELECT *
FROM Alice A JOIN Bob B
ON  A.primary_key = B.primary_key
```

Listing 5.1: Functional equivalent SQL statement for our first use case

Use Case Two Computing joins alone is of limited use if the result of the join can not be subject to further selection. Therefore, in our second use case, we will first compute a join, and then query the result with a classic "SELECT ... FROM ... WHERE" statement. Similar to first experiment Alice and Bob again each hold one table. The tables consist of two columns. The first column serves as primary key and the second column contains a randomly generated boolean value. For our experiment, we will, in a first step, compute the natural join of the two tables, where the primary key column zips the two tables together. The primary keys will be again generated in such a way that 50 % of the columns will be included in the join. In a second step we will apply a where filter to the result of the join and eliminate every row that does not have two identical boolean values. This use case functions also as a simple showcase example for composability and will show how well this mechanism functions in practise.

```
  SELECT PrimaryKey, AliceBool, BobBool
  From AliceTable
  NATURAL JOIN BOB TABLE
  WHERE AliceBool = BobBool
```

Listing 5.2: Functional equivalent SQL statement for our second use case

Use Case Three For our third and last use case, we test a special feature of Conclave. Conclave features a mechanic that allows revealing some of the columns' input data. The revealed input data allows to apply optimizations that speed up the computation while preserving the privacy of the other columns. For a more detailed description see our description Conclave's trust annotations in Chapter 3. Therefore in our third use case, we are going to replicate the setup of our first use case but this time we will allow the leakage of the primary key column. Such leakage will allow Conclave to apply its optimization. Replicating the setup of the first use case enables us to compare the results of the third use case to the first use case. This comparison will show how big the speed up of these optimizations is in practice.

5.2 Implementation

In this section we provide a description of notable details of the implementation of our three cases. The descriptions are listed in order from the least complex to the most complex use case. Since SMCQL only requires the query to be specified in a valid SQL syntax and does the rest of the implementation automatically, we focus here on Conclave and ABY3 as their implementation is more complex. All our implementations are open source and can be found at[1] .

5.2.1 Use Case One

As our first use case is the least complex one, we use it for a full demonstration how ABY3's and Conclave's basic workflow functions. Our first use case consists of a very simple join, the equivalent SQL query can be found in in Listing 5.1.

Conclave In the first step we define the relation scheme of the input. Accordingly, we first define two lists of columns, one for Alice and one for Bob. Each column is defined with a name, a data type, and an integer that indicates its owner. The owner is used to annotate trust and apply optimizations, as described in our framework description of Conclave. In our first use case each party trusts only itself. Once the relation scheme is defined we can populate it by using the `create` function. Create loads the data from a Comma-Separated-Values file(.CSV) that we generated previously. Once the tables are populated we can compute the join. In order to compute a join in Conclave, we parse the two tables of the join, a name for the result, and the two columns that are used as key columns for the join. Lastly, we can reveal the output of our computation using the `collect` function. The `collect` function requires two parameters, one that specifies which table is revealed and a second one that specifies to whom it is revealed.

```
def protocol():
  # define the schema for the input tables
  AliceColumns = [
  defCol("primary_key", "INTEGER", [1]),
  defCol("user_data1_Alice", "INTEGER", [1]),
  defCol("user_data2_Alice", "INTEGER", [1]),
  defCol("user_data3_Alice", "INTEGER", [1]),
  ]
  BobColumns = [
  defCol("primary_key", "INTEGER", [2]),
  defCol("user_data1_Bob", "INTEGER", [2]),
  defCol("user_data2_Bob", "INTEGER", [2]),
  defCol("user_data3_Bob", "INTEGER", [2])
  ]
  # the content of the tables is loaded from a pregenerated .CSV
  AliceTable = create("AliceTable", AliceColumns, {1})
  BobTable = create("BobTable", BobClumns, {2})
  # calculate the join over the two tables
  JOIN = join(AliceTable, BobTable, 'JOIN', ['primary_key'], ['primary_key'
    ])
  # reveal the output of the join to Alice
  collect(JOIN, 1)
  # reveal the output of the join to Bob
```

[1] https://github.com/niklasUPB/Evaluating-database-systems-relying-on-secure-multi-party-computation

```
collect(JOIN, 2)
```

Listing 5.3: The Python protocol of Conclave for our first use case

ABY3 Before the actual execution of the protocol starts, ABY3 requires some setup. Similar to Conclave, the first step in ABY3 is also the definition of the relation scheme. Hence we start with the definition of two lists of `ColumnInfos`. Each column is described by a name, a data type, and some data type-dependent information. In our case, we use integers and annotate that we use 32-bit integers. After the relation scheme is defined, we create local tables and populate them. One last step that needs to be done before the execution of the protocol starts is the initialization of a message server that manages the communication between the parties. Once the execution of the protocol starts, we convert the local input into a table that is secret shared between the parties. This is done by joint effort of parties, the parties providing the input call the `localInput` function to populate the table and the other parties assist them by calling the `remoteInput` function. Before we can compute the join of the shared tables, we need to define which columns of the shared tables we want to select. Therefore, we define a list of references to the shared columns. In order to obtain the desired semantic, the list needs to contain each unique column and one reference to each column that is a duplicate in both tables. In our use case the columns containing the user data are unique and the key columns are duplicates because they have identical naming and data types. Therefore, our list contains seven columns. In the next step we compute the join by providing the join function with one reference to each key column and a list of columns we want to select. The output of the join is still a shared table. To obtain the plaintext values of our result we need to explicitly convert the shared table.

```
void use_case_two(u32 rows, u32 cols ){
  std::vector<ColumnInfo> AliceCols = { ColumnInfo{ "key", TypeID::IntID,
    32}};
  std::vector<ColumnInfo> BobCols = { ColumnInfo{ "key", TypeID::IntID,
    32}};
  for (u32 i = 1; i < cols; ++i)
  {
    AliceCols.emplace_back("Alice" + std::to_string(i), TypeID::IntID, 32);
    BobCols.emplace_back("Bob" + std::to_string(i), TypeID::IntID, 32);
  }
  # Create tables for Alice and Bob and fill them with content
  Table AliceTable(rows, AliceCols);
  Table BobTable(rows, BobCols);
  u32 intersectionsize = rows * 0.5;
  # Fill the primary columns such that 50% of all entries match the join
    criteria
  for (u64 i = 0; i < rows; ++i)
  {
    # if out is false then the entry will be included in the join
    auto out = (i >= intersectionsize);
    AliceTable.mColumns[0].mData(i,0) = i + 1;
    BobTable.mColumns[0].mData(i, 0) = i + 1 + (rows * out);
  }
  # Fill the other columns with random integers
  for (u64 i = 1;  i < cols; ++i){
    for (u64 j =0; j < rows; ++j){
      AliceTable.mColumns[i].mData(j, 0) = rand() ;
      BobTable.mColumns[i].mData(j, 0) = rand();
    }
  }
  # Instanciate server that handels communication
  DBServer server[3];
  ...

  # Here the exectution of the protocoll starts.
  auto protocoll = [&](int i) {
    # create a secret shared version of the input
    SharedTable AliceSharedTable = (i == 0) ?
    server[i].localInput(AliceTable) : server[i].remoteInput(0);
    SharedTable BobSharedTable = (i == 1) ?
    server[i].localInput(BobTable) : server[i].remoteInput(1);
    # define a list of clumns we want to select
    std::vector<SharedTable::ColRef>  Select_columns;
    for (u64 i = 0;  i < cols; ++i){
      Select_columns.emplace_back(SharedTable::ColRef(BobSharedTable,
    BobSharedTable.mColumns[i]));
    }
    for (u64 i = 1;  i < cols; ++i){
      Select_columns.emplace_back(SharedTable::ColRef(AliceSharedTable,
    AliceSharedTable.mColumns[i]));
    }
    # compute the join
    SharedTable JOIN =srvs[i].join( SharedTable::ColRef(AliceSharedTable,
    AliceSharedTable.mColumns[0]), SharedTable::ColRef(BobSharedTable,
    BobSharedTable.mColumns[0]), First_Select_columns);
    # reveal the plaintext values of the result
    aby3::i32Matrix result = RevealAll(server[i], JOIN);
  };
  auto AliceThread = std::thread(protocol, 0); start Alice's thread
  auto BobThread = std::thread(protocol, 1); start Bob's thread
  thread(protocol, 2); # start the assisting third party
  t0.join(); # wait for Alice to finish
  t1.join(); # wait for Bob to finish
}
```

Listing 5.4: Implementation of use case one using ABY3

SMCQL For SMCQL we also provide an overview how of SMCQL is used to obtain a result. Similar to Conclave and ABY3 in first we define the relation schema. We define the relation schema using the `CREATE TABLE` command. In Listing 5.5 we provide an example how the execution is done from Alice's perspective, Bob's perspective works analogue. After we have defined the layout of Alice's table, SMCQL automatically populates the table with input we previously generated. SMCQL's access control mechanism is not optimal, therefore we annotate each column as private. In the last step we parse the query towards SMCQL.

```
# Delete the Table and it's content from previous runs
DROP TABLE IF EXISTS Alice_use_case_one;
# Annotate Alice's table layout
CREATE TABLE Alice_use_case_one
 (Alice_use_case_one_id integer,
 Alice_use_case_one_1 integer,
 Alice_use_case_one_2 integer,
 Alice_use_case_one_3 integer);
# Annotate the acces controll
GRANT SELECT(Alice_use_case_one_id) ON Alice_use_case_one TO
  private_attribute;
GRANT SELECT(Alice_use_case_one_1) ON Alice_use_case_one TO
  private_attribute;
GRANT SELECT(Alice_use_case_one_2) ON Alice_use_case_one TO
  private_attribute;
GRANT SELECT(Alice_use_case_one_3) ON Alice_use_case_one TO
  private_attribute;
# parse the query to SMCQL
SELECT Alice_use_case_one_id, Alice_use_case_one_1, Alice_use_case_one_2,
 Bob_use_case_one_id, Bob_use_case_one_1, Bob_use_case_one_2
FROM A_use_case_one JOIN B_use_case_one ON
 A_use_case_one_id = B_use_case_one_id

```

Listing 5.5: Simplified Setup of SMCQL for our first use case

5.2.2 Use Case Two

For our second use case, we do not implement the query directly, as described in Listing 5.2 Instead, we apply an optimized query that has an identical output. To obtain the desired result, Alice and Bob compute two intermediate results, the union of these intermediate results will then yield our final result, an equivalent SQL statement is listed in Listing 5.6. In the first step, Alice and Bob filter their input once so that it contains only entries with the value false in its boolean column. Afterward, they can compute the join of these two filtered tables. The result of this join is the first important intermediate result, as each of its entries is also part of the final result. The second important intermediate result can be obtained by the same procedure when filtering the input for entries that contain true. The union of these two intermediate results is our final result.

We have chosen to implement the use case in this indirect way for two reasons. First, Conclave cannot evaluate the query directly, as Conclave's current prototype implementation can apply a WHERE filter exclusivity to a table that is either the direct input of a party or the output of a unary operator that has only one table as input. Additionally, our indirect implementation has a significantly better performance compared to the direct implementation, for a detailed analysis see our evaluation in Chapter 6.

```
SELECT PimaryKey, AliceBool, BobBool FROM
(SELECT PrimaryKey, AliceBool FROM AliceTable WHERE AliceBool == false)
NATURAL JOIN
(SELECT PrimaryKey, BobBool FROM AliceTable WHERE BobBool == false)
UNION
SELECT PimaryKey, AliceBool, BobBool FROM
(SELECT PrimaryKey, AliceBool FROM AliceTable WHERE AliceBool == true)
NATURAL JOIN
(SELECT PrimaryKey, BobBool FROM BobTable WHERE AliceBool == true)
```

Listing 5.6: Functional equivalent SQL statement for our optimitized implementation of our second use case

Correctness Despite it not being obvious, our procedure indeed yields the correct result, as each join generates only entries that have a primary key that is included in Bob's and Alice's inputs and have identical boolean values. Therefore, every entry that is part of our result is also part of the correct result.

If any entry is not part of our result, it cannot be part of either intermediate result. Any entry that is not part of either intermediate result features a primary key that is either not represented in both inputs or contains two boolean values that are not equal. Therefore, the entry cannot be part of the correct result. Consequently, each entry that is not part of our result, cannot be part of the correct result. Which is the contraposition and therefore equivalent to the fact that each entry of the correct result is part of our result. As our result contains every entry of the correct result and the correct result contains every entry of our result, our result and the correct result are identical. Therefore, our procedure is indeed correct.

Conclave In order to implement our second use case in Conclave, we start with the import of the input. We annotate the layout of each table. Conclave does not support a dedicated data type for boolean values, consequently, we use integers for every column. In the second step, we generate four new tables by applying the filters to the input. These filters are a showcase example of how Conclave can speed up computation by applying optimizations. A naive approach for applying the filters would be to share the input tables and then filter the shared tables using an MPC algorithm. This is an approach Conclave is in theory capable of. But Conclave is able to utilize the fact, that each filter has only a single input that is known to one party in the clear. In consequence, the parties can apply the filter to their input locally without the use of an MPC algorithm. After they have applied the filters, the parties can then create shared tables of the filtered input. In the next step compute the two intermediate results by joining the shared filter input of Alice and Bob. Here Conclave demonstrates how it supports composability, as the output of the filter function is used as input for the join function. Finally, we compute the row-wise concatenation of the two intermediate results and publish the final result to Alice and Bob.

```
def protocol():
```

```
AliceColumns = [
defCol("primary_key", "INTEGER", [1]),
defCol("AliceBool", "INTEGER", [1])
]
AliceTable = create("AliceTable", AliceColumns, {1})
BobColumns = [
defCol("primary_key", "INTEGER", [2]),
defCol("BobBool", "INTEGER", [2])
]
BobTable = create("input_2", AliceColumns, {2})
AliceFilterFalse = cc_filter(AliceTable, "AliceFilterFalse", "AliceBool",
   "==", scalar=0)
BobFilterFalse = cc_filter(BobTable, "AliceFilterFalse", "BobBool", "==",
   scalar=0)
AliceFilterTrue = cc_filter(AliceTable, "BobFilterTrue", "AliceBool", "==
  ", scalar=1)
BobFilterFalse = cc_filter(BobTable, "BobFilterTrue", "BobBool", "==",
  scalar=1)
intermediateFalse = join(AliceFilterFalse, BobFilterFalse, '
  intermediateFalse', ['primary_key'], ['primary_key'])
intermedidateTrue = join(AliceFilterTrue, BobFilterTrue , 'join_result1',
   ['primary_key'], ['primary_key'])
final_result = concat([ intermediateFalse , intermediateTure ], "
  final_result" )
collect(final_result, 1)
collect(final_result, 2)
```

Listing 5.7: Simpifiyed Protocol for our second use case in Conclave

ABY3 Our second use case is also a showcase example of how we use ABY3's integrated benchmarking capability to sample valuable information. With the in ABY3 integrated timer, we are able to set `TimePoints` at arbitrary potions of the protocol. After the execution is finished, the timer returns how much time is passed before each individual `TimePoints` was reached. For this reason, we obtain detailed information on how the different parts of the computation compose its total runtime. Since Conclave is able to apply the filter locally, in order to hold the comparison between them fair, we also filter the locally. Alice and Bob apply the filters to their input using standard C++ before the protocol begins. The majority of the work is done by two functions we present in Listing 5.8 and Listing 5.9. The first function generates the intermediate results, in order to do so, in the first step we load the input and convert it into a secret shared table. In the next step we calculate the `join` over the shared tables, to do so we need to pass a reference towards the two key columns, together with a list of the columns we want to select over. Our second function uses the first function to obtain the intermediate results. Once we have obtained those we compute their union.

```
# include the timer and instanciate it
include "cryptoTools/Common/Timer.h"
Timer t;
...
```

```
# generate input and apply filter to input locally, omited for reason of simplicty.
...
# i denotes the party id where i==0 means Alice, i==1 Bob and i==2 denotes the third assisting party
auto getIntermediate =[&] (int i, int filter){
  SharedTable AliceTable;
  SharedTable BobTable;
  #  Load the prefilterd input and convert it into a secret share.
  if (filter == 1) {
    AliceTable = (i == 0) ? srvs[i].localInput(AliceInputFiltertOne):
    srvs[i].remoteInput(0);
    BobTable = (i == 1) ? srvs[i].localInput(BobInputFiltertOne):
    srvs[i].remoteInput(1);
  }else{
    AliceTable = (i == 0) ? srvs[i].localInput(AliceInputFiltertZero):
    srvs[i].remoteInput(0);
    BobTable = (i == 1) ? srvs[i].localInput(BobInputFiltertZero):
    srvs[i].remoteInput(1);
  }# Here we annotate that we want to select the first two columns of Alice's Table and the second column of Bob's Table
  std::vector<SharedTable::ColRef>  First_Select_collumns;
  Select_columns.emplace_back(SharedTable::ColRef(BobTable, BobTable.mColumns[0]));
  Select_columns.emplace_back(SharedTable::ColRef(BobTable, BobTable.mColumns[1]));
  Select_columns.emplace_back(SharedTable::ColRef(AliceTable, AliceTable.mColumns[1]));
  # We calculate the join and return it
  return srvs[i].join( SharedTable::ColRef(AliceTable , AliceTable.mColumns[0]), SharedTable::ColRef( BobTable, BobTable.mColumns[0]), Select_columns);

};
```

Listing 5.8: Function for our second use case that generates an intermediate result

```
auto use_case2 = [&](int i, int filter)
{
  t.setTimePoint("start");
  SharedTable IntermediateOne = getIntermediate(i,0);
  t.setTimePoint("FirstIntermediate");
  SharedTable ItermediateZero = getIntermediate(i,1);
  t.setTimePoint("SecondIntermediate");
  SharedTable UNION;
  # Here we annotate which columns of the first Intermediate
  # will be part of the UNION
  std::vector<SharedTable::ColRef>  SelectOnes;
  std::vector<SharedTable::ColRef>  SelectZeros;
  for(u64 index=0; index <3; ++index)  {
    # Here we annotate that all columns of both intermediate results
    # will be part of the UNION
    SelectOnes.emplace_back(SharedTable::ColRef( IntermediateOne ,
    IntermediateOne.mColumns[index]));
```

```
      SelectZeros.emplace_back(SharedTable::ColRef(IntermediateZero ,
      IntermediateZero.mColumns[index]));
    }
    t.setTimePoint("UNION");
    # Compute the union of the two intermediate results
    UNION = srvs[i].rightUnion(SharedTable::ColRef( IntermediateZero,
    IntermediateZero.mColumns[0]),
    SharedTable::ColRef( intermediateOne, intermediateOne.mColumns[0]),
    SelectZeros,SelectOnes);
    t.setTimePoint("end");

  };
  auto t0 = std::thread(use_case2, 0,1); #Start Alice thread
  auto t1 = std::thread(use_case2, 1,1); #Start Bob thread
  use_case2(2,1); # Start the assisting third party
  t0.join(); # Wait for Alice to finish
  t1.join(); # Wait for Bob to finish
  write_to_file(t) # Write timer information to file for further analysis
```

Listing 5.9: Simpifiyed Protocol for our second use case in ABY3

5.2.3 Use Case Three

As our first and third use case are a nearly identical their implementation differs only in minor details. Especially for Conclave the implementation is identical besides the fact that we need to use different trust annotations. In order to allow Conclave to apply its public join optimization, we annotate that each party is allowed to learn both of the key columns.

```
def protocol():
# define the schema for the input tables
AliceColumns = [
# annotate that Alice trust Bob to learn hear key column
defCol("primary_key", "INTEGER", [1,2]),
defCol("user_data1_Alice", "INTEGER", [1]),
defCol("user_data2_Alice", "INTEGER", [1]),
defCol("user_data3_Alice", "INTEGER", [1]),
]
BobColumns = [
# annotate that Bob trust Alice to learn his key column
defCol("primary_key", "INTEGER", [1,2]),
defCol("user_data1_Bob", "INTEGER", [2]),
defCol("user_data2_Bob", "INTEGER", [2]),
defCol("user_data3_Bob", "INTEGER", [2])
]
...
...
collect(JOIN, 1)
collect(JOIN, 2)
```

Listing 5.10: The Python protocol of Conclave for our last use case

6 Evaluation

In this chapter we discuss the results of our benchmarks. We provide figures for the individual performance of the frameworks and highlight notable details. We evaluate the result of each use case individually, ordered from least complex to most complex. At the end of the chapter we discuss our results and provide our final conclusion.

Experimental Setup All our benchmarks are conducted on a single machine that hosts all parties. Our machine runs on a 64 bit Linux operations system, more precisely we use Debian 11 with the Linux Kernel version 5.10.0-18.amd64. The system features 4 CPUs that each run at 2,3 GHz and 128 GB of RAM. In order to fit the results of different frameworks into shared figures, we use a logarithmic scaling to base 10 for our figures. We always annotate the total amount of input-rows included in a computation. For example, if a computation involves two parties that each provide an input of 100 rows, we will annotate that as 200 rows of input. To obtain more reliable measurements, we perform each measurement multiple times, wherever possible, and select the median of our measurements as final result.

6.1 Use case One

For our first evaluation we are inspecting the result of our first use case in the localhost setting. Our first use case is a simple join for a detailed description see Section 5.1. We have visualised a comparison of runtime and space consumption in Figure 6.1 and 6.2. In the localhost setting the parties are connected with a nearly perfect LAN connection. At the end of the section we also test the frameworks with a WAN connection, in our WAN setup each connection has a simulated latency of 100 milliseconds and a simulated maximum bandwidth of 10000 KB/s.

6.1.1 LAN Setting

Required Runtime For SMCQL we have been able to evaluate the query for up to 140 input-rows. In order to compute the query with 140 input-rows SMCQL took about 10 hours. For Conclave we have been able to scale up to 500 input-rows, with 500 rows taking about 8 hours. Neither SMCQL nor Conclave could produce a result for larger datasets within 24 hours and after 24 hours we stopped the computation. A visitation of the result for our result can be found in Figure 6.1. We have observed that the difference in speed between SMCQL and Conclave grows significantly with increasing input size. For an input of size 20, Conclave is 5 times faster than SMCQL, for an input of size 100

Conclave already is more than 50 times faster and for an input of size 140, Conclave is more than 75 times faster than SMCQL. In the first use case ABY3 is by far the best scaling framework. ABY3 is able to compute the query with 140 and 500 input columns in less than a second. ABY3 runtimes does not increase in any significant way before its input reaches the mark of 128000 input-rows, for which it requires 1107 milliseconds. We have been able to evaluate the query with 16.000.000 input-rows in one minute and 46 seconds.

Evaluation of used Space It is notable that for large input sizes SMCQL is more space efficient than Conclave, while for small input input sizes Conclave is more space efficient than SMCQL. As shown by figure 6.2, they break even at an input size of 60 rows, for which they both require about 870 MB of space. For input larger than 60 rows SMCQL scales significantly better than Conclave. For an input of size 140 rows SMCQL only needs about 1300MB while Conclave needs more than 4 times as much. As the space consumption of SMCQL grows linear and the space consumption of Conclave doubles every time the input-size grows by 100, we can only assumes that this trend would continue for large datasets. Similar to the case of the runtime, ABY3 also performs significantly better than Conclave and SMCQL. For input size 140, ABY3 only allocates 28MB of space and therefore is over 46 times more efficient than SMCQL. For input size 500, ABY3 allocates about 32 MB of space and is therefore 1800 times more efficient than Conclave, which requires over 58 GB of space.

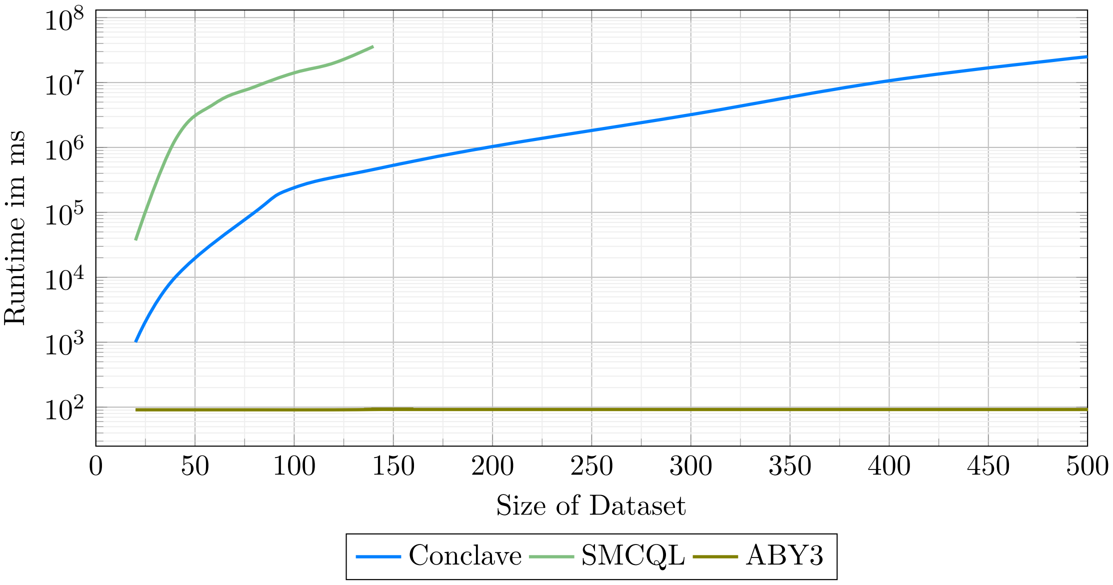

Figure 6.1: Runtime of ABY3, Conclave and SMCQL for in our first use case

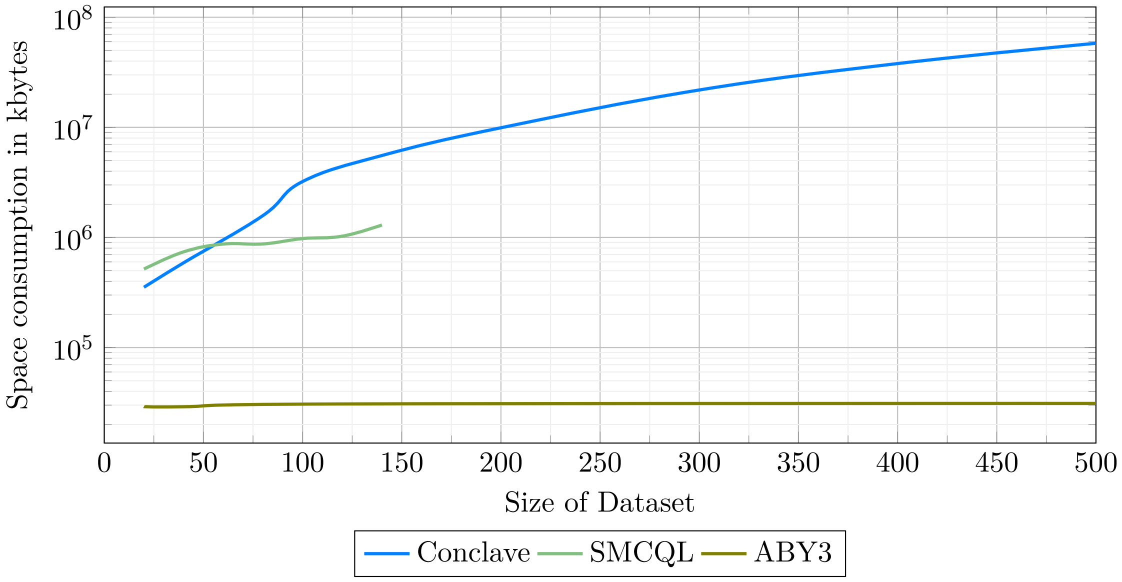

Figure 6.2: Memory usage of ABY3, Conclave and SMCQL in our first use case

6.1.2 WAN Setting

For both Conclave and SMCQL we have observed a significant reduction in performance. For SMCQL we have been able to obtain a result for an input of size 80 within 5 hours and 54 minutes, for Conclave we obtained a result for an input of size 120 within 4 hours. Neither of the two frameworks produced an output for larger inputs within 10 hours and after 10 hours we have terminated to computation. Just like in the LAN setting, ABY3 was able to compute a correct result for an input of size 16.000.000. Yet we also observed an reduction in performance, because ABY3 was significantly slower in the WAN setting. As Figure 6.4 depicts ABY3 was constantly between 10 and 4 times slower in the WAN setting than in LAN setting. The factor between the performances shrink with increasing input size. For example for an input of size 4.000.000 the LAN setting takes 25000 ms and the WAN setting requires 110000 ms, while for an input of size 128.000 the WAN setting is 10 times slower than the LAN setting. The largest loss in performance has been demonstrated by Conclave. As we show in Figure 6.3 Conclave is for an input of size 40 more then 113 times faster in the LAN setting than in WAN setting. With increasing input size the difference between the LAN and the WAN does not change in a significant way, as for an input of size 80 Conclave is 116 times faster in the WAN than in the LAN setting. For SMCQL we have observed an similar, yet better performance like Conclave. We summarise that ABY3 proved to be the most resilient to unfavourable networking conditions, while Conclave showed to be the least resistant. Even moderate networking conditions have a signifiant impact on the performance of any of the frameworks.

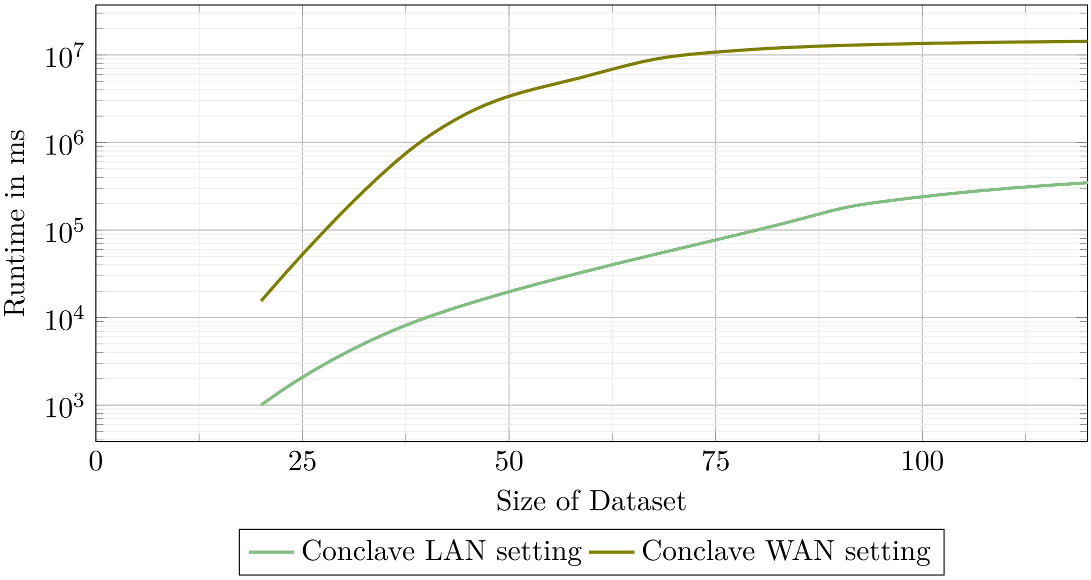

Figure 6.3: Runtime of Conclave for use case one in LAN and WAN setting

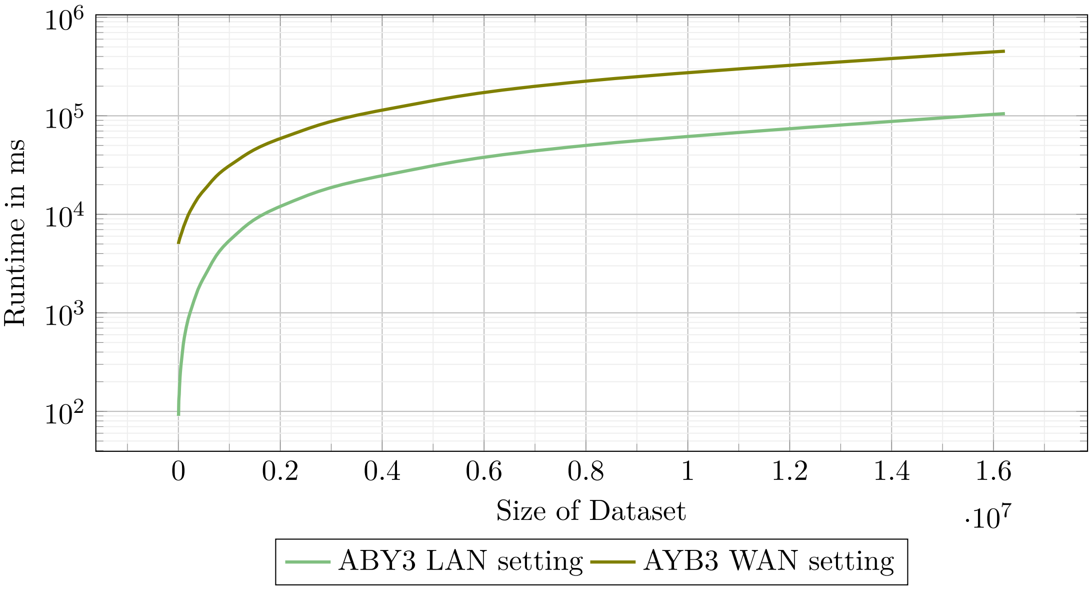

Figure 6.4: Runtime of ABY3 for use case one in LAN and WAN setting

6.2 Use Case Two

In Section 5.2 we have described an optimized way to evaluate the query of our second use case. In order to see if how large our efficiency gain is in practice, we have implemented this use case for ABY3 twice. One implementation is optimised to evaluate the query indirectly, such that in obtains a result with two joins of size n/2, a `union` operation over to inputs of size n/2 and a local application of filters, the other implementation is fully unoptimized and obtains a result with one join of size n and an application of filter with an MPC algorithm. Because our other result are very similar to the results of use case one, we focus on a comparison between the two implementation of ABY3.

Evaluation of Runtime As Figure 6.5 depicts, our measurement shows that for small input sizes the unoptimised implementation is faster, while for larger input sizes the optimized implementation is faster. For input size 2000, the unoptimised implementation runs within 182 milliseconds, which is 1,5 times faster then our optimised implementation, which needs 276 milliseconds. With increasing input size, the difference between their performances becomes less significant, for input size 8000 the unoptimized implementation is less then 1,4 times faster. For an input of size 64000 the two implementation have very similar runtime and both need about 500 milliseconds. From there on onwards, our optimization starts to pay of and is constantly about 1,5 times faster then the unoptimised version. A disadvantage of our optimization is that it requires the use of an additional `union` operator to concatenate the two important intermediate results, which is not needed in the naive implementation. The reason that our optimization is less efficient in small datasets is that the additional cost of the union operator exceeds the savings of the optimizations. This thesis is supported by the collected data. With the ability of cryptoTools to measure individual parts of the protocol, we are able to measure the contribution of the `union` operator to the total runtime precisely. In small datasets the `union` operator contributes more then a third of the total runtime. For example, for an input of size 400, it contributes 90 out of 275 milliseconds. Its contribution becomes less significant with increasing input size. For an input of size 1 million the `union` operator only contributes about 10 % of the total runtime. Therefore we conclude that in small datasets the `union` operation is an expensive operation, while in large datasets it cost is an insignificant additional factor compared to the cost of the other operations.

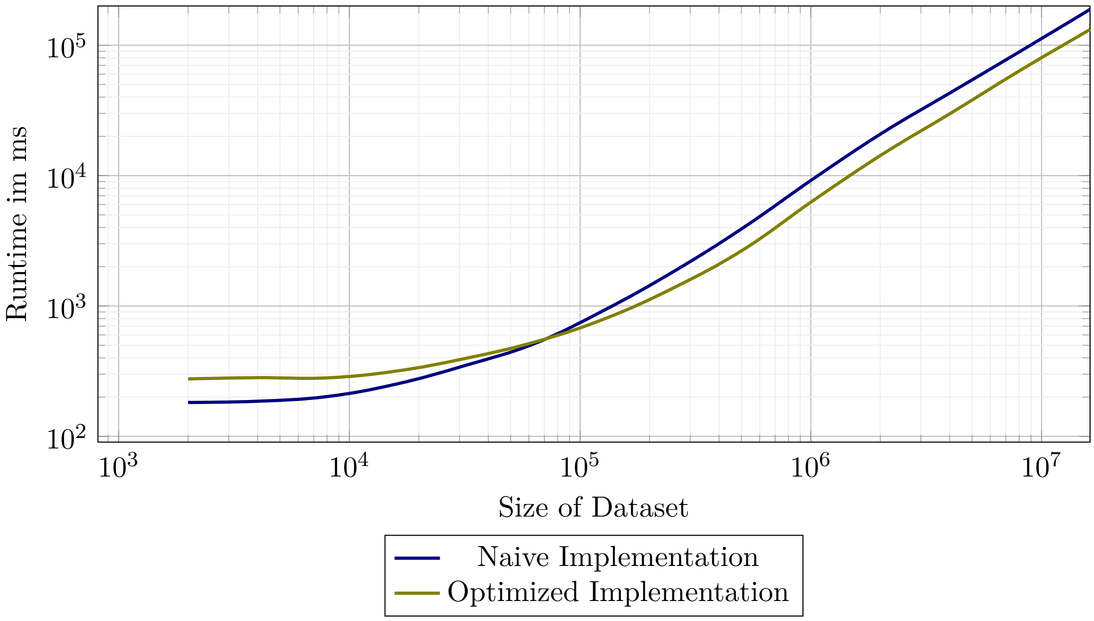

Figure 6.5: Runtime of our two implementations of ABY3 use case two

Evaluation of Space From the perspective of memory consumption our optimization is a strict improvement over the unoptimized implementation. The optimized implementation is strictly better, as for every single input size we have tested, it requires less memory. We visualise the memory requirements of our two implementations in Figure 6.6. It is notable how similar both implementations scale. For large inputs, they both very reliably double their memory consumption each time their input is doubled and the optimized implementation consistently takes half the memory of the unoptimised implementation. This trend start with input of size 64000 where our optimization needs about 300MB and the naive implementation about 150MB and continues from there on.

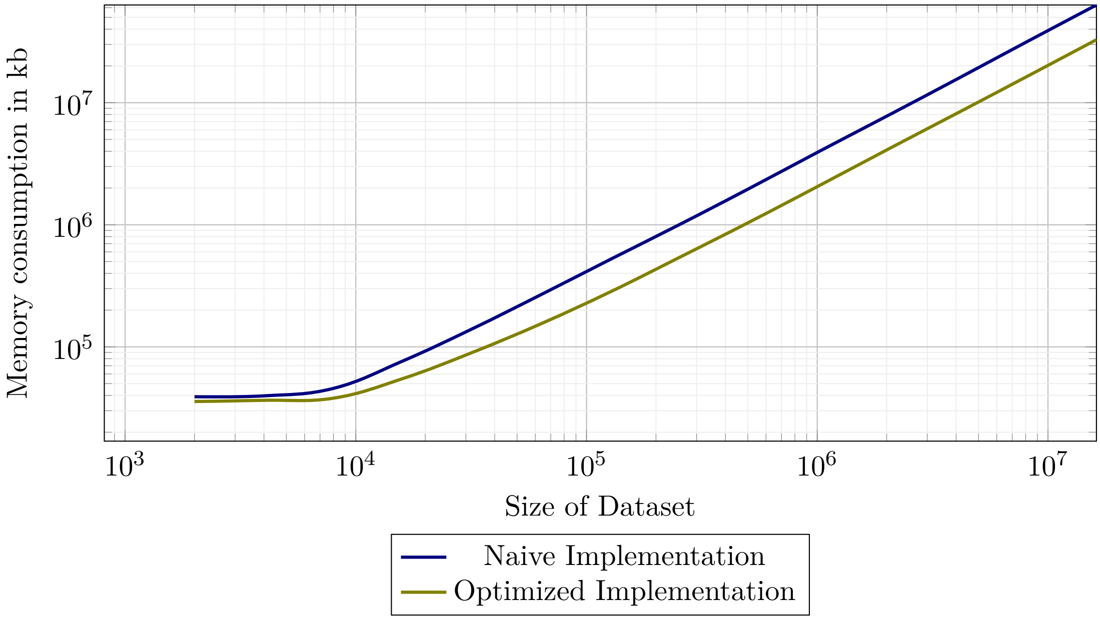

Figure 6.6: Space requirement of our two implementations using ABY3 in use case two

Comparison between Conclave and ABY3 In our second use case Conclave scales better than in the first one. For a visualisation of its results see Figure 6.7. We have been able to obtain a result for 1000 input columns in 6 hours and 32 minutes. Compared to the 500 input columns in 8 hours from the first use case, Conclave has been able to handle an input twice as large in less time. Yet despite these better results Conclave is still unable to compete with the results of our two ABY3 implementations for the second use case results, that both have been able to compute the result for 1000 input-rows in less then a second.

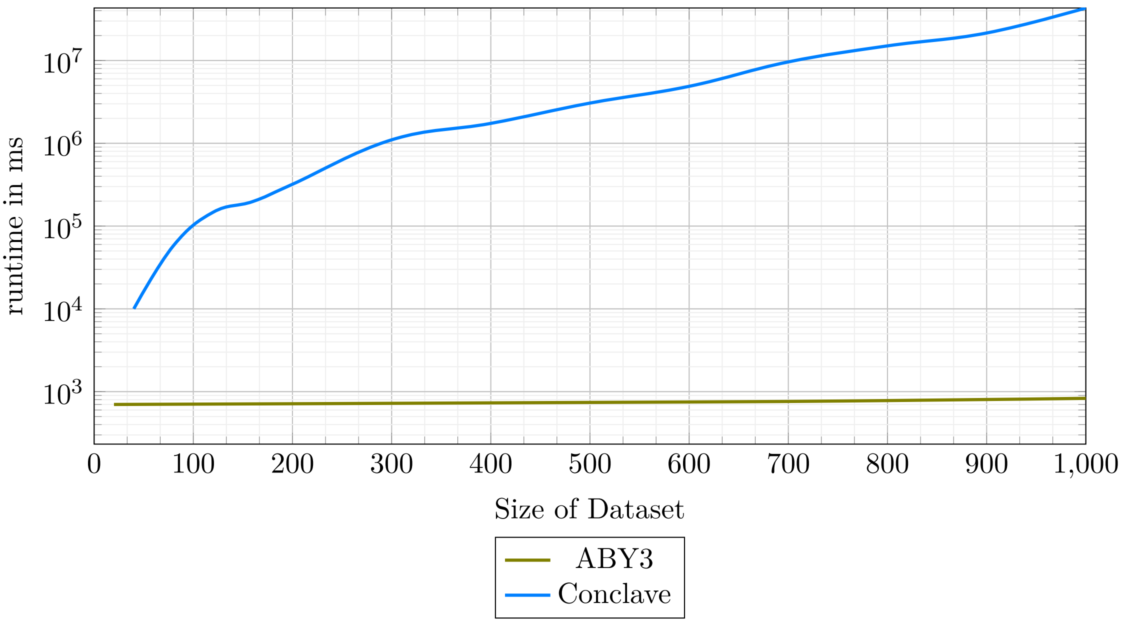

Figure 6.7: Comparison between Conclave and ABY3 implementation in our second use case

6.3 Use Case Three

As we described in Section 5.1 our third use case has an identical semantic to our first use case but we make use of Conclave's trust annotations, that allow the leakage of some of the input data, which allows Conclave to apply optimizations that speed up the computation. Therefore we focus on a comparison between the performance of Conclave in the first and third use case. At the end out of the section we also compare the first implementation of ABY3 with the implementation of use case three with Conclave to see if Conclave yields comparable result if it has the advantage of its trust annotations. A visualisation of our comparison can found in Figures 6.8 and 6.9.

Comparison between Use Case One and Use Case Three As Figure 6.8 depicts, with the usage of trust annotation we have observed an significant improvement in speed and efficiency. We have been able to evaluate the query for input-sizes of up to 3000 rows, which are 6 times larger then in use case one. For input-sizes larger then 3000 Conclave tends to crash because of internal errors that are outside of our control. Therefore we were unable to obtain result for larger inputs. For an input of size 500 in use case three, Conclave needs less then 5 minutes which more then 80 times faster then the 8 hours required in use case one. For an input of size 3000, Conclave is able to compute the correct result in less then 25 minutes. From the perspective of memory requirement, the situation is very similar, as Figure 6.9 depicts, Conclave requires less memory in use case three then in use case one, for all input-sizes we observed.

Comparison between ABY3 and Conclave It is notable that even with the unfair advantage of leaking some input data, Conclave is not able to yield similar performance to ABY3. The implementation of use case one with ABY3, which has no such advantage, is still multiple orders of magnitude faster than Conclave in use case three and also requires significantly less memory. On average ABY3 is more than 360 times faster. The difference in speed becomes more significant with larger input-sizes, in the extreme case of input-size 3000 ABY3 is more then 1300 times faster than Conclave.

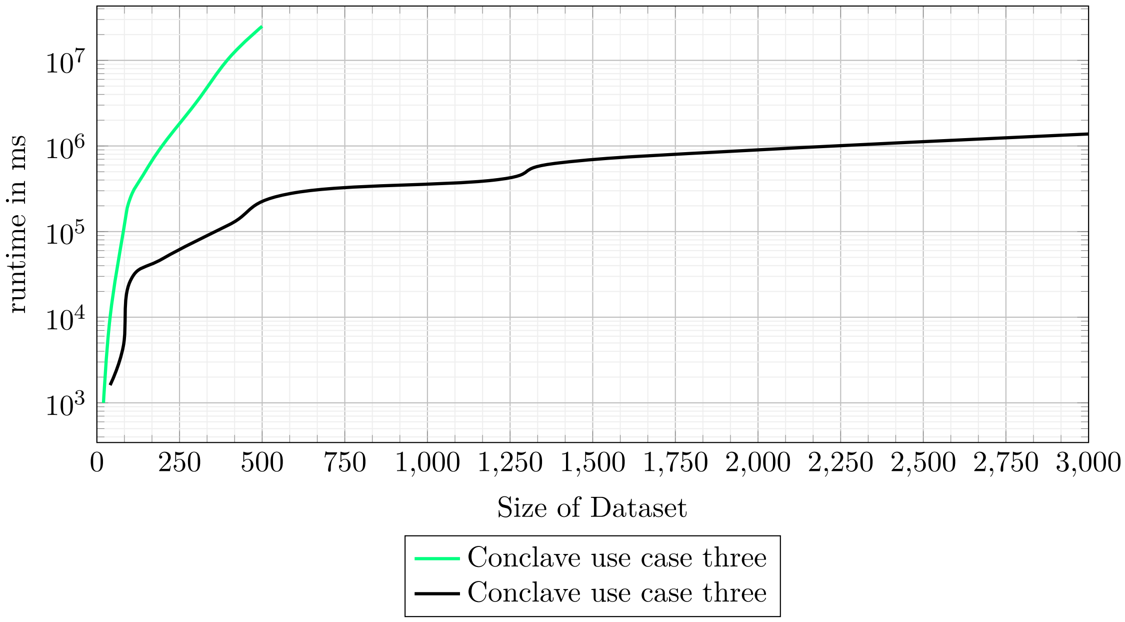

Figure 6.8: Runtime of Conclave in use case one and use case three

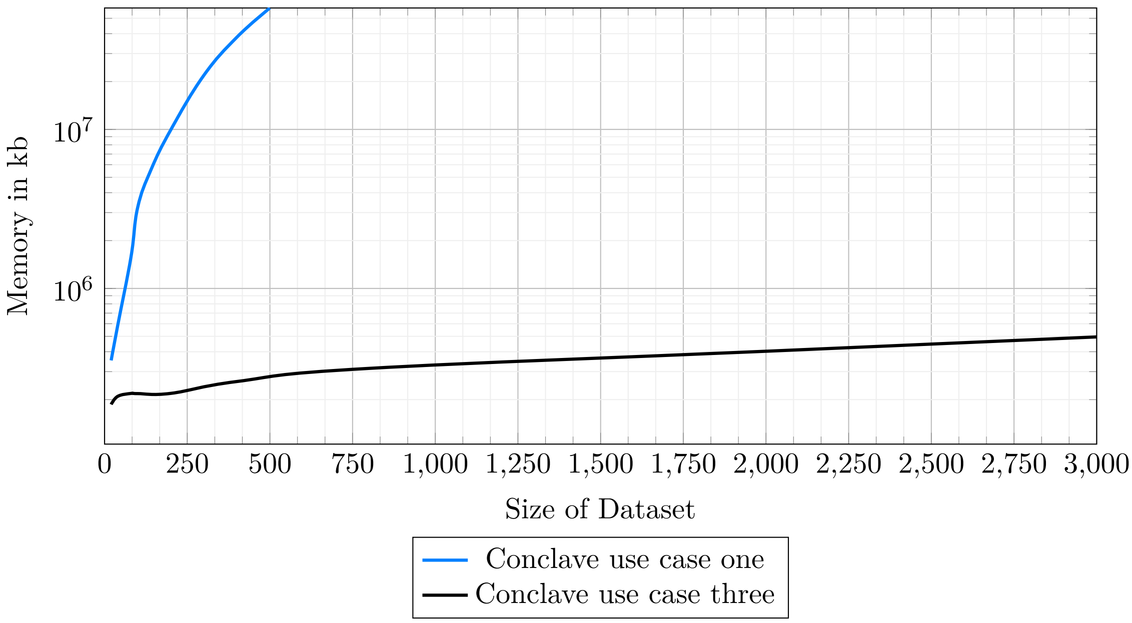

Figure 6.9: Comparison between Conlave's space consumption in use case one and use case three

7 Conclusion

In our work we have implemented several different use cases for the secure multiparty computation frameworks ABY3, SMCQL, Conclave. We encountered problems like missing documentation and complex installation procedures. However, we have able to overcome them and therefore demonstrated the feasibility of the task. In particular, the reasons for problems like the lack of documentation are not inherent properties of multiparty computation, but limited time and resources of the frameworks' authors. From the frameworks we used in our tests, only ABY3 was able to demonstrate the capability to handle input sizes of practical relevance, as ABY3 has demonstrated that it is able handle millions of input rows running on moderate hardware. Conclave and SMCQL could only handle a few hundred or thousand input rows. It is notable that each generation of frameworks was more efficient the previous one, as Conclave was more efficient than SMCQL, which was the oldest framework, and ABY3 was more efficient than Conclave, the second oldest framework. One aspect of relational databases we could not explore in depth ,due to limited time, are aggregate functions. Therefore, a survey focused on aggregate functions remains an open task for further research. Since we have only considered open-source applications, all frameworks included in our study have been developed in an academic context. It remains an open task for future research to evaluate similar frameworks developed in an industrial context, such as CipherCompute or Sharemind [BLW08], and evaluate how efficient they are and if they have similar usability problems.

Bibliography

[ABL$^+$18] David W. Archer, Dan Bogdanov, Yehuda Lindell, Liina Kamm, Kurt Nielsen, Jakob Illeborg Pagter, Nigel P. Smart, and Rebecca N. Wright. From keys to databases - real-world applications of secure multi-party computation. *Comput. J.*, 61:1749–1771, 2018.

[ACC$^+$21] Abdelrahaman Aly, K Cong, D Cozzo, M Keller, E Orsini, D Rotaru, O Scherer, P Scholl, N Smart, T Tanguy, et al. Scale–mamba v1. 12: Documentation, 2021.

[AGJ$^+$21] Surya Addanki, Kevin Garbe, Eli Jaffe, Rafail Ostrovsky, and Antigoni Polychroniadou. Prio+: Privacy preserving aggregate statistics via boolean shares. Cryptology ePrint Archive, Report 2021/576, 2021. `https://ia.cr/2021/576`.

[AIL$^+$19] Kinan Dak Albab, Rawane Issa, Andrei Lapets, Peter Flockhart, Lucy Qin, and Ira Globus-Harris. Tutorial: Deploying secure multi-party computation on the web using jiff. *2019 IEEE Cybersecurity Development (SecDev)*, pages 3–3, 2019.

[BEE$^+$16] Johes Bater, Gregory Elliott, Craig Eggen, Satyender Goel, Abel Kho, and Jennie Rogers. Smcql: Secure querying for federated databases. *arXiv preprint arXiv:1606.06808*, 2016.

[BJSV15] Dan Bogdanov, Marko Jõemets, Sander Siim, and Meril Vaht. How the estonian tax and customs board evaluated a tax fraud detection system based on secure multi-party computation. In *International conference on financial cryptography and data security*, pages 227–234. Springer, 2015.

[BLW08] Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A framework for fast privacy-preserving computations. In *European Symposium on Research in Computer Security*, pages 192–206. Springer, 2008.

[CCLW20] Jamie Cui, Chaochao Chen, Alex X Liu, and Li Wang. Secure graph database search with oblivious filter. *Cryptology ePrint Archive*, 2020.

[CD05] Ronald Cramer and Ivan Damgård. Multiparty computation, an introduction. In *Contemporary cryptology*, pages 41–87. Springer, 2005.

[CGB17] Henry Corrigan-Gibbs and Dan Boneh. Prio: Private, robust, and scalable computation of aggregate statistics. In *14th USENIX Symposium on*

Networked Systems Design and Implementation (NSDI 17), pages 259–282, Boston, MA, March 2017. USENIX Association.

[DPSZ11] I. Damgard, V. Pastro, N. P. Smart, and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. Cryptology ePrint Archive, Paper 2011/535, 2011. `https://eprint.iacr.org/2011/535`.

[DW82] Danny Dolev and Avi Wigderson. On the security of multi-party protocols in distributed systems. In David Chaum, Ronald L. Rivest, and Alan T. Sherman, editors, *Advances in Cryptology: Proceedings of CRYPTO '82, Santa Barbara, California, USA, August 23-25, 1982*, pages 167–175. Plenum Press, New York, 1982.

[HHNZ19] Marcella Hastings, Brett Hemenway, Daniel Noble, and Steve Zdancewic. Sok: General purpose compilers for secure multi-party computation. In *2019 IEEE symposium on security and privacy (SP)*, pages 1220–1237. IEEE, 2019.

[JNS17] Noah M Johnson, Joseph P Near, and Dawn Xiaodong Song. Practical differential privacy for sql queries using elastic sensitivity. *CoRR, abs/1706.09479*, 2017.

[KKJ$^+$13] Himanshu Kumar, Sudhanshu Kumar, Remya Joseph, Dhananjay Kumar, Sunil Kumar Shrinarayan Singh, Ajay Kumar, and Praveen Kumar. Rainbow table to crack password using md5 hashing algorithm. In *2013 IEEE Conference on Information Communication Technologies*, pages 433–439, 2013.

[Lin17] Yehuda Lindell. How to simulate it–a tutorial on the simulation proof technique. *Tutorials on the Foundations of Cryptography*, pages 277–346, 2017.

[LWN$^+$15] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. Oblivm: A programming framework for secure computation. In *2015 IEEE Symposium on Security and Privacy*, pages 359–376. IEEE, 2015.

[MRR20] Payman Mohassel, Peter Rindal, and Mike Rosulek. Fast database joins and psi for secret shared data. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1271–1287, 2020.

[PSZ14] Benny Pinkas, Thomas Schneider, and Michael Zohner. Faster private set intersection based on {OT} extension. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 797–812, 2014.

[RAB$^+$22] Jennie Rogers, Elizabeth Adetoro, Johes Bater, Talia Canter, Dong Fu, Andrew Hamilton, Amro Hassan, Ashley Martinez, Erick Michalski, Vesna

Mitrovic, et al. Vaultdb: A real-world pilot of secure multi-party computation within a clinical research network. *arXiv preprint arXiv:2203.00146*, 2022.

[VSG+19] Nikolaj Volgushev, Malte Schwarzkopf, Ben Getchell, Mayank Varia, Andrei Lapets, and Azer Bestavros. Conclave: Secure multi-party computation on big data. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[WMK16] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. EMP-toolkit: Efficient MultiParty computation toolkit. `https://github.com/emp-toolkit`, 2016.

[Yao82] Andrew C. Yao. Protocols for secure computations. In *23rd Annual Symposium on Foundations of Computer Science (sfcs 1982)*, pages 160–164, 1982.

[ZE15] Samee Zahur and David Evans. Obliv-c: A language for extensible data-oblivious computation. *Cryptology ePrint Archive*, 2015.