



Title

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
???

Paderborn, September 27, 2022

Eidesstattliche Versicherung

Nachname: _____ Vorname: _____

Matrikelnr.: _____ Studiengang: _____

☐ Bachelorarbeit ☐ Masterarbeit

Titel der Arbeit: Title

☐ Die elektronische Fassung ist der Abschlussarbeit beigelegt.

☐ 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 Studienganges der Universität Paderborn, für die Verfolgung und Ahndung der Geldbuße ist die Vizepräsidentin für Wirtschafts- und Personalverwaltung.

Contents

| | | |
|----------|--|-----------|
| 1 | Abstract | 1 |
| 2 | Introduction | 3 |
| 2.1 | MPC and Databases | 3 |
| 2.2 | related work | 3 |
| 2.2.1 | From Keys to Databases—Real-World Applications of Secure Multi-Party Computation | 3 |
| 2.3 | goals | 3 |
| 2.4 | structure | 3 |
| 3 | Preliminary | 5 |
| 3.1 | Secure Multiparty Computation | 5 |
| 3.1.1 | Real World and Ideal World | 5 |
| 3.2 | Adversarial Models | 6 |
| 3.2.1 | Additional Properties | 7 |
| 3.3 | Databases | 7 |
| 4 | framework description | 9 |
| 4.1 | Conclave | 9 |
| 4.1.1 | Optimizations | 10 |
| 4.2 | ABY3 | 11 |
| 4.2.1 | functionality | 11 |
| 4.2.2 | underlying MPC technology | 12 |
| 4.3 | SMCQL | 13 |
| 4.3.1 | Optimizations | 14 |
| 4.3.2 | Comparison between SMCQL and Conclave | 15 |
| 4.4 | Rejected Frameworks | 15 |
| 5 | Benchmarking | 17 |
| 5.1 | Measuring Runtime | 17 |
| 5.2 | Networking | 18 |
| 5.3 | Measuring Space | 20 |
| 6 | Use-Cases | 21 |
| 6.1 | Use-Case Description | 21 |
| 6.2 | Implementation | 22 |
| 7 | Evaluation | 27 |

1 Abstract

2 Introduction

2.1 MPC and Databases

A famous problem in the context of MPC is Yao's millionaire's problem. In Yao's millionaire's problem there are two millionaires Alice and Bob. We will call Alice's wealth x and Bob's wealth y . Alice and Bob want to know who of them has more money. i.e. they want to compute the function $F(x,y) := \begin{cases} \text{Alice is richer} & y \leq x \\ \text{Bob is richer} & y > x \end{cases}$. Yet neither of them is willing to trust the other and tell him how much money he has. Yao's millionaire's problem can be generalised into the general MPC problem. Instead of Bob and Alice, we now consider n parties p_0, \dots, p_{n-1} and each party i holds an arbitrary input x_i for an arbitrary function $F(x_0, \dots, x_{n-1})$, that all parties have agreed upon. A MPC protocol π is protocol, that allows p_0, \dots, p_{n-1} to compute $F(x_0, \dots, x_{n-1})$ without revealing any information about x_0, \dots, x_{n-1} .

Andrew Yao proposed a solution for Yao's millionaire's problem in 1982 [1]. It has also been shown that MPC is Turing-complete[2]. This means that for any function f that can be computed with a Turing machine. There exists a MPC protocol π that can compute f . -Databases ...

2.2 related work

2.2.1 From Keys to Databases—Real-World Applications of Secure Multi-Party Computation

2.3 goals

In this section we describe the goals of our work.

2.4 structure

In this section we outline the structure this document.

3 Preliminary

In this chapter we describe

3.1 Secure Multiparty Computation

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

3.1.1 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.

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 relative weak adversary that can only read messages to a very powerful adversary. 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 consist of the inputs of each corrupted party, their obtained messages throughout the protocol and A 's obtained messages. The protocol achieves the security goal of confidentiality if the view A contains no more information beyond what can be deduced from the corrupted parties' input and outputs alone.

Ideal World In the ideal world the parties do not need run a protocol. Instead they 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 send P its input. P now holds all information it needs to compute F . Afterwards P can send each party the result in a second round of communication. Like in the real world,

Schaubild einfügen ?

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 sends 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.

Security Given a real-world adversary A , a secure multiparty 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 outputs a real world view. 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 against if, for all A π is secure against A .

3.2 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 can not 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, more precisely the set of all abilities an active adversary has, is a super set of the set of all abilities a passive adversary has. Furthermore 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 chooses 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 choose 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.

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. Because monolithic adversary is more powerful then multiple adversary's that control the same total amount of parties but to not corporate 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 adversary's that corrupt n parties total and do not coordinate their efforts. In the following we will assume a monolithic adversary unless explicitly stated otherwise.

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 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. A general adversary is limited in his choice which party he corrupts by an adversary structure $Z = \{Z_1, \dots, Z_l\}$. 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 want to assume that no adversary can corrupt both of these parties, that can be formalised by using a general adversary an 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 adversary's. A static adversary is bound 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. As he can try to identify "weak links" based on the information he gets during the execution of the protocol and then choose corrupt those.

3.2.1 Additional Properties

binary secret sharing

garbled circuits

3.3 Databases

coming soon

4 framework description

In this chapter we describe the different MPC frameworks that were 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. First we describe the functionality they provide. Afterwards we describe the MPC technology they utilise to realise their functionality. At last 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.

4.1 Conclave

functionality 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 relative 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, standard deviation.

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 concrete frameworks of use are Sharemind and Obliv-C. As both of these frameworks are designed to withstand passive adversary's and do not support more than 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. 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 threshold adversary that corrupts statically and is bound by an honest majority.

Documentation Another important property of a framework is its documentation. Conclaves documentation features three key components

4.1.1 Optimizations

MPC techniques are multiple orders of magnitude slower than cleartext processing. Its Conclave key principle to achieve better performance by avoiding the use of MPC techniques 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 - moving operations outside of MPC to maximise performance
- maintaining same end-to-end security as "pure" MPC
- contrary to conventional sql operations that aims to minimize the total amount of work e.g. filters before join

Trust Annotations Conclave features optional trust annotations that allow for 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 these mechanisms. For example, the sensitivity of data may largely differ between columns. Therefore it may be desirable, for a party to reveal some less sensitive 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 such optimization are Conclave's hybrid operators.

Hybrid Operations When possible Conclave substitutes expansive 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 its normal 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, does contain the STP but no other party, or does not contain the STP and could be withstood by a normal operation.

Sorts and Shuffles Many of Conclaves high-level operators include "sub-protocols" like sorts and shuffles. These sorts and shuffles are MPC operations. As such they are highly expansive operations. Yet not all of these sorts and shuffles are always necessary. If for example an operator produces a sorted intermediate result like for example 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 expansive MPC operations provides significant performance gains.

- published in 2019,
- compares to "SMCQL most similar existing system"
- jiff dependency
- requires python 3.5
- ... - no secure channel setting

4.2 ABY3

ABY3 [Rin] is a three party MPC framework that is based on secret sharing and uses new join algorithms to compute relational database joins in a efficient way.

4.2.1 functionality

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. Therefore 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 query's that have a comparable semantic to the "SELECT ... FROM ... WHERE" statement in SQL. For example a selection like "select X1 from X where X2 > 42" 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 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.

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 illegitimacy 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.

Documentation An important characteristics of every software is its documentation. Probably the most common form of documentation for any software are comments within the code, in this aspect ABY3 provides some documentation but has plenty room to improve, as on one hand it features classes that are very well commented, while on the other hand 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 capability's 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 important details 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.

4.2.2 underlying MPC technology

ABY3 works within a 3 party setting with a honest majority. This is a conscious decision as the two party and tree party setting 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 [Rin]. ABY3 guarantees security against a semi-honest 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 form can be archived by omitting the final reconstruction step, that would transform the output into the clear. Therefore having a secret shared based protocol be composable can be archived 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 be extended to be composable in trivial way [Rin].

ABY3's 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.

Cuckoo Hashing 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 $h_0, h_1: X \mapsto \{1, \dots, m\}$. 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 $T[h_i(x)]$ contains already an element the element is removed from the hash table and afterwards reinserted. This approach yields an important invariant(1): 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.

Computing Joins 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 j such that $X_1[i] = Y_1[j]$, 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 . 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 $T[h_1(Y_1[i])]$. 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 $h_1(X_1[i]) = h_1(Y_1[y])$. With invariant(1) 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 a match can be found by comparing $Y[y]$ to $T[h_0(Y_1[y])]$ and $T[h_1(Y_1[y])]$ in a secure way.

To 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 for the hash table are the keys for 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.

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 [Rin].

4.3 SMCQL

SMCQL [BEE⁺16] is an MPC based framework for relational database operations that is based on an already existing MPC framework, namely OblivM [LWN⁺15]. With SMCQL one can specify a query and SMCQL automatically generates secure code for evaluating the query.

functionality 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. With its SQL like Syntax SMCQL can evaluate every query consisting of a combination of these operators that would be a valid query in plain-standard SQL.

Documentation

underlying MPC technology SMCQL currently works in a two-party setting and provides security against a semi-honest, threshold adversary that can corrupt at most one party. So, it essentially resides in a 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 does always include only the party's 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 nodes annotates that the incoming node consumes the output data of the outgoing node. With the operator graph, the honest broker is able to analyze 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 4.3.4. 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.

OblivM TODO hier beschreibung von OblivM und ORAM

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 its 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 expansive MPC operations.

4.3.1 Optimizations

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

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 in two phases. First, a 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 split, 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.

4.3.2 Comparison between SMCQL and Conclave

4.4 Rejected Frameworks

CypherCompute One candidate for our study was CypherCompute [Cos21]. With the CypherCompute framework it is possible to solve a huge range of MPC problems using Rust. 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 well-known SPDZ [DPSZ11] protocol. Unfortunately the early access version of CypherCompute is out of maintenance by the time we conducting this study. Therefore we have decided to not include CypherCompute in our study.

Prio+ Prio+ [AGJ⁺21] is the next generation of the highly influential Prio [CGB17]. 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 CypherCompute 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] - uses EMP toolkit [WMK16] - demonstrates proof of concept

5 Benchmarking

For benchmarking performance 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 clean results.

5.1 Measuring Runtime

A basic metric of how well a program functions, 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 program blocks for a longer period of time its wall-clock and process execution time may largely differ. 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.

Conclave For python3 the two primary candidates are `timeit` [tim21b] and the python profiler [cPr21], both are python libraries that offer a simple way to measure wall-clock or process execution time. `Timeit` measures only end-to-end execution time, The python profiler comes with a more detailed analysis that includes detailed 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 to slows the execution down. Therefore using it would result in an unfair disadvantage for Conclave, thus we have decided to use `timeit` to measure the execution time of Conclave.

ABY3 Since ABY3 is based on C++ we can not use python specific tools for it we use for Conclave. Fortunately the `cryptoTools` library [Ran21] is integrated into ABY3. `CryptoTools` is a C++ toolbelt to 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 end-to-end execution time or to measure the execution time of specific parts of the protocol. As `croytoTools` provides to functionality

we need and is already "inbuilt" into ABY3, we have decided to use it for measuring ABY3's runtime.

SMCQL TODO time [tim21a]

5.2 Networking

In our standard setup all parties run on the same machine and communicate through localhost. This simulates a practically perfect LAN connection with very low latency and high throughput. It is also of interest how well the frameworks function in less ideal conditions. Therefore we are 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 needs to do is delaying incoming messages. Setting up such a proxy server is 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. One the other hand in Conclave, every party is connected to a Node.js server that forwards the messages. In order handle these various approaches correctly, an analysis of the communication patterns is required. 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 [wir21] is an open source packet sniffer that allows to capture network traffic and save it for a detailed analysis.

- überprüfen ob proxy correct arbeitet

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 and 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 [Sho22]. Toxiproxy is a Go framework that allows to simulate different hazardous network conditions. These include a connection that delays it 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(hypertext transfer protocol) interface, Toxiproxy provides multiple different dedicated HTTP clients for this purpose. The clients differ in that they offer an interface in different programming language . We have chosen to use the provided Ruby client as it is the one that provides the . A simplified example how to use Toxiproxy to simulate latency can be found in Listing 5.1. In order to use Toxiproxy on first needs to set up the proxy so it starts to accept new connections. This is done

by calling Toxiproxy populate and specifying to address the proxy listens to and the address the messages get forwarded to. By default Toxiproxy does not add any network hazards to a connection. In our example we apply two hazards, to simulate a latency of 1000ms. The first hazard 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 hazard is allied to the "downstream". Therefore it affects every response that comes from the address the server listens to.

```

0  #First we instantiate a connection between the two parties.
   Toxiproxy.populate([
2  {
   name: "aby3_party2_party1",
4  #party 3 sends its messages for party1 to port 50010 therefore the proxy
   must listen to this port
   listen: "127.0.0.1:50010",
6  #party 1 listens to port 50001 therefore theproxy must forward to this
   port
   upstream: "127.0.0.1:50001"
8  }
   ])
10 #Then we simulate a latency of 1000ms
   toxiproxy-cli add aby3_party2_party1 -type latency -name upstream latency
   =1000 -upstream
12 toxiproxy-cli add aby3_party2_party1 -type latency -name downstream
   latency=1000 -downstream
14

```

Listing 5.1: 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 send and received over a connection. Node-http-proxy is based on JavaScript and relies heavily on a event driven programming paradigm. A simplified example how to use node-http-proxy can be found in Listing 5.2. In our 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 messages for 500 milliseconds and the send 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. The event is raised before the message is transmitted to its destination and we can submit a anonymous function as event handler.

The event handler is executed for each message before the message is transmitted. We could for example use this mechanism to modify responses or for various other practical use cases. In our concrete 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 through the websocket. This is a showcase example, how a proxy can help to measure important metrics that would be non trivial to measure otherwise.

```
0  #create proxy server and specify the address it forwards to
1  #with the ws:true parameter we enable support for websockets
2  var proxy = new httpProxy.createProxyServer({
3      target: {
4          host: 'localhost',
5          port: 9005
6      },
7      ws: true
8  });
9  # Here we create a standard HTTP server that delays every incoming
10 # message for 500ms and then forwards it to the proxy server.
11 var proxyServer = http.createServer(function (req, res) {
12     setTimeout(function () {
13         proxy.web(req, res);
14     }, 500);
15 }).listen(9000);
16 # for every outgoing message the proxy emits a proxyReqWs event that we
17 # react to and delay the message for 500 milliseconds
18 proxy.on('proxyReqWs', function () {
19     setTimeout(function () { }, 500);
20 });
21 # each time a connection is closed the proxy emits a "close" event that
22 # we react to and save the amount of bytes transmitted through the
23 # connection
24 proxy.on('close', function (res, socket, head) {
25     send = socket.bytesRead;
26     received = socket.bytesWritten;
27 });
```

Listing 5.2: Setting up a proxy that simulates latency with node-http-proxy

5.3 Measuring Space

6 Use-Cases

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 do implement four use-cases that are ordered from the least complex to the most complex.

6.1 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 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 to one hundred percent, 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 represents 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 equijoin with the primary key generated in such a way that 50 % of the entries in each table will match the join criteria. As result, we will reveal the entire outcome of the join. So 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.

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

Listing 6.1: 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 the 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 boolean values that is generated at random. 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.

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

Listing 6.2: Functional equivalent SQL statement for our second use-case

Use-Case Three Besides joins, another very important group of SQL operations are aggregate functions. Over a third of all privacy-sensitive analytics queries requires a aggregation [JNS17]. Therefore our third use-case is centred around an aggregate function, or more precisely a maximum operator. TODO

Use-Case Four For our fourth and last use-case, we compare two special features of SMCQL and Conclave. Both Conclave and SMCQL feature a mechanic, that allows revealing some of the columns of the input data. The revelation of the input data allows them to apply optimizations that speed up computation while preserving the privacy of the other columns. For a more detailed description see SMCQL's access control and Conclave's trust annotations. Therefore in our fourth 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 both Conclave and SMCQL to apply their optimization. Replicating the setup of the first use-case enables us, to compare the results of the fourth use-case to the first use-case. This comparison will show how big the speed up of these optimizations is in practice.

6.2 Implementation

In this section we provide a description of notable details of the implementation of our use-cases.

Use Case One TODO

```

0  def protocol():
1  # define the schema for the input tables
2  columns_in_party1 = [
3  defCol("primary_key", "INTEGER", [1]),
4  defCol("user_data1_Alice", "INTEGER", [1]),
5  defCol("user_data2_Alice", "INTEGER", [1]),
6  defCol("user_data3_Alice", "INTEGER", [1]),
7  ]
8  # the content of the tables is loaded from a pregenerated .csv
9  input_1 = create("input_1", columns_in_party1, {1})
10
11 columns_in_party2 = [
12 defCol("primary_key", "INTEGER", [2]),
13 defCol("user_data1_Bob", "INTEGER", [2]),
14 defCol("user_data2_Bob", "INTEGER", [2]),
15 defCol("user_data3_Bob", "INTEGER", [2]),
16 ]
17 input_2 = create("input_2", columns_in_party2, {2})
18 # calculate the join over the two tables
19 join_result = join(input_1, input_2, 'join_result', ['primary_key'], ['primary_key'])
20 # reveal the output of the join to Alice
21 collect(join_result, 1)
22 # reveal the output of the join to Bob
23 collect(join_result, 2)
24
25 if __name__ == "__main__":
26 with open(sys.argv[1], "r") as config:
27 # load the configuration data
28 config = json.load(config)
29 # tell Conclave to generate secure code for the protocol and execute it
30 workflow.run(protocol, c, mpc_framework="jiff", apply_optimisations=True)

```

Listing 6.3: The Python protocol of Conclave for our first use-case

```

0  std::vector<ColumnInfo> AliceCols = { ColumnInfo{ "key", TypeID::IntID,
1  keyBitCount } };
2  std::vector<ColumnInfo> BobCols = { ColumnInfo{ "key", TypeID::IntID,
3  keyBitCount } };
4
5  for (u32 i = 1; i < cols; ++i)
6  {
7      AliceCols.emplace_back("Alice" + std::to_string(i), TypeID::IntID, 32);
8      BobCols.emplace_back("Bob" + std::to_string(i), TypeID::IntID, 32);
9  }
10 # Create tables for Alice and Bob and fill them with content
11 Table AliceTable(rows, AliceCols);
12 Table BobTable(rows, BobCols);
13 # Fill the primary columns
14 for (u64 i = 0; i < rows; ++i)
15 {
16     # if out is false then the entry will be included in the join

```

```

16     auto out = (i >= intersectionsize);
17     for (u64 j = 0; j < 2; ++j)
18     {
19         AliceTable.mColumns[0].mData(i, j) = i + 1;
20         BobTable.mColumns[0].mData(i, j) = i + 1 + (rows * out);
21     }
22 }
23 # Fill the other columns with random integers
24 for (u64 i = 1; i < cols; ++i){
25     for (u64 j = 0; j < rows; ++j){
26         AliceTable.mColumns[i].mData(j, 0) = rand();
27         BobTable.mColumns[i].mData(j, 0) = rand();
28     }
29 }
30 # instantiate Timer for benchmarking
31 Timer t;
32 # Alice and Bob each run their own thread
33 auto routine = [&](int i) { setThreadName("t0");
34     t.setTimePoint("start");
35     #
36     auto A = (i == 0) ? srvs[i].localInput(AliceTable) : srvs[i].
remoteInput(0);
37     auto B = (i == 0) ? srvs[i].localInput(BobTable) : srvs[i].remoteInput
(0);
38
39     if (i == 0) t.setTimePoint("inputs");
40     if (i == 0) srvs[i].setTimer(t);
41     std::vector<SharedTable::ColRef> First_Select_collumns;
42     for (u64 i = 0; i < cols; ++i){
43         First_Select_collumns.emplace_back(SharedTable::ColRef(B,B.mColumns[i
44         ]));
45     }
46     for (u64 i = 1; i < cols; ++i){
47         First_Select_collumns.emplace_back(SharedTable::ColRef(A,A.mColumns[i
48         ]));
49     }
50     auto result =srvs[i].join( SharedTable::ColRef(A,A.mColumns[0]) ,
SharedTable::ColRef(B,B.mColumns[0]) , First_Select_collumns);
51     if (i == 0) t.setTimePoint("intersect");
52     for (u64 index = 0; index < result.mColumns.size(); ++index)
53     {
54         aby3::i64Matrix reveal(result.mColumns[index].rows(), result.
mColumns[index].i64Cols());
55         server[i].mEnc.revealAll(server[i].mRt.mComm, T.mColumns[index],
reveal);
56         if (i == 0) std::cout << reveal << std::endl;
57     }
58 };
59
60 auto t0 = std::thread(routine, 0);
61 auto t1 = std::thread(routine, 1);
62 t0.join();
63 t1.join();

```

62 }

Listing 6.4: Simplified Protocol for our first use-case in ABY3

Use-Case Two For our second use-case we do not implement the query directly. Instead we implement another 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 listened in Listing 6.5. In a first step Alice and Bob filter their input once so that it contains only entries with the value false in its boolean column. Afterwards 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 the 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 direct input of a party, or the output of unary operator, that has only one table as input. Additionally our indirect implementation has a significant better performance compared to the direct implementation, for a detail analysis see Evaluation.

```

0 SELECT PrimaryKey, AliceBool, BobBool FROM
  (SELECT PrimaryKey, AliceBool FROM AliceTable WHERE AliceBool == false)
2 NATURAL JOIN
  (SELECT PrimaryKey, BobBool FROM AliceTable WHERE BobBool == false)
4 UNION
  SELECT PrimaryKey, AliceBool, BobBool FROM
6  (SELECT PrimaryKey, AliceBool FROM AliceTable WHERE AliceBool == true)
  NATURAL JOIN
8  (SELECT PrimaryKey, BobBool FROM BobTable WHERE AliceBool == true)

```

Listing 6.5: Functional equivalent SQL statement for our implementation of use-case two

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, feature 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 indent correct.

7 Evaluation

Bibliography

- [ACC⁺21] Abdelrahman 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>.
- [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.
- [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.
- [Cos21] Cosmian. Ciphercompute. <https://github.com/Cosmian/CipherCompute>, 2021.
- [cPr21] The python profilers. <https://docs.python.org/3/library/profile.html>, 2021.
- [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>.
- [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.
- [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.
- [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.
- [Ran21] Peter Randal. cryptotools. <https://github.com/ladnir/cryptoTools>, 2021.
- [Rin] Peter Rindal. The ABY3 Framework for Machine Learning and Database Operations. <https://github.com/ladnir/aby3>.
- [Sho22] Shopify. toxiproxy. <https://github.com/Shopify/toxiproxy>, 2022.
- [tim21a] time. <https://manpages.debian.org/stretch/time/time.1.en.html>, 2021.
- [tim21b] timeit. <https://docs.python.org/3/library/timeit.html>, 2021.
- [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.
- [wir21] wireshark. <https://www.wireshark.org/>, 2021.
- [WMK16] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. EMP-toolkit: Efficient MultiParty computation toolkit. <https://github.com/emp-toolkit>, 2016.