Scalable Oblivious Data Analytics

D4.2 Proof-of-concept for results of tasks 1.2 and 2.2: Implementations of key MPC technologies

Paul Koster (PHI), Peter Norholt (ALX), Tore Frederiksen (ALX), Jonas Lindstrøm (ALX)

The project SODA has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 731583.

Project Information

Scalable Oblivious Data Analytics

Project number:	731583
Strategic objective:	H2020-ICT-2016-1
Starting date:	2017-01-01
Ending data:	2019-12-31
Website:	https://soda-project.eu/

Document Information

Title:	Proof-of-concept for results of tasks 1.2 and 2.2: implementations of key MPC technologies				
ID:	D4.2	Type:	OTHER	Dissemination level:	PU
		Month:	M24	Release date:	18.12.2018

Contributors, Editor & Reviewer Information

Contributors (person(partner): sections)	Paul Koster (PHI): Chapters 4, 5, 6, 7 and 8 Peter Norholt (ALX): Chapter 3 Tore Frederiksen (ALX): Chapters 3 and 8 Jonas Lindstrøm (ALX): Chapters 2 and 8
Editor (person/partner)	Jonas Lindstrøm (ALX)
Reviewer (person/partner)	Peter van Liesdonk (PHI), Niek Bouman (TUE)

Release History

Release num-ber	**Date issued**	**Release description / changes made**
1.0	18-12-2018	Initial release

SODA Consortium

Full Name	Abbreviated Name	Country
Philips Electronics Nederland B.V.	PHI	Netherlands
Alexandra Institute	ALX	Denmark
Aarhus University	AU	Denmark
Göttingen University	GU	Germany
Eindhoven University of Technology	TUE	Netherlands

Table 1: Consortium Members

Executive Summary

This report summarizes the deliverable D4.2 proof-of-concept implementations developed in the SODA project, particularly related to tasks 1.2 and 2.2. These proof-of-concepts demonstrate:

- Efforts in making MPC more accessible to application developers through easy-to-use development frameworks FRESCO and MPyC,
- Inclusion of novel MPC advancements in the FRESCO framework,
- Work on practical verifiable computations using zk-SNARKs,
- Implementations of data analytics and machine-learning algorithms using MPC, namely DNA string search and logistic regression.

About this Document

Role of the deliverable

This deliverable consists of proof-of-concept implementations developed in task T4.1 in the SODA project so far. The implementations are based on the results of tasks T1.2. and T2.2. The proof-of-concept deliverables are available as open-source solutions online, and this report gives a brief description of each prototype and links to where the source code can be obtained.

Relationship to other SODA deliverables

This deliverable describes the proof-of-concept implementations created as part of task T4.1 which are based on tasks 1.2 and 2.2 (general purpose protocols and algorithms). Deliverable 4.4 will consist of proof-of-concept implementations created as part of task T4.1 based on tasks 1.3 and 2.3 (special purpose protocols and algorithms), and some of these might be extensions of the proof-of-concepts presented in the present deliverable. Some of the proof-of-concept implementations provide a basis for the demonstrators of D4.5 in task T4.2.

Relationship to other versions of this deliverable

N/A

Structure of this document

Chapter 3 describes the improvements and new functionality added to the FRESCO framework, and Chapter 4 describes the MPyC framework. Chapter 5 and 6 describe the Geppetri and PySNARK prototypes for verifying the correctness of computations. Chapter 7 describes a prototype for DNA string search prototype. In chapter 8 a prototype for performing logistic regression on secret shared data is presented as well as a Kotlin wrapper for the FRESCO framework allowing operator overloading.

Acknowledgements

Besides the authors this work reflects implementation efforts from: Berry Schoenmakers (TUE), Mark Spanbroek (PHI), Meilof Veeningen (PHI), Sakina Asadova (PHI), Stefan van den Oord (PHI), Nikolaj Volgushev (ALX) and others.

1 Table of Contents

Release History 5
SODA Consortium 6
Executive Summary 7
About this Document 8
Role of the deliverable 8
Relationship to other SODA deliverables 8
Relationship to other versions of this deliverable 8
Structure of this document 8
Acknowledgements 8
1 Table of Contents 9
2 Introduction 11
3 FRESCO 12
3.1 Introduction 12
3.2 General Improvements 12
3.3 SPDZ2k 13
3.4 Resources 14
4 MPyC 15
4.1 Introduction 15
4.2 Functionality and features 15
4.3 Applications 15
4.4 Resources 15
5 Geppetri 17
5.1 Introduction 17
5.2 Functionality and features 17
5.3 Application 17
5.4 Resources 17
6 PySNARK 18
6.1 Introduction 18
6.2 Functionality and features 18
6.3 Applications 18
6.4 Resources 19
7 DNA Inexact String Search 20
8 Developer-friendly MPC: operator overloading and co-routines 21
8.1 Introduction 21
8.2 Kotlin 21
8.3 Application 22

8.4 Resources 22
9 Bibliography 23

2 Introduction

This deliverable describes the status of six proof-of-concept implementations developed in the SODA project as part of task T4.1. The main part of the deliverable are the actual proof-of-concept implementations, all of which are open-source with source code available online. This report gives a brief overview of the features and status of the different implementations.

Due to the complexity of MPC, implementing fast and secure MPC-based applications can be a difficult task for application developers. To make the development of such applications easier, two development frameworks have been co-developed and matured as part of SODA. In Chapter 3 we present improvements to the FRESCO framework, and what new functionality and support for new cryptographic protocols has been added. The MPyC Python framework has been released in 2018, and we present some of the features in Chapter 4. We have also experimented with a Kotlin-based wrapper around the FRESCO framework to allow operator overloading, which could make complex calculations easier to write. This is presented in Chapter 8.

Some work on verifiable computations is presented in chapter 5 and 6. In Chapter 5 we present Geppetri which prototypes the concepts of zk-SNARK and adaptive Trinocchio, and in Chapter 6 PySNARK, which is a Python-based system for zk-SNARK based verifiable computations and smart contracts, is presented.

Two prototypes for data analytics are presented in chapter 7 and 8. In context of DNA string search, an implementation was made in TUeVIFF (predecessor of MPyC) and related secret indexing for verifiable computation in PySNARK. This is presented in Chapter 7. A privacy preserving logistical regression algorithm was implemented using the FRESCO framework using the Kotlin wrapper and is presented in chapter 8.

Partners shared the code of the proof-of-concept implementations with the project and the general public, e.g. on GitHub under open source or research licenses. The location is provided for each of the prototypes at the end of their corresponding chapters.

3 FRESCO

3.1 Introduction

FRESCO is a Java framework for efficient secure computation aiming at making the development of prototype applications based on secure computation easy.

As part of this work package we have made substantial improvements and expansions to the FRESCO framework. Both as part of a paper currently in submission [1], but also as part of other projects along with overall improvements to enhance stability, usability and security.

3.2 General Improvements

During the SODA project FRESCO has moved towards a more organized approach to the development and maintenance of the framework. This includes an increased focus on code quality by requiring full test coverage, and doing code review of all newly added code. Furthermore, we have moved to a more frequent release model, aiming at releases every 2-3 months and resulting in releases 1.0.0, 1.1.0, 1.1.1 and 1.1.2 over the SODA period so far. These releases include a major refactoring and restructuring of the framework code that, among other things, split the framework into a number of sub-modules making it simpler to maintain, extend and use the framework. Additionally, a number of new features has been added in these releases:

- **New Application Interface**: A lot of work was done in order to improve the API towards application programmers. I.e., the API to with which to specify MPC operations to be evaluated as part of a larger application. Here we introduced a new builder pattern-based approach making it easier to access the various library MPC functionality included in FRESCO, and to build efficient MPC programs.

- **SPDZ Preprocessing**: An implementation of the preprocessing phase for the SPDZ protocol was added, meaning that FRESCO now supports the full SPDZ protocol. The implementation added to FRESCO is based on the MASCOT protocol [2]. The MASCOT protocol in turn relies on Oblivious Transfer Extension which was also included in FRESCO based on the protocol of Keller *et al* [3].

- **Network Improvements**: The networking functionality has been simplified and new networking implementations were added to replace an old less stable implementation. Also, a networking implementation has been added to make it easier to support secure networking using TLS and examples of how to add TLS based communication was added.

- **New Protocols**: Two new MPC protocol implementations were added to the framework. A basic version of the TinyTables protocol with semi-honest security (including its preprocessing phase), and the SPDZ2k protocol as described below.

- **New Library functionality**: A number of new generic MPC functionalities were implemented as library functionality in FRESCO. Among these are:

    - A library for arithmetic with fixed point numbers including various linear algebra operations on matrices which will be useful for the ongoing work in SODA with MPC based machine learning.

    - Aggregation functionality based on MiMC [4] encryption in MPC, which also required new shuffling functionality to be added to the library.

- New implementations of comparison protocols based on the methods of Catrina and de Hoogh [4] to complement the earlier implementation based on the protocols of Lipmaa and Toft [5].

Apart from these additions to the main FRESCO repository a number of more experimental projects based on FRESCO has been started as separate projects that are currently not (yet) included in the main FRESCO repository. These include:

- **Tools for outsourced MPC**: Outsourced MPC refers to a setting where MPC is performed among a few servers that securely take their inputs and deliver output to a much larger set of clients. Thus, outsourced MPC is a simple way for MPC-based applications to scale well in the number of parties. For this reason, we started work on a set of reusable tools for outsourced MPC applications. Specifically, our work is focused on protocols for delivering output and receiving input to the MPC server to and from the clients. Initially, we have implemented the protocol of Damgård *et al.* (we note that while this has earlier been implemented in FRESCO, the implementation was not suitable for reuse), and plan to also implement the approach of Jakobsen *et al.* and possibly newly developed protocols resulting from the SODA project. The repository for this work is available publicly via GitHub under the MIT license.

- **Kotlin wrapper for FRESCO:** Inspired by some work done by PHI as part of the prototype on logistic regression, a wrapper for FRESCO allowing a developer to write applications using FRESCO in Kotlin was developed. This is explained in more detail in section 8.

- **MPC based Machine Learning**: As part of the SODA project we are doing a number of experiments with MPC based Machine Learning. This work includes the implementation of SVM and Decision Tree algorithms in FRESCO to showcase the SPDZ2k protocol as described below. Additional we have done work a basic implementation of Neural Networks as well. These experiments are gathered in a publicly available repository on GitHub under the MIT license.

Apart from the code added to the various FRESCO repositories, the documentation for FRESCO has also been updated to reflect the various changes made. Furthermore, we have opened a public chat channel dedicated to FRESCO, which has proven quite valuable in providing support to new users.

3.3 SPDZ2k

The SPDZ2k paper [2] completed as part of the SODA project in T1.2, i.e. D1.2 section 2.5, and published at CRYPTO 2018 showed how to use the SPDZ [3] paradigm for computation over the ring $\mathbb{Z}_{2^k}$, rather than the field $\mathbb{F}_p$. One major motivation of that work is that it facilitates integer arithmetic over $\mathbb{Z}_{2^{32}}$ and $\mathbb{Z}_{2^{64}}$ as most CPUs do. For this reason, it is expected to allow for a more efficient implementation of the online phase. Furthermore, when computing over the same domain as CPUs, we expect it to be possible to leverage operations and algorithms in this space, that might not work over $\mathbb{F}_p$.

To test this conjecture, SODA authors worked on writing protocols for basic operations over $\mathbb{Z}_{2^k}$ such as equality testing, truncation, and comparison [1]. To give credence to the conjecture, we implemented SPDZ2k, along with the new protocols for basic operations, in FRESCO. Besides being essential for the paper, this expansion of FRESCO furthermore enhances the general usefulness of the framework, by now allowing computation over standard 32- and 64-bit integers. Thus, opening up the option of implementing algorithms requiring such domains.

To show that this expansion also yielded more efficient online computation times in a sensible setting, we chose to implement oblivious decision tree evaluation and support vector machine evaluation. Our tests showed the online execution time of these to be between 2x and 5x times faster than SPDZ, in the batched setting. We note that implementing exactly these algorithms is of general interest to the SODA project, because they consider privacy preserving computation on big data. Specifically, the case of a provider holding a large amount of data used to develop a model which it wishes to keep private, while still allowing consumers to get input data classified using this model. Thus, yielding a privacy-preserving machine-learning-as-a-service.

More specifically we added the following implementations to FRESCO:

- Online support for SPDZ2k with an optimized, custom class handling arithmetic over 32- and 64-bit domains.
- A computationally efficient constant round equality testing protocol for SPDZ2k.
- A computationally efficient constant round comparison protocol for SPDZ2k.
- An efficient truncation protocol for SPDZ2k.

We expanded the machine learning library for FRESCO, called FRESCO-ML, with the following:

- An implementation of two-party oblivious decision tree evaluation in $O(\log(depth))$ rounds.
- An implementation of two-party oblivious SVM evaluation in $O(\log(classes))$ rounds.

The additions to FRESCO will be available in release v. 1.1.3 and the additions to FRESCO-ML will be available in the master branch.

3.4 Resources

FRESCO is available at GitHub at https://github.com/aicis/fresco, and the documentation can be found at Read the Docs at https://fresco.readthedocs.io/en/latest/.

The public channel where we receive questions, bug reports and suggestions on how to improve FRESCO can be found at https://gitter.im/FRESCO-MPC/Lobby.

The Kotlin-wrapper prototype can be found at https://github.com/aicis/fresco-kotlin-applications.

The prototype on tools for outsourced using FRESCO can be found on GitHub at https://github.com/aicis/fresco-outsourcing.

The various prototypes on machine learning with FRESCO can be found on GitHub at https://github.com/aicis/fresco-ml.

4 MPyC

4.1 Introduction

MPyC is a Python framework for secure multiparty computation. It was released 30 May 2018 at TPMPC 2018. MPyC succeeds VIFF and TUeVIFF. MPyC targets usability and ease of use balanced with efficiency.

Above properties make it well-suited for education and rapid prototyping of MPC applications, which is demonstrated by its use in the academic curriculum, master projects and prototyping activities in and outside SODA.

4.2 Functionality and features

In MPyC the details of the secure computation protocols are mostly transparent due to the use of sophisticated operator overloading combined with asynchronous evaluation of the associated protocols. The latter is realized through the use of so-called co-routines, which results in a natural control flow.

MPyC supports secure m-party computation tolerating a dishonest minority of up to t passively corrupt parties, where $m \geq 1$ and $0 \leq t \leq (m-1)/2$. The underlying protocols are based on threshold secret sharing over finite fields (using Shamir's threshold scheme as well as pseudorandom secret sharing).

MPyC features a variety of protocols, e.g. for integers, fixed point numbers, linear programming. It also supports easy input / output methods and hides (pseudorandom) secret sharing.

Communication is transparent. There is largely no difference between developing on a single host or a distributed set of hosts. It supports secure communication through TLS.

The ability for 1-party computation enables developers to very easily prototype an MPC application. Subsequently, very little changes are required to turn it into a secure multi-party computation.

4.3 Applications

MPyC comes with a number of sample applications, e.g. ID3 decision trees, sorting networks (depicted in Figure 1).

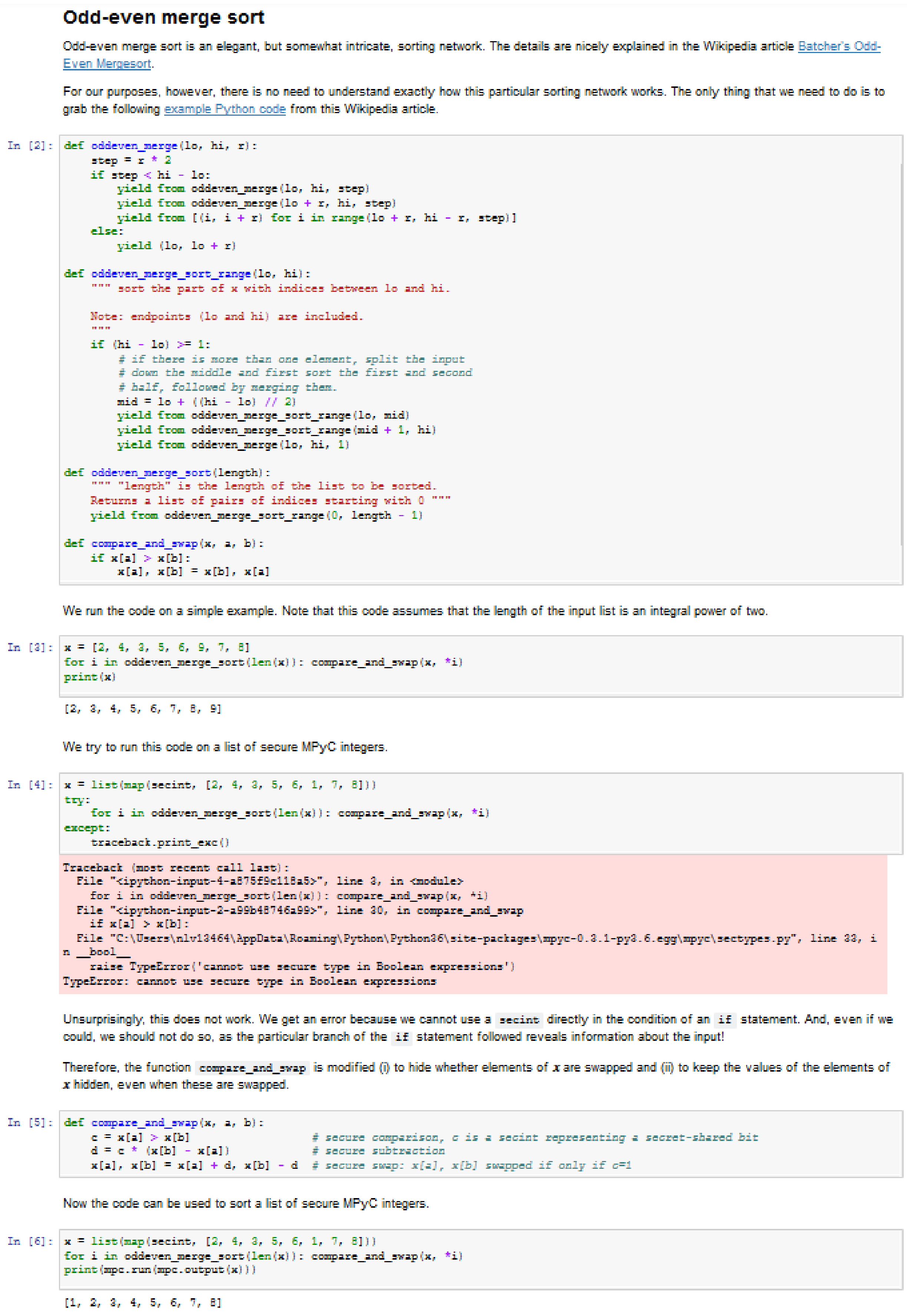
Also SODA results from D1.2 and D2.2 have been prototyped using MPyC. This includes e.g. fast secure comparisons for medium sized integers (D2.2 chapter 2), MPC implementation of BWT algorithms for inexact DNA string search (D2.2 chapter 3), and secure convolutional networks (D2.2 chapter 4).

4.4 Resources

MPyC is available on GitHub at https://github.com/lschoe/mpyc. It also includes documentation.

MPyC is also available as a package in the Python Package Index (PyPI).

An introduction to MPyC is available at http://conferences.au.dk/fileadmin/user_upload/TPMPC18/Berry_Schoenmakers.pdf, which is also available as a video including demo at http://vc.au.dk/videos/video/6826/ (from 51:37).



Odd-even merge sort

Odd-even merge sort is an elegant, but somewhat intricate, sorting network. The details are nicely explained in the Wikipedia article Batcher's Odd-Even Mergesort.

For our purposes, however, there is no need to understand exactly how this particular sorting network works. The only thing that we need to do is to grab the following example Python code from this Wikipedia article.

In [2]:
```
def oddeven_merge(lo, hi, r):
    step = r * 2
    if step < hi - lo:
        yield from oddeven_merge(lo, hi, step)
        yield from oddeven_merge(lo + r, hi, step)
        yield from [(i, i + r) for i in range(lo + r, hi - r, step)]
    else:
        yield (lo, lo + r)

def oddeven_merge_sort_range(lo, hi):
    """ sort the part of x with indices between lo and hi.

    Note: endpoints (lo and hi) are included.
    """
    if (hi - lo) >= 1:
        # if there is more than one element, split the input
        # down the middle and first sort the first and second
        # half, followed by merging them.
        mid = lo + ((hi - lo) // 2)
        yield from oddeven_merge_sort_range(lo, mid)
        yield from oddeven_merge_sort_range(mid + 1, hi)
        yield from oddeven_merge(lo, hi, 1)

def oddeven_merge_sort(length):
    """ "length" is the length of the list to be sorted.
    Returns a list of pairs of indices starting with 0 """
    yield from oddeven_merge_sort_range(0, length - 1)

def compare_and_swap(x, a, b):
    if x[a] > x[b]:
        x[a], x[b] = x[b], x[a]
```

We run the code on a simple example. Note that this code assumes that the length of the input list is an integral power of two.

In [3]:
```
x = [2, 4, 3, 5, 6, 9, 7, 8]
for i in oddeven_merge_sort(len(x)): compare_and_swap(x, *i)
print(x)
```

```
[2, 3, 4, 5, 6, 7, 8, 9]
```

We try to run this code on a list of secure MPyC integers.

In [4]:
```
x = list(map(secint, [2, 4, 3, 5, 6, 1, 7, 8]))
try:
    for i in oddeven_merge_sort(len(x)): compare_and_swap(x, *i)
except:
    traceback.print_exc()
```

```
Traceback (most recent call last):
  File "<ipython-input-4-a875f9c118a5>", line 3, in <module>
    for i in oddeven_merge_sort(len(x)): compare_and_swap(x, *i)
  File "<ipython-input-2-a99b48746a99>", line 30, in compare_and_swap
    if x[a] > x[b]:
  File "C:\Users\nlv13464\AppData\Roaming\Python\Python36\site-packages\mpyc-0.3.1-py3.6.egg\mpyc\sectypes.py", line 33, i
n __bool__
    raise TypeError('cannot use secure type in Boolean expressions')
TypeError: cannot use secure type in Boolean expressions
```

Unsurprisingly, this does not work. We get an error because we cannot use a `secint` directly in the condition of an `if` statement. And, even if we could, we should not do so, as the particular branch of the `if` statement followed reveals information about the input!

Therefore, the function `compare_and_swap` is modified (i) to hide whether elements of *x* are swapped and (ii) to keep the values of the elements of *x* hidden, even when these are swapped.

In [5]:
```
def compare_and_swap(x, a, b):
    c = x[a] > x[b]                  # secure comparison, c is a secint representing a secret-shared bit
    d = c * (x[b] - x[a])            # secure subtraction
    x[a], x[b] = x[a] + d, x[b] - d  # secure swap: x[a], x[b] swapped if only if c=1
```

Now the code can be used to sort a list of secure MPyC integers.

In [6]:
```
x = list(map(secint, [2, 4, 3, 5, 6, 1, 7, 8]))
for i in oddeven_merge_sort(len(x)): compare_and_swap(x, *i)
print(mpc.run(mpc.output(x)))
```

```
[1, 2, 3, 4, 5, 6, 7, 8]
```

Figure 1: secure sorting networks MPyC example application

5 Geppetri

5.1 Introduction

Geppetri prototypes the concepts presented SODA deliverable D1.2 chapter 8 on Pinocchio-Based Adaptive zk-SNARKs and Secure and Correct Adaptive Function Evaluation.

That chapter and associated paper [4] show how to practically verify the correctness of the results of computations that have been performed in a privacy-preserving manner. To do this it gives a new construction of adaptive zk-SNARKs based on Pinocchio, which allows a prover to repeatedly prove statements about some committed input, whilst hiding the input in a zero-knowledge manner. It then shows how this can be applied to the Trinocchio system, to obtain a verifiable multi-party computation protocol, where private computations can be repeatedly performed and verified, such that the sensitive data is even hidden from the prover by multi-party computation.

Geppetri is built on top of a custom VIFF implementation.

5.2 Functionality and features

In Geppetri computations can be specified at a high level using a Python frontend. These can be executed either locally or in a privacy-preserving way using multi-party computation and then automatically proven and verified to be correct by a C++ backend.

For further details on features and implementation we refer to section 6.1 of [4] and the software.

5.3 Application

The Geppetri tooling has been applied to build a prototype in a medical research case study, i.e. logrank statistical test. For this and other simple examples see the README.

5.4 Resources

Geppetri code is available on GitHub at https://github.com/meilof/geppetri.

6 PySNARK

6.1 Introduction

PySNARK is a Python-based system for zk-SNARK (zero-knowledge succinct non-interactive argument of knowledge) based verifiable computations and smart contracts. It is based on the Pinocchio system and the Geppetri extension for proofs on authenticated data. For Geppetri see chapter 8 of SODA deliverable D1.2. PySNARK succeeds in part the Geppetri prototype presented in the previous section.

6.2 Functionality and features

PySNARK makes it possible to program verifiable computations in Python. This solves the problem that it is quite hard to specify verifiable computations. PySNARK automatically creates a proof.

To verify proofs, PySNARK can automatically generate smart contracts. Particularly, verifiable computations can be turned into Solidity smart contracts for use on the Ethereum blockchain.

The PySNARK toolchain supports each of the different types of parties in a verifiable computation: trusted party, prover, or verifier.

Values used in a verifiable computation are called variables of the Var class, which can represent inputs/outputs, witnesses, or committed data. Python operator overloading for addition, subtraction and multiplication is used to work with these variables in a convenient manner.

PySNARK also includes a library with efficient verifiable computations. Examples include fixed-point arithmetic, GGH hash function, secret indexing. The privacy-preserving verifiable fixed-point division is presented in SODA D1.2 chapter 11. SODA D1.2 chapter 10 on data-oblivious array slicing with sliding windows is a further candidate for PySNARK.

6.3 Applications

PySNARK has been put into practice with a number of applications. To illustrate its use consider the examples folder on the GitHub repository.

PySNARK is also used to prototype SODA verifiable computation work. SODA deliverable D1.2 section 8.4 applies verifiable computation on a statistical test for medical survival analysis known as Kaplan-Meier.

Similarly PySNARK was used for oblivious verification of inexact string matches (SODA D1.2, chapter 9), which builds on the MPC-implementation of the BWT algorithm to search in DNA. A possibility to authenticate the search result in a larger string through a Merkle hash tree has been added too.

```
$ python cube.py 3
The cube of 3 is 27
   main main 8d72a44bde04cf2 5 *
qaplen: 5 blklen: 0 extlen: None sz 8 pubsz 0
*** Generating master key material
Generating new mastersk and keys, sizes 8/0
done
New signature for function main, rebuilding keys
Generating functon key material using master secret key
Reading QAP...
Generating keys...
Using QAP degree=8
Writing to pysnark_ek_main
Writing to pysnark_vk_main
Reading QAP from (pysnark_eqs_main,pysnark_ek_main)
Proving main (pysnark_ek_main)
Compute h 1
```

```
Reading QAP vk: pysnark_vk_main
Verifying main (pysnark_vk_main) 1
Verification succeeded
  prover keys/eqs:  pysnark_masterek pysnark_ek_main pysnark_eqs_main pysnark_schedule
  prover data:
  verifier keys:    pysnark_masterpk pysnark_vk_main pysnark_schedule
  verifier data:     pysnark_proof pysnark_values
  verifier cmd:     pysnark/qaptools/qapver pysnark_masterpk pysnark_schedule
pysnark_proof pysnark_values
```

Figure 2: example PySNARK run for cube 3

6.4 Resources

The PySNARK software and source code is available on GitHub via http://pysnark.tk which forwards to https://github.com/Charterhouse/pysnark.

Above repository also hosts documentation and write-up available through https://github.com/Charterhouse/pysnark/blob/master/docs/PySNARK.pdf.

A one-minute introduction talk of PySNARK at RealWorldCrypto 2018 is available at https://youtu.be/o4U0Gfh-0L4?t=441.

7 DNA Inexact String Search

The DNA inexact string search work presented in SODA deliverable D2.2 chapter 3 was prototyped. This provides an MPC-based implementation of the BWT algorithm.

The implementation is made in TUeVIFF.

```
> python simple_bwt.py player1_1.ini ACGT A 1
2018-10-24 15:30:53+0200 [-] Log opened.
2018-10-24 15:30:53+0200 [-] indexed search string [1]
2018-10-24 15:30:53+0200 [-] mismatch limit 1
2018-10-24 15:30:53+0200 [-] Sorting [['C', 'G', 'T', '$', 'A', 1], ['G', 'T', '$',
'A', 'C', 2], ['T', '$', 'A', 'C', 'G', 3], ['$', 'A', 'C', 'G', 'T', 4], ['A', 'C',
'G', 'T', '$', 0]]
2018-10-24 15:30:53+0200 [-] Sorting [['C', 'G', 'T', '$', 'A', 1], ['G', 'T', '$',
'A', 'C', 2], ['T', '$', 'A', 'C', 'G', 3], ['$', 'A', 'C', 'G', 'T', 4], ['A', 'C',
'G', 'T', '$', 0]]
2018-10-24 15:30:53+0200 [-] transform= ['T', '$', 'A', 'C', 'G'] suffix array= [4, 0,
1, 2, 3]
2018-10-24 15:30:53+0200 [-] BWT: 0.001512765884399414
2018-10-24 15:30:53+0200 [-] [('A', 0, [0, 0, 1, 1, 1]), ('C', 1, [0, 0, 0, 1, 1]),
('G', 2, [0, 0, 0, 0, 1]), ('T', 3, [1, 1, 1, 1, 1]), ('$', 0, [0, 1, 1, 1, 1])]
2018-10-24 15:30:54+0200 [-] inexact search: 1.1039559841156006
2018-10-24 15:30:54+0200 [-] Initializing VC (data dir=data/)
2018-10-24 15:30:54+0200 [-] general search result:  res [{1}, {1}, {1}, {1}, {1}]
found {1} pos {4} mymap [{4}, {2}, {4}]
2018-10-24 15:30:54+0200 [-] map printed: [{4}, {2}, {4}]
2018-10-24 15:30:54+0200 [-] inverse: ['A', 'C', 'G']
```

Figure 3: prototype run on 1-party small example

A related prototype concerns DNA proofs (see SODA deliverable D1.2 chapter 9), which is implemented using PySNARK. Here, the code has to search for a substring in a DNA string, up to a maximal edit distance, and to prove using verifiable computation that such a match was indeed found.

The prototype first reports the location on which a match was found, and then it prints the hashes for hash tree authentication, finishing with the hash of the root node of this particular hash tree.

For efficiency, the prototype also includes the method presented in SODA deliverable D1.2 chapter 10 on a data-oblivious technique for array slicing. This is used for proving, in a verifiable computation, that a DNA string contains a particular substring as presented in the previous section. One of the steps in this computation is to retrieve a substring from the DNA string in order to prove that it matches the requested substring, e.g., with some spelling mistakes. Here, the DNA string is typically very long, so being able to do this with just one linear scan through the DNA string is highly desirable.

8 Developer-friendly MPC: operator overloading and co-routines

8.1 Introduction

Developer-friendly MPC is critical to support adoption of MPC technology. SODA partners experimented with two programming concepts: operator overloading and co-routines.

Operator overloading enables developers to use regular notations for operators, e.g. arithmetic, which abstract from MPC complexities when operating on secret shared data types.

Co-routines similarly enable developers to preserve a (relatively) simple control flow in MPC (distributed) computation. Co-routine language constructs enable hiding of many of the complexities caused by MPC deferred evaluation. It particularly avoids many of the disadvantages of dealing with callbacks as a developer.

Ideation and proof of concept implementation on this topic is an example of cross-fertilization between SODA partners Philips, Alexandra and TUE. PHI suggested co-routine inclusion in TUeVIFF (now MPyC, see section 4). PHI also experimented with operator overloading and co-routines through a Kotlin language wrapper around the FRESCO APIs and a logistic regression implementation. ALX adapted and improved the wrapper.

8.2 Kotlin

Kotlin is a statically typed programming language whose code is compiled to Java bytecode, allowing it run on top of the JVM. Unlike Java, Kotlin supports operator overloading, meaning the arithmetic operators such as -, +, * and / and be overridden to reflect custom code within a certain scope.

Overriding these operators with their corresponding MPC implementations within a build scope in FRESCO allows for much more direct, faster and easier to read implementations of MPC applications. For this reason, PHI worked on a Kotlin wrapper to work on top of FRESCO. This wrapper adds operator overloading for standard arithmetic operations, but also adds overloaded implementations for vector and matrix operations as well. For example, calculating the standard logistic function of a secret shared value in FRESCO is done using the builder pattern, and could be written like this:

```
builder.realNumeric().div(1.0, builder.realNumeric().add(1.0, builder.realAdvanced()
                     .exp(builder.realNumeric().negate(x))))
```

With operator overload in Kotlin and access to library functions such as the exponential function, the same expression can be written as:

```
1.0 / (1.0 + exp(-x))
```

Furthermore, a custom Kotlin implementation leveraging the readability of co-routines in this language, was made of FRESCO's MPC execution pipeline for the specific setting of privacy preserving logistic regression. However, this proof of concept has limitations, because it implies sequential execution and the entire MPC application to stay in memory.

ALX generalized the Kotlin wrapper to work universally with any MPC implementation using FRESCO. Concretely this was done by writing a Kotlin wrapper for overloading operations -, +, * and / along with <= and >. Furthermore -, + and * was also implemented as component-wise operations on lists. This generalized wrapper was made to work with FRESCO 1.0.0 for any type of MPC application, using FRESCO's pipelined execution. That is, based on scopes of sequential and parallel execution, where only a single scope is required to remain in memory.

8.3 Application

The operator overloading and co-routine concepts are further demonstrated in a proof of concept to privately construct a logistic regression model based on abovementioned Kotlin language wrapper for FRESCO.

The specific algorithm for privacy preserving logistic regression implemented is based on an approach by Shi *et al*. [5], optimized by using Cholesky decomposition as part of the process to solve linear systems [6]. Specifically, this can be used to construct classification models based private data, which is of particular importance in the medical setting.

8.4 Resources

The initial Kotlin wrapper and the logistic regression implementation is available at https://github.com/meilof/Fresco-Logistic-Regression.
A test suite is available at https://github.com/meilof/TestLogisticRegression.

A write up on the use of co-routines together with documentation of the pipeline implementation is available at https://github.com/meilof/TestLogisticRegression/blob/master/coroutines.pdf [7].

The adapted and improved Kotlin wrapper implementation is available at https://github.com/ai-cis/fresco-kotlin-applications. The adapted implementation of logistic regression is available at https://github.com/jonas-lj/Fresco-Logistic-Regression

9 Bibliography

[1] I. Damgård, D. Escudero, T. Frederiksen, P. Scholl and N. Volgushec, "New Primitives for Actively-Secure MPC mod 2k with Applications to Private Machine Learning," in *In submission*, 2018.

[2] M. Keller, E. Orsini and P. Scholl, "MASCOT: Faster Malicious Arithmetic Secure Computation with Oblivious Transfer," *ACM Conference on Computer and Communications Security,* pp. 830-842, 2016.

[3] M. Keller, E. Orsini and P. Scholl, "Actively Secure OT Extension with Optimal Overhead," *CRYPTO,* vol. 1, pp. 724-741, 2015.

[4] M. Albrecht, L. Grassi, C. Rechberger, A. Roy and T. Tiessen, "MiMC: Efficient encryption and cryptographic hashing with minimal multiplicative complexity.," *Advances in Cryptology – ASIACRYPT 2016. ASIACRYPT 2016. Lecture Notes in Computer Science,* vol. 10031, pp. 191-219, 2016.

[5] O. Catrina and S. de Hoogh, "Improved primitives for secure multiparty integer computation," *SCN'10 Proceedings of the 7th international conference on Security and cryptography for networks,* pp. 182-199, 2010.

[6] H. Lipmaa and T. Toft, " Secure Equality and Greater-Than Tests with Sublinear Online Complexity," *ICALP,* vol. 2, pp. 645-656, 2013.

[7] R. Cramer, I. Damgård, D. Escudero, P. Scholl and C. Xing, "SPDZ2k: Efficient MPC mod 2k for Dishonest Majority," in *CRYPTO,* Santa Barbara, California, 2018.

[8] I. Damgård, V. Pastro, N. Smart and S. Zakarias, "Multiparty Computation from Somewhat Homomorphic Encryption," in *CRYPTO*, Santa Barbara, California, 2012.

[9] M. Veeningen, "Pinocchio-Based Adaptive zk-SNARKs and Secure/Correct Adaptive Function Evaluation," in *Africacrypt 2017*, 2017.

[10] H. Shi, C. Jiang, W. Dai, Y. Tang, L. Ohno-Machado and S. Wang, "Secure Multi-pArty Computation Grid LOgistic REgression (SMAC-GLORE," in *Translational Bioinformatics Conference*, Tokyo, 2015.

[11] V. Nikolaenko, U. Weinsberg, S. Ioannidis, M. Joye, D. Boneh and N. Taft, "Privacy-Preserving Ridge Regression on Hundreds of Millions of Records," in *IEEE Symposium on Security and Privacy*, Berkeley, 2013.

[12] M. Veeningen, "Programming MPC applications with FRESCO using Kotlin and Coroutines," 24 January 2018. [Online]. Available: https://github.com/meilof/TestLogisticRegression/blob/master/coroutines.pdf.