D14.4
Validation Report

Project number:	609611
Project acronym:	**PRACTICE**
Project title:	Privacy-Preserving Computation in the Cloud
Project Start Date:	1 November, 2013
Duration:	36 months
Programme:	FP7/2007-2013

Deliverable Type:	Report
Reference Number:	ICT-609611 / D14.4 / 1.0
Activity and WP:	Activity 1 / WP14
Due Date:	October 2016 - M36
Actual Submission Date:	2nd November, 2016

Responsible Organisation:	ALX
Editor:	Peter Sebastian Nordholt
Dissemination Level:	Public
Revision:	1.0

Abstract:	We validate the work on the PRACTICE platform implementations done in WP14. We do this from the point of view of both protocol developers and developers of applications based on secure computation. We address the former by demonstrating how protocols can be implemented and integrated into the platform. The later we do by summarizing a number of real world scenarios in witch the frameworks of PRACTICE have been used to implement secure computation based solutions.
Keywords:	Protocol, Protocol Suite, Secure Computation, Sharemind, FRESCO, ABY

This project has received funding from the European Unions Seventh Framework Programme for research, technological development and demonstration under grant agreement no. 609611.

Editor

Peter Sebastian Nordholt (ALX)

Contributors (ordered according to beneficiary numbers)

Martin Deutschmann (TEC)
Mario Münzer (TEC)
Daniel Demmler (TUDA)
Ágnes Kiss (TUDA)
Thomas Schneider (TUDA)
Michael Zohner (TUDA)
Kasper Damgård (ALX)
Peter Sebastian Nordholt (ALX)
Michael Stausholm (ALX)
Manuel Barbosa (INESC PORTO)
Vitor Pereira (INESC PORTO)

Executive Summary

The main goal of WP14 was to produce a platform that supports working with secure computation from two distinct perspectives; the protocol developer, developing new secure computation techniques, and the application developer, developing academic prototypes or real world application somehow utilizing secure computation. In this report we validate the platform from these two perspectives by 1) demonstrating how new secure computation techniques can be implemented and integrated into the platform and 2) giving a summary of how the platform has been used in the development of various prototypes of solutions based on secure computation to various real world problems.

We demonstrate how new secure computation techniques can be implemented and integrated into the platform, by two examples. The first example is a formally verified secure computation engine based on Yao's protocol. The second example demonstrates how a Fresco based application can be run using different secure computation techniques, while only modifying a configuration.

The prototypes solutions to various real world problems, includes systems for performing secure surveys, performing genome studies, detecting tax fraud, computing credit ratings and collecting data.

Contents

1 Introduction **1**

2 Protocol Development **2**
- 2.1 Integrating a Formally Verified Secure Computation Engine 2
  - 2.1.1 Protocol Description 3
  - 2.1.2 Implementation 4
  - 2.1.3 Architectural View 5
  - 2.1.4 Integration 6
- 2.2 Application Independence 11
  - 2.2.1 Problem Scenario and Solution 11
  - 2.2.2 Implementation and How to Run it 12

3 Application Development **14**
- 3.1 Sharemind 14
  - 3.1.1 Survey System 14
  - 3.1.2 Genome Studies 15
  - 3.1.3 Tax Fraud Detection 16
- 3.2 Fresco 17
  - 3.2.1 Credit Rating 17
  - 3.2.2 Survey System 18
  - 3.2.3 Data Collection 19
- 3.3 ABY 19
  - 3.3.1 Privacy-preserving biometric matching 20
  - 3.3.2 Private Set Intersection 20
- 3.4 UC Compiler 21
  - 3.4.1 Privacy-preserving applications with private functions 22

4 Summary **23**

List of Figures

2.1 A component view of the architecture. 5
2.2 Overview of the Verfied Yao Protocol Suite 8

3.1 Overview of the ABY framework 20

List of Tables

2.1 Execution times (milliseconds) 11

3.1 Benchmark of Biometric Identification 21
3.2 Benchmark of PSI 21
3.3 Running time and communication for PFE 22

Chapter 1

Introduction

Workpackages 11-14 deals with the design and realization of secure computation protocols. In WP14 we have worked with the implementation of such protocols and frameworks which can in turn be used by application developers to implement full applications, that utilize secure computation as part of their functionality. As such, PRACTICE WP14 draws on the new secure computation techniques designed in WP13 and supplies the implementation and application frameworks used to produce the real world prototypes in Activity 2 (Workpackage 21-24). In D14.1 we first gave a description of an architecture for such a secure computation framework. In D14.2 we reported on a concrete implementation of a secure computation framework following the architecture of D14.1 namely the FRESCO framework. In D14.3 we reported on the implementation of a number of secure computation protocols which have made it into the secure computation frameworks of the PRACTICE platform. Finally, in this report we will validate the platform by demonstrating that it lives up to its goals.

The main goal of WP14 was to produce a platform that supports both the implementation of secure computation protocols, i.e., the underlying cryptographic techniques involved in secure computation, and the implementation of new applications based on secure computation. In other words, a platform that supports the work with secure computation from two distinct perspectives: the protocol developer and the application developer. In the remainder of this report we treat these two perspectives separately. In Chapter 2 we first take the protocol developers perspective and show how the platform supports the development of new protocols and integration of protocol implementations into the framework. Additionally, we give an example of how the framework allows applications to be written independently of the underlying secure computation technique. In Chapter 3 we demonstrate the usability of the PRACTICE platform from the application developer point of view by summarizing a number of the real world scenarios in which the secure computation frameworks of PRACTICE has been used to implement secure computation based solutions.

Chapter 2

Protocol Development

In this chapter we describe the work done to validate the secure computation framework architecture described in D14.1 and implemented in D14.2 from the perspective of the developer of secure computation technologies. In particular one of the main goals of the architecture was a flexible framework allowing to work with multiple secure computation technologies in the same framework. The architecture was designed to make it easy to 1) implement new technologies in the framework and to 2) integrate existing implementations of secure computation technologies into the framework. In deliverable D14.3 we demonstrated 1) by describing the implementation of the TinyTables protocol in the FRESCO framework. In the first part of this chapter we will demonstrate 2) by describing the work done to integrate the formally verified implementation of the Yao protocol, described in D14.3, into the Fresco framework.

Furthermore, an important goal was to make it possible for application developers to work with multiple secure computation technologies without having to know about the internals of each technology. This was achieved in FRESCO by making it possible for to write applications in a format making minimal assumptions on the underlying technology. At run time it should then be possible to chose an appropriate technology simply by configuration of the system. We demonstrate this property of the framework by describing a small application that can be run on top of both the TinyTables and Verified Yao technologies described above.

2.1 Integrating a Formally Verified Secure Computation Engine

The PRACTICE formally verified secure computation framework was outlined in deliverable D22.2 [17], and specified in detail in D12.3 [7]. One fundamental component of this framework is a formally verified secure computation protocol suite that is capable of evaluating arbitary computations expressed as Boolean circuits. The work dedicated to the implementation and validation of this protocol, namely the effort required to obtain a mechanised proof of security and correctness, as well as a formally verified implementation, were conducted within WP14 and were reported in D14.3 [18].

This document describes the integration of the resulting engine implementation, which is automatically generated as OCaml code, into the FRESCO framework, as part of the validation of the general architecture for secure computation engines developed in WP14. The work reported here has also been closely integrated with activities carried out in WP22, namely in the development of formally verified computation specification generation tools that are compatible with the protocol implementation that we present here. More precisely, the computations that can be evaluated by the FRESCO framework and, in particular, the formally verified secure computation engine, can be generated in a certified way by a formally verified compiler that transforms C programs into Boolean circuit descriptions. This complementary effort is presented in D22.4.

2.1.1 Protocol Description

Yao's protocol was described in D12.3, where the formal verification requirements for this high-assurance secure computation protocol suite were specified. We also provide a succinct description here, which is common to that included in D14.4, so as to facilitate the understanding of our description of integration work into FRESCO. Also described in detail in D12.3 are the potential usages of Yao's protocol in a wide range of applications; we omit a discussion of these applications here and refer the interested reader to D12.3 for more details.

Yao's protocol allows one to perform two-party secure function evaluation, i.e., to securely compute functions expressed as circuits composed of Boolean gates, that output a single value to be revealed to both parties, and that take secret inputs from both parties. Informally, Yao's idea of garbling circuits consists of:

- Expressing a circuit as a set of truth tables with information about the wiring between gates.
- *Garble* the Boolean values in the truth tables by replacing the Boolean values with random cryptographic keys, or labels.
- Translate the wiring relation using a system of locks, meaning that for each possible combination of the input wires, the corresponding labels are used as encryption keys that lock the label for the correct Boolean value at the output of that gate.

The evaluation of a garbled circuit is straightforward: given a set of labels representing values for the input wires encoding the inputs of both parties, only one entry in the corresponding truth table will be decryptable, revealing the label of the output wire. The output of the circuit will comprise the labels at the output wires of the output gates.

There are two security assets regarding Yao's protocol: i. unless one is given the association between labels and Boolean values for input and output wires, no information about the input or output of the circuit are revealed; ii. given the label/Boolean value associations for x_1 and the output wires of the circuit, but only an encoding of x_2 in the form of an input label assignment, the garbled circuit reveals nothing other than $f(x_1, x_2)$ about x_2.

To build a Secure Function Evaluation (SFE) protocol between two honest-but-curious parties, one can use Yao's garbled circuits as follows:

1. Bob (holding x_2) garbles the circuit and provides this to Alice (holding x_1) along with the label assignment for the input wires corresponding to x_2 and all the information required to decode the Boolean values of the output wires.

2. Using an oblivious transfer (OT) protocol, Alice obtains the labels that encode x_1 from Bob, without revealing anything about x_1 and learning no more than the labels she requires.

3. Finally, Alice evaluates the circuit, recovering the output, and providing the output value back to Bob.

An oblivious transfer protocol is a particular case of a two-party SFE protocol that will allow one party to obtain the labels corresponding to the encoding of its input without revealing anything about it and without the party learning anything more than the labels it requires.

Formally, party P_1 inputs to the protocol an array of bits of size n, i.e., $I_1 = (x_1, \ldots, x_n) \in 0, 1^n$. Party P_2 inputs to the protocol an n-tuple of pairs of tokens, i.e.,

$$I_2 = ((X_1^0, X_1^1), \ldots, (X_n^0, X_n^1)) \in (\mathsf{Token} \times \mathsf{Token})^n$$

where Token is the type of keys (or labels) associated with the Boolean values of wires in a garbled circuit (bitstrings of fixed value). The evaluation algorithms ev establish that Party P_2 receives no local output at the end of the protocol, whereas Party P_1 obtains $Y = (X_{i_1}^1, ..., X_{i_n}^n)$. We require that ev is so defined for all $n > 0$, thereby imposing that the OT protocol is correct for arbitrary input lengths.

2.1.2 Implementation

Our verified implementation of Yao's protocol was obtained in three steps. We first specified the protocol in EasyCrypt (cf. D22.1 and D12.3), an interactive proof assistant for cryptography. We then used this tool to prove that the protocol is correct and secure, according to the goals specified in D12.3. Finally, we used EasyCrypt and the Why3 framework to extract an OCaml implementation of the verified protocol, which preserves the correctness and security properties of the EasyCrypt specification by construction. This process has been detailed in D14.3 [18]. The resulting OCaml implementation files are the following:

- *Cyclic_group_prime.ml* - abstract operations over a cyclic group of prime order q, in which the oblivious transfer relies on (instantiated using a multiplicative subgroup of integers modulo a large prime).
- *DKC.ml* - implementation of a dual-key cipher encryption scheme based on an abstract implementation of a pseudorandom function (instantiated using AES-128).
- *Hash.ml* - implementation of the SHA256 hash function.
- *Prime_field.ml* - implementation of a prime field corresponding to arithmetic modulo a prime, instantiated using q, used for randomness generation in the OT protocol.
- *Word.ml* - implementation of the operations over bit strings of a fixed length.
- *SFE.ml* - implementation of the concrete definition of the SFE in EasyCrypt. This file represents an OCaml implementation of a garbling scheme, an oblivious transfer protocol and a combination of both that constructs a SFE protocol.

Additionally, the implementation includes files that represent OCaml instantiations for the EasyCrypt native types:

- *ecBool.ml* - OCaml interface for the EasyCrypt Boolean theory.
- *ecIArray.ml* - OCaml interface for the EasyCrypt array theory.
- *ecInt.ml* - OCaml interface for the EasyCrypt integer theory.
- *ecPair.ml* - OCaml interface for the EasyCrypt pair theory.

- *ecPervasives.ml* - OCaml interface for the EasyCrypt pervasives theory.

All of these files are then compiled into a shared library that can be integrated into higher level applications. In what follows, we will describe how the WP14 architecture for the low-level layers in the PRACTICE software stack has facilitated the integration of this shared library into fully operational secure computation applications.

2.1.3 Architectural View

We recall in Figure 2.1 the development view of the PRACTICE general architecture as presented in the draft version of D21.2 [12]. This diagram aims to provide an in-depth view of the different components and how they interact. In this scheme, the activity of Formal Verification (which is fulfilled by our formal verification framework) should be seen as providing high-assurance instances to some of the components shown in the diagram within the DAGGER package: a Secure Computation Protocol Suite and a Secure Language and Compiler capable of generating Secure Computation Specifications.

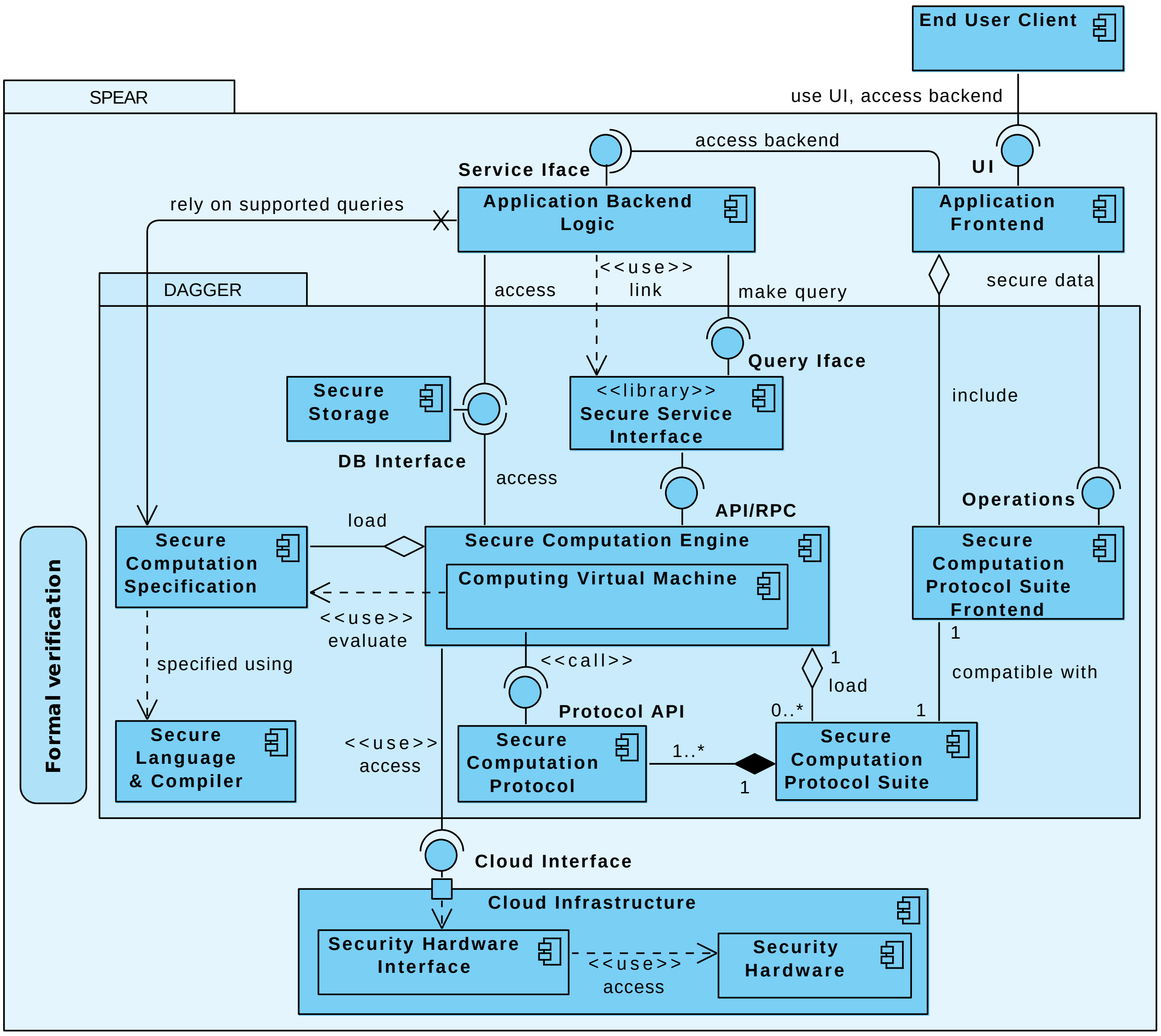

Figure 2.1: A component view of the architecture.

In this deliverable we focus on the lowest layer, which is the target of WP14, and we describe the integration of a new Secure Computation Protocol Suite based on the verified implementation of Yao's protocol presented above, into the PRACTICE software stack via the FRESCO framework. Concretely, we want to embed our secure computation protocol as one of the cryptographic protocols that can be utilized by generic applications developed using the FRESCO toolset, and particularly those that resort to computation descriptions specified using primitive Boolean gate operations.

The FRESCO framework is focused in supporting two different groups of developers: the protocol developers and the application programmers.

A *protocol developer* is responsible for implementing an interface for basic multi-party operations (such as inputting and outputting values), as well as a few basic numeric or binary operators (such as *add*, *multiply*, *and* and *xor*). FRESCO builds on these operators to provide generic versions of more advanced operations (such as *comparison* or *linear program solvers*). Advanced operations inherit properties from the basic protocols, such as performance and the associated security assumptions. In particular, if the properties of a given protocol allow for some optimisation of the advanced operations, the generic FRESCO operation can be replaced by some protocol-specific operation. We will see later in this section how the generic architecture adopted in FRESCO for guiding protocol developers has enabled the rapid integration of the formally verified engine as a new protocol suite.

An *application programmer* will make use of a library operating on a combination of primitive types, implemented as a set of Java objects (SInt, SBool, OInt, OBool) to represent secret or publicly observable integers and Booleans. The application programmer is responsible for combining these objects with a number of Java operations to construct a circuit representing the intended computation. The associated run-time for executing the application is also chosen by the application programmer, and is independent of the circuit. This means that, for instance, a 2-party FRESCO application circuit can be executed using either a Yao-based or a secret based implementation.

Including a new run-time protocol suite does not require a full reimplementation of the network communication, as FRESCO provides a generic interface for sending/receiving byte arrays. Each run-time considered for FRESCO is expected to take an instance of this interface and use it for communicating with the other parties according to the protocol. However, it is likely necessary for the run-time to provide some wrapper around the interface to convert natural run-time types to byte arrays and vice-versa.

In what follows, we describe the steps we took to integrate the shared library produced from OCaml code implementing our verified secure computation protocol into a new FRESCO run-time.

2.1.4 Integration

The integration of our formally verified secure computation engine into FRESCO implied two important challenges in order to bridge the gap between the Java implementation of FRESCO and our formally verified OCaml implementation:

- On the OCaml side, in order to parse the information that was being sent by the Java code (so that it would mach the types involved in the OCaml verified implementation of Yao's protocol) and in order to return to Java information that would be easily used by FRESCO inside its network.

- On the Java side, in order to produce a configuration during the evaluation of the protocol that would correspond to the one used by OCaml.

In a nutshell, our integration strategy is as follows. We use the FRESCO evaluation mechanism in order to build the complete circuit in memory, and then we query the formally verified secure computation engine to execute each state of the protocol. Note that this approach deviates from the usual approach adopted in the FRESCO system. Indeed, FRESCO recommends an *on-the-go* evaluation, meaning that the circuits are evaluated as needed, which brings significant advantages in, for example, memory saving. However, in the formally verified computation engine, the circuit needs to completely known at the beginning of the protocol (following directions of [2], thus requiring it to be built in memory.

Interfacing between OCaml and Java

The most challenging aspect of the integration of the formally verified secure computation engine into FRESCO relied on the interface between OCaml and Java, namely, how to invoke OCaml code in Java. Following related work in this topic, there were two possible approaches that could be used to achieve this goal: using the OCaml-Java compiler or define wrappers for the OCaml code in C and then use JNI in order to invoke C code from Java.

OCaml-Java is a toolset that aims to allow seamless integration of OCaml and Java. The ocamljava compiler is able to generate Java archives that can then be runed on the Java virtual machine. Additionally, it also contemplates a wrapper generation mechanism that produces Java classes that wrap OCaml code. However, there were two major downsides in using this tool.

OCaml-Java is based on OCaml version 4.01.0, which is not compatible with our verified implementation of Yao's protocol. In fact, we relied on the most recent Why3 code extraction mechanism to obtain our implementation, resulting on OCaml version 4.02.3 code. Moreover, the compatibility of the tool with OCaml primitives/libraries is high but it is not perfect and many of the libraries used in the implementation are not yet supported by the tool.

The second approach also carries some significant disadvantages. Recent versions of the OCaml native code compiler do not yet interface correctly with C code, thus restricting the usage of this mechanism.

To surpass this difficulty, we use OCaml as a *blackbox evaluator* that can be called from within Java. We defined four OCaml programs (one for each step of the protocol) that themselves invoke functions from the verified Yao's protocol implementation:

- *p2_stage1.ml* - a program that executes the first stage of the protocol, where party P2 sets up the protocol given its input and its randomness. It outputs the parameters of the OT protocol.

- *p1_stage1.ml* - a program that starts the execution of the OT protocol on the side of party P1.

- *p2_stage2.ml* - a program that executes the OT protocol on the side of party P2. At the end of this program, P2 sends to P1 the necessary information needed to evaluate the garbled circuit.

- *p1_stage2.ml* - the final step of the protocol, where P1 evaluates the garbled circuit, decodes the encoded result and obtains the evaluation of the circuit.

These programs can then be executed inside the Java virtual machine during the execution of a FRESCO application. Naturally, some work is needed to convert between input/output types of both programs.

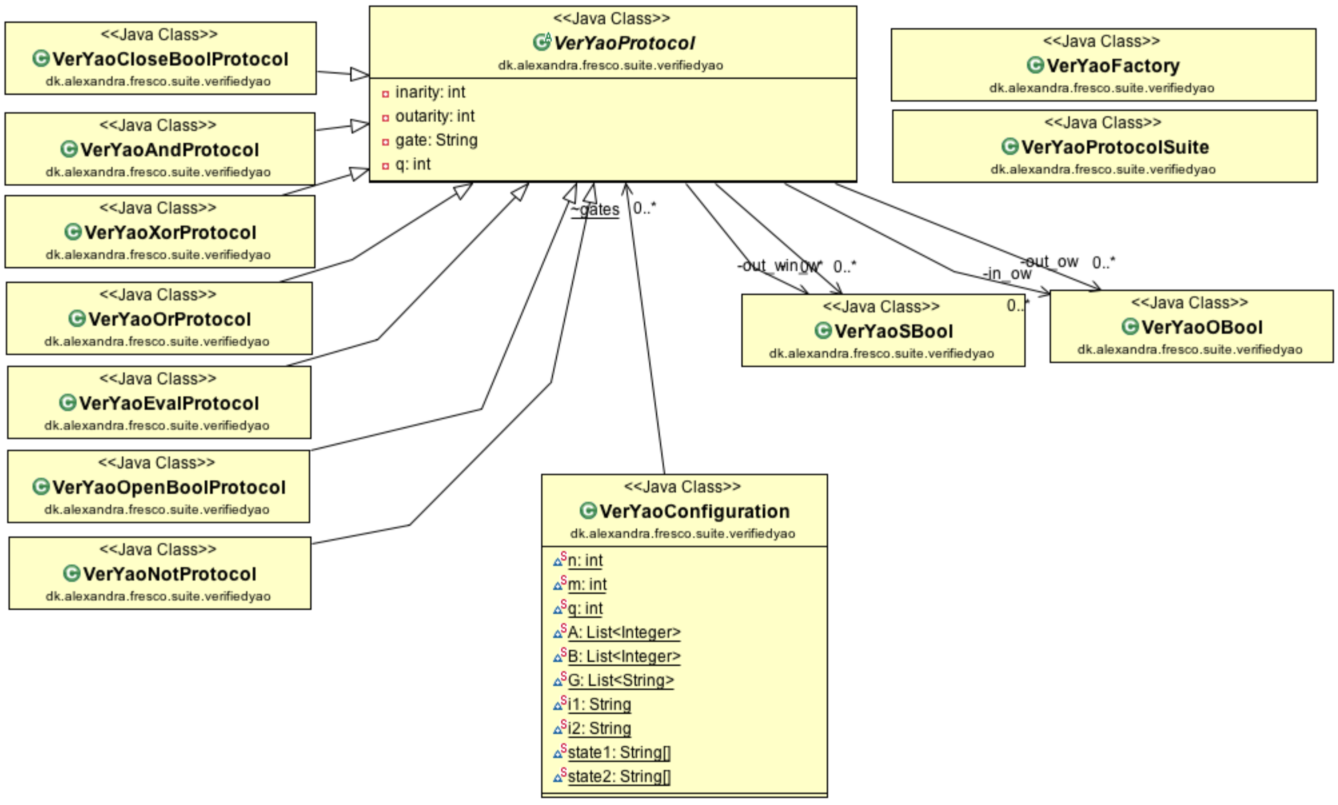

Figure 2.2: Overview of the Verfied Yao Protocol Suite

Type conversion

We concentrated all the efforts regarding type interfacing on the OCaml side because the language contemplates richer libraries concerning type conversion. In fact, we only use strings when transferring data between Java and OCaml due to two main reasons: i. on one hand, receiving the information from the OCaml execution of some protocol's step in the string format requires no additional computation in order to send that information to the other party involved in the protocol using the FRESCO network; and ii. on the other hand, one is able to easily convert strings into every type involved in the protocol (and vice-versa).

A FRESCO verified Yao's protocol suite

The development of our verified Yao's protocol suite inside the FRESCO framework was geared towards the production of a valid input to our OCaml evaluator. To represent this input, we defined a configuration class (*VerYaoConfiguration*) that contemplates the following information:

- The number of input wires of both parties, n.
- The number of output wires of the circuit, m.
- The number of gates of the circuit, q.

- A list storing the first incoming wire of each gate, A.
- A list storing the second incoming wire of each gate, B.
- A list storing the *functionality* of every gate, G. This functionality is described by means of the output column of a truth table. For example, a XOR gate would be represented by the value *0110*.
- The *state* of both parties. This state stores relevant information of both parties that is used throughout the protocol.

The configuration class instance will be constructed during the FRESCO *evaluation* of the protocol. At the end, when finishEval method from FRESCO is invoked, the values of the configuration instance will be used to produce readable input to our OCaml evaluator, that will actually evaluate the circuit according to the inputs of both parties. We provide a more detailed description of the entire protocol suite (including how we transform a FRESCO circuit into circuit that can be evaluated by our OCaml implementation) below.

Verified Yao types

The SFE Yao's protocol evaluates functions described by means of Boolean circuits. Therefore, we are interested in defining two types of Booleans inside our suite: an *open* Boolean, corresponding to the outputs of the circuit that can be released to both parties and a *closed* Boolean, which corresponds to secret inputs on either side. Inside the FRESCO nomenclature, an *open* Boolean is a *public* Boolean such that its value can be read or modified and a *closed* Boolean is a *secret* Boolean such that one is not allowed to read its value.

Verified Yao gates

We represent a circuit gate inside FRESCO by means of an abstract class *VerYaoProtocol*, that all other classes of gate extend. A verified Yao gate has the following attributes:

- The number of input wires of the gate.
- The number of output wires of the gate.
- The input(s) wire(s) of the gate.
- The output(s) wire(s) of the gate.
- The functionality of the gate (using the format described above).
- An integer identifier.

Inside this class, we also define a series of constructors to be used according to the gate. For example, an INV gate (negation gate) should be defined using the constructor that takes one input wire and one output wire. We allow four types of gates in our protocol suite: AND, OR, INV and XOR gates.

Attached to the two types described in section 2.1.4, we also defined two gates:

- *VerYaoOpenBoolProtocol*, that takes a *close* Boolean and transforms it into an *open* Boolean, and that is used to *open* the Boolean values of the output of the circuit, so that they can be read.

- *VerYaoClosedBoolProtocol*, that takes an *open* Boolean and transforms it into a *close* Boolean, and that is used to *close* the Boolean values of the circuit, so that they can be evaluated.

FRESCO gate evaluation

As mentioned above, the FRESCO evaluation process is used to first compute all the values of the configuration class. The circuit, being sequentaly evaluated, is built in memory so that it can subsequently be transformed into an input to the OCaml evaluator. We restrict this process to just one party (with identifier 2), so that there are no unnecessary executions and so that the protocol suite becomes closer to the formalisation.

Consequently, when party 2 invokes the evaluate method in a *protocol gate* (AND, OR, INV or XOR), it will define the functionality of the gate (AND, OR, INV or XOR), assign an identifier to it, store it in a list for further processing and increase the q parameter of the configuration.

Regarding the *closed* Boolean and *open* Boolean gates, whenever the evaluate method is invoked, party 2 will increase the value of its input wires and of the output wires, respectively. Note that, in what concerns the *VerYaoClosedBoolProtocol* gate, we do not restrict the evaluation just for party 2, since we also need to capture the amount of input wires from party 1.

The VerYaoEvalProtocol gate

At the end of the FRESCO evaluation, all the parameters of the configuration instance are defined according to the circuit and one is able to call the formally verified secure computation engine. In order to do so, we defined a new VerYaoEvalProtocol gate that will invoke our OCaml evaluator when the finishEval method form the protocol suite is called.

The VerYaoEvalProtocol gate will execute depending on the party identifier and on the current protocol round and will be responsible for both the evaluation and the exchange of messages. A typical evaluation of this gate would be:

1. Party 2 pre-processes the data of the configuration instance in order to produce valid input to the formally verified secure computation engine and executes the first stage of the protocol, while party 1 is on hold.
2. Upon receiving the first message of the protocol from party 2, party 1 executes the second iteration of the protocol (which is the start of the OT protocol) and sends the second message of the protocol (information about the tokens to receive) to party 2.
3. Party 2 finishes the OT protocol and sends to party 1 the necessary information needed to evaluate the garbled circuit.
4. Finally, party 1 calls the OCaml evaluator in order to obtain the final output of the protocol, corresponding to the evaluation of the circuit. It after informs FRESCO network that the evaluation is completed.

We include a series of JUnit tests of our protocol suite in FRESCO. Those tests include the addition of two 32-bit integers, the multiplication of two 32-bit integers, the AES or DES ciphers, the SHA1 and SHA256 functions and the comparison of two 32-bit integers. In order to run those tests, one simply needs to run the TestVerYaoProtocolSuite class inside the FRESCO test package.

Table 2.1: Execution times (milliseconds)

Circuit	NGates	TTime	TTime FRESCO
ADD32	408	346	528
MUL32	12438	376	7144
AES	33744	1333	33744
SHA1	106761	2993	643015

Performance

For assessing the performance of the formally verified secure computation engine inside the FRESCO platform, we benchmarked some tests mentioned aboved and compared the evaluation time of the protocol when the formally verified secure computation engine is invoked inside FRESCO with the evaluation time when the protocol is invoked by itself. Table 2.1 sums up this comparison.

The significant difference between the execution times arises due to the need to have a circuits pre-processing phase inside FRESCO. Our formally verified secure computation engine requires circuits to be in a specific format, with a series of restrictions defined in [2], and, therefore, there is the need to convert a circuit that was generated by some FRESCO evaluation into a circuit that is a valid input for the formally verified secure computation engine. If the circuit has a small number of gates (like the ADD32 circuit) the difference is barely noticeable, however, for bigger circuits, the disparities are significant.

2.2 Application Independence

In this section we demonstrate how the framework allows the application programmer to write applications based on secure computation but independent of the underlying secure computation technologies used at run time. Concretely, we describe a simple proof-of-concept application implemented in the FRESCO framework, and how it can be run on the TinyTables and formal verified Yao implementations described in this deliverable and deliverable 14.3.

2.2.1 Problem Scenario and Solution

The chosen application is motivated by the following real world problem scenario, which have been discussed with a large consultancy house in Denmark.

The consultancy house is hired to do general statistics on the lending habits of users from two public libraries within the same local region. The dataset from each library, contains a record for each user registered at the library. Each record describes the users book borrowing history. The records identify the user using a national ID number, which is assigned to all persons living in Denmark. The libraries are willing to provide these datasets to the consultancy house. However, because of privacy laws the national ID numbers must be removed, so that the consultancy house can not associate the records to concrete identities of physical persons.

For statistics on the libraries individually this is not a problem because the identities are not relevant. However, when doing statistics across the libraries some users may be registered at both libraries meaning the statistics will be inaccurate. Since, the consultancy house does not have the ID numbers associated with each record, there is no way for them to make the link between records in each data set. Since the libraries are not simply allowed to share their datasets with each other they cannot help in linking the records either.

This exemplifies a frustrating situation where privacy concerns do not allow to do an otherwise simple operation on the datasets. The situations is particularly frustrating because the consultancy house has no interest in learning the ID numbers, they simply need them to link the records.

For our proof-of-concept application we solve this problem using secure computation in the following way. First, we reduce the problem by noting that if the libraries were able to figure out which users have registered at both libraries, the problem is easy to solve. The libraries could simply provide records of the common users in some sorted order, making it easy for the consultancy house to link the records. Thus, we now just need a mechanism for the libraries to find the users they have in common without learning anything else about each other's datasets. Fortunately, this problem, i.e. finding the intersection between two lists, is a classic problem in secure computation also known as private set intersection (PSI).

Here we chose the following basic approach to PSI using general purpose secure computation: we first let the each library input their list of users Id's into the secure computation system. Additionally, the libraries each pick a random and secret 128 bit value and input these values into the secure computation system. We denote these values as k_1 and k_2 respectively. Now using secure computation the libraries jointly compute an AES encryption of each ID in both of their lists under the key $k = k_1 \oplus k_2$. Finally, the output from the secure computation system to both parties is the two lists of encrypted Id's. The libraries can now find their common users simply by comparing the two lists. Since the computation was done inside the secure computation system, and since the Id's are encrypted using a random key unknown to any of the parties, the libraries learn nothing except for the identity of their common users. In particular, the libraries does not learn the identity of the users they do not have in common.

We chose this method as it was relatively easy to implement, since we already had the implementation of AES in the standard library of FRESCO. The solution also offers decent performance as it avoids the naive solution of doing a quadratic number of equality tests using secure computation. We do, however, note that there exists many solutions to the PSI problem, both using general purpose secure computation and dedicated protocols which likely offer considerably better performance. Many of these are developed and described in the PRACTICE project, in e.g. D11.1 or D13.1.

2.2.2 Implementation and How to Run it

The proof-of-concept application was implemented using the FRESCO secure computation application framework. The application code does not make any assumptions on the underlying secure computation technique apart from it being able to work on Boolean values.

The application implementation can be split into two parts. A protocol independent application building phase and a configuration phase:

The application building phase is given a concrete instance of the *ProtocolFactory* class as well as the various (secret) inputs, i.e. the 128 bit k_i and a list of n integers. The *ProtocolFactory.getCloseProtocol()* is used to secret share the two keys and the bit representation of the lists of integers. This results in two size 128 *SBool* arrays and $2 * n$ size 128 *SBool* arrays representing the integer lists[1].

The key bits are then XOR'ed using the *ProtocolFactory.getXorProtocol()* and the result is then used to encrypt the input lists using an AES implementation provided by the *BristolCryptoFactory* class. Note that the constructor of this class takes the *ProtocolFactory* instance as input. This allows the implementation of *BristolCryptoFactory* to be independent of the secure computation technique. The resulting $2 * n$ size 128 *SBool* arrays, representing the ci-

[1]For simplicity it is assumed that both parties input n integers. The application can be expanded to handle lists that are not of equal length, but it would require a round of communication to declare the input lengths.

phertexts, are then opened using *ProtocolFactory.getOpenProtocol()* and can finally be revealed to both parties.

All operations in the application building phase are done using the supplied *ProtocolFactory* instance. This is what allows the actual application to become independent of the secure computation technology. The *ProtocolFactory* class must therefore be instantiated before the application can run. This instantiation is done in the configuration phase, which is also responsible for configuring storage, network, etc. for each party.

While the same tasks related to configuration must always be done, this can be achieved in a manner of ways and is independent of the actual computation. Currently the sample application supports two forms of configuration: as a stand alone jar application or as part of a JUnit test.

- Using the stand alone jar application, the various parameters and inputs are given as commandline arguments.
- Using the JUnit test, each test case creates a configuration which is then passed to a general method which runs the actual computation/application.

The stand alone application is mainly suited to demonstrate an MPC application built using FRESCO. To run the sample application, the following command can be used: *java -jar <jarname> -i1 -s <protocol> -p1:localhost:9994 -p2:localhost:9995 -key:000102030405060708090a0b0c0d0e0f -in1,3,66,1123*.

This will start the application as player 1, using the specified protocol suite, listen for incoming connections on port 9994, connect to player 2 on localhost:9995 and give an input key as well as a list of inputs (1, 3, 66 and 1123). The protocol can be dummy, tinytableprepro, tinytables or some other protocol suite implemented in FRESCO. The *main* method of the class is responsible for parsing the commandline arguments, initializing network, etc.

Implementing and testing new protocols using this approach is however somewhat tedious, as the jar (and the entire FRESCO framework) will have to be rebuilt. This is why it is recommended to utilize the JUnit test case, when implementing a new protocol in FRESCO.

To run the sample application using JUnit, a single testcase for each protocol should be made. The test case must, for each party, create a generic configuration object. Each configuration object contains some general configuration data such as and adress for each party, what kind of storage should be used, etc. More important, the configuration object also contains a reference to concrete implementations of a protocol suite and the corresponding protocol suite configuration. Once the set of configurations have been properly instatiated, the set of configurations are passed as parameters to a generic test method, which will run the MPC application according to the provided configurations. Assuming the chosen protocol suite implements the required interface, i.e. supports various boolean operations such as XOR, the JUnit test should succeed.

Chapter 3

Application Development

In Chapter 2 the focus was mainly on the platform from the point of view of the developer of new secure computation technologies. In this chapter we focus on the platform from the point of view of the application developer. We do this by giving a short summary of various scenarios where the developed frameworks have been used in more or less real world oriented prototypes. Many of these applications have already been discussed in more detail in other deliverables, thus the goal here is mainly to provide a convenient overview.

In the following sections, we first give a short introduction to the different frameworks and then describe the applications that has been developed in those frameworks.

3.1 Sharemind

Sharemind is currently one of the most mature frameworks for general secure computation [1]. It supports the development of secure computation based applications through its domain specific language SecreC. These applications can then be evaluated on the Sharemind system. Previous versions of the Sharemind system was focused on secure computation using a fixed secure computation technique involving three parties in the semi-honest security setting. It has since evolved to a more flexible framework following an architecture similar to that described in D14.1. Thus, the current version of Sharemind allows application to be evaluated on multiple different protocol suites (called *Protection Domains*), both with various numbers of parties and security properties. Additionally, Sharemind features a large library of highly optimized secure computation functionality, ranging from basic arithmetic to advanced statistics, that can be used in the development of secure computation applications. For more details on Sharemind we refer deliverable D22.1.

In PRACTICE, Sharemind has been extended with new protocols in the underlying protocol suites as described in D14.3. Additionally new tools to help the development of Sharemind applications have been developed and described in D22.2.

The Sharemind framework has been used to develop a number of prototypes which we will summarise in the following sections.

3.1.1 Survey System

Surveys are commonly used to collect sensitive information on individuals in order to derive aggregated information on a larger group of individuals. This happens, for example, in personal health surveys or employee satisfaction surveys. In these cases it is important for the quality of the gathered information, that the respondents answer the survey honestly. This in turn often requires that the respondents can be confident that their private answers will not be leaked. Traditionally, this is ensured by letting some trustee (e.g., a consultancy firm) collect survey answers, analyze the answers and announce the aggregated statistic information, without revealing the answers of the individual respondents. This solution, however, requires trusted third party, which in some cases can be hard and/or expensive to come by. This trusted third party can be replaced by an MPC computation between multiple stakeholder, e.g. an employee representative and the employer in the employee satisfaction case.

Deliverable D23.1 described a prototype of a survey system replacing the traditional central trustee with a secure computation solution. Removing the single point of trust would mean that leaking private information would require collaboration across organizations. Thus risk of leaking individual answers either intentionally or by error should be reduced.

The initial prototype allows an *organizer* to design surveys, including traditional survey question types such as ratings, multiple choice selection, and free text in a modern web-based UI. The respondents answer the survey using a similar web UI with the individual answers stored securely in the backend. Once the survey is complete the organizer can request a simple analysis such as histograms. Additionally, the free text questions can be revealed, but in shuffled order to protect the identity of the respondent. These analyses are computed using the secure computation backend, and thus no other information than the results of the analysis is revealed to the organizer or any other party in the system. Depending on the application scenario, the organizer can then decide to publish the findings from the survey.

Following the initial prototype described in D23.1, the prototype has been considerably improved and the functionality extended in D23.3. The current version now supports additional question types, such as numeric questions, and analyses, such as measures of correlation between questions and filtering of respondents into groups based on selected questions.

The prototype was developed with a common frontend implemented using essentially generic web technologies and interchangeable secure computation backends implemented on either the Sharemind or FRESCO frameworks. The prototype version based on the Sharemind backend implementation was deployed as a three party setup, running a protocol suite based on additive secret sharing, providing security against semi-honest corruption. The main work on the prototype in terms a secure computation perspective was implementing the analyses of the survey answers on Sharemind. These analysis mainly required simple arithmetic operations and a few more complex statistical computations, all relatively easily implementable given the extensive standard library of Sharemind.

The Sharemind based secure survey prototype has been used a number of real world case studies: inside PRACTICE the system was used in D24.3 to run a survey among companies in the aeronautics industry about their use of cloud technologies and their perceived risks in doing so. The PRACTICE partner Cybernetica also used the system internally to conduct the companies employee satisfaction survey. Finally, the second largest city in Estonia, Tartu, also used the system to run an satisfaction survey among the 300 employees of the city government. We refer to deliverable D23.3 for more information on these case studies. The case studies were all executed satisfactory; validating that the secure survey system is indeed a viable solution when dealing with surveys collecting and analyzing confidential information.

3.1.2 Genome Studies

The analysis of the genome data of human individuals has great medical promise, however, also comes with obvious privacy concerns. For this reason since 2014 iDASH group of the University of California San Diego has hosted the Privacy and Security workshop. The workshop focuses on applying secure computation to various to problems in genome analysis. Each year iDASH challenges teams in a competition to produce solutions to specific privacy related real world problems in genome analysis. The submitted solution are then judged on performance, accuracy and security.

In 2016 Cybernetica participated in the competition with a solution to the following problem: Given a gene sequence of a cancer patient a medical researcher wants to find the patients with the most similar genes at a hospital hosting a large database of patients gene sequences. However, neither the medical researcher nor the hospital is allowed to reveal the gene sequences they are holding. To solve this problem the parties should use secure computation to reveal the identies of the k most similar patients without revealing any of the concrete genome data. The problem is further complicated by the fact that a measure known as *edit distance* is used as the relevant measure of similarity. However, computing edit distance is computationally expensive to the point that secure computation currently becomes infeasible. Thus part of the challenge involved coming up with an appropriate approximation to the edit distance measure.

The solution entered by Cybernetica is based on the Sharemind framework, using a novel secure computation technique based on both Boolean and arithmetic secure computation with semi-honest security. Solution required the implementation of many interesting secure computation functionalities such as matrix multiplication shuffling, indexing and sorting. The developed solution is too extensive to cover in this summary and we refer the interested reader to D23.3 for details.

At the time of writing the 2016 iDASH competition has not been judged yet, so we can not say how the Sharemind solution compares. However, the solution was benchmarked on a two Xeon E2-2640 v3 servers with 128GB of RAM, connected by either a 10Gbit/s LAN or a 10Mbit/s WAN. In the LAN setup making 200 queries against a database of 500 gene sequences took 13 minutes. The WAN setup required 149 minutes.

3.1.3 Tax Fraud Detection

In 2013 Estonia had an estimated loss of 220 million euros in tax revenue do to Value Added Tax (VAT) fraud. VAT should be paid when whenever a product is sold, and companies can cheat by simply neglecting to declare correctly the amount of sales to other companies. The cheating could go undetected because the limited information accessible by the tax authorities meant that investigating cases became extremely time consuming.

A suggested solution to this problem was by law to force all companies the register every sale and purchase transaction made with another company with the Estonian tax authorities. Roughly speaking, the tax authorities could this way simply balance all purchases and sales in order to detect undeclared sales. This solution, however, was politically rejected. Creating a central database containing the sensitive business data at the tax authorities was considered too large of a privacy risk for the companies of Estonia.

As an alternative, a prototype solution was build in collaboration with the Estonian tax authorities using the Sharemind framework. The prototype gathers essentially the same transaction information described above. However, instead of storing the information centrally, the information is stored securely in the distributed secure computation system. Computing which companies are likely to have committed fraud, is then done using secure computation. If, for some companies, the transactions do not add up, the tax authorities can be advised to look in to the discrepancies. This way, the Sharemind based solution reveals to the tax authorities with high accuracy only which companies may be suitable target for further investigation. In particular no one organization would have direct access to the transaction information of any of the companies.

The prototype was set up as a three party secure computation system using the additive secret sharing based protocol suite of Sharemind. While the analysis done to balance transactions is rather simple, the main challenge in this prototype was scaling the secure computation system to deal with the large amount of transaction data generated monthly by Estonian businesses (estimated at 50 million transactions per month).

The initial prototype reported on in [3] was estimated capable of handling the monthly data analysis in 10 days on 20,000 euros worth of hardware. This was not satisfactory to tax authorities, as they had only three days to processes the VAT returns. However, a later version of the prototype reported on in deliverable D21.3, greatly improved performance using parallelization similar to techniques used in Big Data analytics such as MapReduce. The improved prototype was reported to process the monthly data in between 3 to 9 hours depending on the network setup and cost between 70 and 200 dollars to run on Amazons AWS cloud. Unfortunately, while this was well within the three day time limit, the new political agreement was in place to let the tax authorities do a similar calculation without using secure computation.

3.2 Fresco

The prototype implementation of the FRESCO framework for general secure computation was described in deliverable D14.2 following the architecture of D14.1. FRESCO is still under development and through WP14 it has been further improved in relation to user friendliness and stability. Furthermore, the new protocols suites described in Chapter 2 was added, giving more flexibility in the choice of underlying secure computation technologies to use when working with FRESCO based applications. Through the lifetime of the PRACTICE project the framework has been validated through the implementation of a few prototype applications. These will be described in the following sections.

FRESCO is a framework for developing both new cryptographic protocol suites, but also for building secure computation applications. The concept is to separate the two roles which enables an application developer to write secure computation applications without any knowledge of the protocol suite that powers his application. At the same time, constructing a new protocol suite can be done without knowing how to write applications, but the new protocol suite can still be tested using the protocol suite agnostic tests in FRESCO. FRESCO also works in a streaming fashion by allowing construction of the circuit to be evaluated on the fly. This is not always possible, and depends on the protocol suite used. There are a lot of facets in choosing the right configurations for an application, and therefore it is a great advantage when using FRESCO that the user can switch protocol suite and other settings by just changing a few lines in a properties file, while keeping the application as it is. This way, the optimal balance between performance, security requirements, number of servers etc. can quickly be determined. We refer the reader to D14.2 for a much more extensive in-depth walk through of FRESCO.

3.2.1 Credit Rating

The credit rating application is described in depth in deliverable D23.2, and then further extended in deliverable D23.3. The application enables banks to access a much larger data foundation when determining if the performance of their portfolio of farmers, and when benchmarking potential new customers. The data foundation comes from a consultancy company for farmers (SEGES) with access to a lot of financial records from farmers all over Denmark. The system reveals nothing to SEGES, and only reveals the benchmarks and financial data of the banks own customers.

The prototype was controlled by The Alexandra Institute, but the idea is to let SEGES control one server and let The Danish Bankers Association control the other. SEGES initiates the application by secret sharing the financial data of the farmers with The Danish Bankers Association. Along with this, SEGES also secret shares the benchmarking score for each customer. This can be done in the clear as they have all the data needed. After this step, a bank can log in to the system and upload a list of identifiers for his customers. This is then secret shared in the browser using the technique described in [9]. For each segment of farmers, the bank can now request an analysis which benchmarks it's portfolio against the entire dataset of SEGES. This is done using a relative naive approach, where each identifier uploaded by the bank is compared with all identifiers in the segment. If a match is found, the financial data and benchmarking score is copied to the banks' table entry in the database. This table is then output to the bank employees browser using the technique also described in [9]. JavaScript code then filters away results where no match was made and presents the remaining results to the user through a table view or a graph representation.

The secure computation used here is quite simple, but that is not the case for the second use case where a new customer needs to be benchmarked. Here the benchmark score is needed without revealing the financial data, and since we don't have the data from SEGES in the clear, this means that we need to do linear programming within the secure computation. More details on this can be found in D23.2. We used the protocol suite SPDZ for the task since we worked with arithmetic numbers, and the SPDZ protocol suite os the fastest for this purpose in FRESCO at the time writing. The performance numbers for the linear programming can also be found within [9], but to get an idea of the speed, consider a single dataset which has to be benchmarked up against a dataset of 70 entries. Each entry consists of 6 variables and the linear program has 4 constraints. The computation was run on two amazon *m4.large* instances which has 8GB RAM, and 2 cores running 2.4 GHz Intel Xeon processors. Solving the linear program on these machines took 23 seconds on average.

The prototype was tested in a real world setting by a number of Danish banks. Their reaction was positive, indicating that the extra information provided could be usefull for their credit rating process, and that the performance was adequate.

In order to demonstrate interoperability between secure computation applications the FRESCO version of credit rating prototype and the survey system (described below) was additionally extended in D23.3. This was motivated by feedback from the banks in testing of the initial prototype. Namely, the banks requested more subjective information on the customers to factor into the credit rating. For this purpose we designed a combined use case where subjective information can be collected using the survey system, and then securely exported in to the credit rating application. In the credit rating application this data would then be linked to the objective scores of the survey respondents and aggregated information would be computed using secure computation and displayed to the bank.

3.2.2 Survey System

As described in Section 3.1.1, the survey system described in D23.1 and extended in D23.3 was implemented to be able to run on both a Sharemind and FRESCO backend. From the application user point of view there is, however, little difference between the two versions. On the backend, the FRESCO version of the survey system is implemented using the two party version of SPDZ protocol providing security against one malicious corruption. What this essentially means is that the FRESCO version provides stronger security, but at the cost of considerable worse performace.

3.2.3 Data Collection

Within the deliverable D21.3 we described the architecture of an application for collecting data as a foundation for further analyses on that data foundation. This application is currently being developed using FRESCO in project Big Data by Security fonded by the Danish Industry Foundation. The application focuses only on the first part of large class of secure computation application. Namely, the data collection phase. In many applications, we want to run analyses against a data foundation collected from not just one provider, but on a joint set.

There are three roles in the application: Data Providers, Data Users and Organizers. The latter is responsible for the data collection process i.e. which data should which data provider provide and how will the final joint dataset look like. He also decides how to handle possible data corruption, lacking values and collisions in entries. Data providers are responsible for providing data in the format described by the organizer, and data users can use the joint datasets to do analyses if allowed by the organizer. Which analyses can be done is inherently specific to the use case and domain of the data users. Thus, no generic application can be developed for that purpose, but we here list a few examples:

The credit rating application described in Section 3.2.1 could use the data collection application as a way to include more data providers than the single consultancy house mentioned. There might be accountants or even banks who could input financial data about the farmers such that the joint data foundation would be even richer and thus give more accurate results.

Consider also medical data which is normally not legally allowed to share with others. This might be legally possible with this application if the correct setup could be found. This would enable general practioners and hospitals to create a joint secret shared database which could benefit society if the analyses resulted in better medicine, better diagnostics, etc.

The system is still in development and funded by the Danish project Big Data by Security funded by the Danish Industry Foundation. It is therefore not possible to say how much data can be processed how fast, but it depends on the choices of the organizer. If e.g. collisions should be checked, this will be a lengthy process since it must be done as a secure computation as the data cannot be revealed to either the organizer or the MPC servers to which the data is secret shared. The Big Data by Security project intends to develop prototypes using the data collection application as a basis for the prototypes. One of those concerns energy data from both Statistics Denmark and Energinet.dk, the latter being the Transmission System Operator (TSO) in Denmark. The data is joined in order to improve benchmarks of companies energy consumption. A company can enter their data securely in the browser and get a benchmark of their performance against the best companies, comparable to their own.

3.3 ABY

In secure two-party or secure multi-party computation, the function is often to be expressed and evaluated as either a Boolean or arithmetic circuit. As described in deliverable D13.1 [14], ABY allows for mixing the secure computation protocols that are used for the secure evaluation of the circuit. ABY efficiently combines arithmetic sharing, Boolean sharing with the GMW protocol and Yao's garbled circuits.

The prototype implementation of the ABY secure computation framework from [10] is available as an open source project at `https://github.com/encryptogroup/ABY` and is described in more detail in deliverable D22.4 [4, 5] and D14.3 [18].

3.3.1 Privacy-preserving biometric matching

For benchmarking ABY, we have implemented some prototype applications that were described in [18] and in [10]. The first such prototype is **privacy-preserving biometric matching**, where one party wants to determine whether its biometric sample matches one of several biometric samples that are stored in a database held by another party. A fundamental building block of these protocols is to compute the squared Euclidean distance between the query and all biometrics in the database and afterwards determine the minimum value among these distances. In our experiments, each sample has $d = 4$ dimensions and each element is 32-bit long, but we increase the database size to $n = 512$ entries. More specifically, we securely compute $\min\left(\sum_{i=1}^{d}(S_{i,1} - C_i)^2, \ldots, \sum_{i=1}^{d}(S_{i,n} - C_i)^2\right)$, where P_0 inputs the database $S_{i,j}$ and P_1 inputs the query C_i.

We benchmark four different instantiations: a pure Yao-based variant (**Y**-only), a pure Boolean-based variant (**B**-only), a mixed-protocol that uses Arithmetic sharing for the distance computation and Yao sharing for the minimum search (**A+Y**), and a mixed-instantiation that uses Arithmetic sharing for the distance computation and Boolean sharing for the minimum search (**A+B**). For each instantiation, we give the setup, online, and total run-time, overall communication, and number of rounds in the online phase in Table 3.1.

3.3.2 Private Set Intersection

In the private set intersection application, two parties want to identify the intersection of their n-element sets, without revealing the elements that are not contained in the intersection. Boolean circuits that compute the private set intersection functionality were described in [13] and evaluated using Yao's garbled circuits protocol. For bigger sets with elements of longer bit-lengths, the sort-compare-shuffle set intersection circuit was shown to be most efficient; for sets with n σ-bit elements this circuit has $\mathcal{O}(\sigma n log^2 n)$ AND gates.

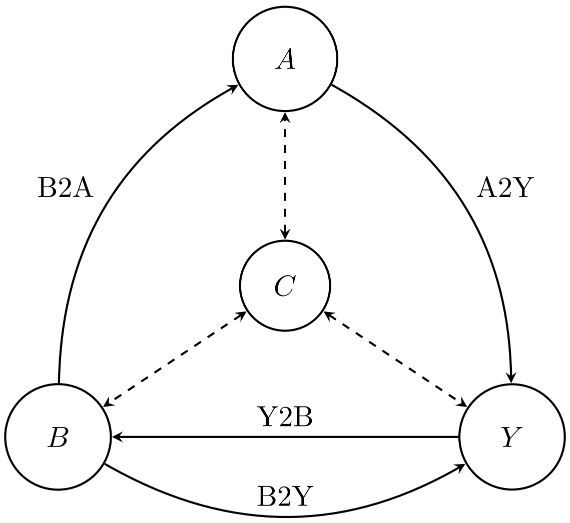

Figure 3.1: Overview of the ABY framework that allows efficient conversions between **C**leartexts and three types of sharings: **A**rithmetic, **B**oolean, and **Y**ao.

We implement the sort-compare-shuffle circuit of [13] in our ABY framework and instantiate it in three versions: a Yao-only instantiation (**Y**-only), a Boolean-only instantiation (**B**-only), and a mixed-instantiation (**B+Y**) that evaluates the sort and compare parts using the Yao sharing and the shuffle part using the Boolean sharing. The Boolean sharing benefits from the improved evaluation of MUX operations that frequently occur in the shuffle part of the circuit. We run all three instantiations in the local and cloud setting and compare their setup, online, and total run-time as well as their communication complexity and number of rounds in Table 3.2

Table 3.1: Biometric Identification: Setup, Online, and Total run-times (in s), communication, and number of messages for biometric identification on 512 elements with a length of $\sigma = 32$-bits and with dimension $d = 4$ and long-term security. Smallest entries marked in bold.

	Local			**Cloud**			**Comm. [MB]**	**#Msg**
	S	O	T	S	O	T		
Y-only	2,24	0,31	2,55	23,78	0,84	24,62	147.7	**2**
B-only	2,15	0,28	2,43	10,34	29,07	39,41	99.9	129
A+Y	0,14	**0,05**	**0,19**	2,98	**0,44**	**3,42**	5.0	8
A+B	0,08	0,13	0,21	2,34	24,07	26,41	**4.6**	101

Table 3.2: PSI: Setup, Online, and Total run-times (in s), communication, and number of messages for the Private Set Intersection application on $n = 4\,096$ elements of length $\sigma = 32$-bits and long-term security. Smallest entries marked in bold.

	Local			**Cloud**			**Comm. [MB]**	**#Msg**
	S	O	T	S	O	T		
Y-only	3,5	0,7	4,3	32,2	**1,8**	34,0	247	**2**
B-only	2,0	**0,6**	**2,6**	11,5	22,6	34,1	**163**	123
B+Y	2,6	0,7	3,3	23,4	7,1	**30,0**	182	27

3.4 UC Compiler

Universal circuits (UCs) can be programmed to evaluate any circuit up to a given size k. They provide elegant solutions in various application scenarios, e.g. for private function evaluation (PFE). The optimal size of a universal circuit is proven to be $\Omega(k \log k)$. Valiant proposed a size-optimized UC construction in [19], which has been put in practice in [15].

Any computable function $f(x)$ can be represented as a Boolean circuit with input bits $x = (x_1, \ldots, x_u)$. Universal circuits (UCs) are programmable circuits, which means that beyond the u inputs, they receive $p = (p_1, \ldots, p_m)$ program bits as further inputs. Using these program bits, the UC is programmed to evaluate the function, such that $UC(x, p) = f(x)$.

The most prominent application of UCs is the evaluation of private functions based on *secure function evaluation* (SFE) or *secure two-party computation.* Many secure computation protocols use Boolean circuits for representing the desired functionality. In some applications the function itself should be kept secret. This setting is called *private function evaluation* (PFE), where only one of the parties P_1 knows the function $f(x)$, whereas the other party P_2 provides the input to the private function. P_2 learns no information about f besides the size of the circuit defining the function and the number of inputs and outputs.

PFE can be reduced to SFE [16] by securely evaluating a UC that is programmed by P_1 to evaluate the function f on P_2's input x. Thus, P_1 provides the program bits for the UC and

Circuit	**UC Compile**	**UC I/O**	**GMW**		**Yao**	
	Time (ms)	Time (ms)	Time (ms)	Communic. (KB)	Time (ms)	Communic. (KB)
`Branching_18`	4,8	31,4	26.23	307,77	17.34	145,87
`CreditCheck`	1,2	11,4	26.25	113,35	5.67	45,15
`MobileCode`	3,2	26,3	25.71	202,50	28.16	103,45

Table 3.3: Running time and communication for our UC-based PFE implementation with ABY. We include the compile time, the I/O time of the UC compiler, and the evaluation time (in milliseconds) and the total communication (in Kilobytes) between the parties in GMW as well as in Yao sharing.

P_2 provides his private input x into an SFE protocol that computes a UC. The complexity of PFE in this case is determined mainly by the complexity of the UC construction. The security follows from that of the SFE protocol that is used to evaluate the UC. If the SFE protocol is secure against semi-honest, covert or malicious adversaries, then the PFE protocol is secure in the same adversarial setting.

3.4.1 Privacy-preserving applications with private functions

[8] shows an application for secure computation, where evaluating UCs or other PFE protocols would ensure privacy: when *autonomous mobile agents* migrate between several distrusting hosts, the privacy of the inputs of the hosts is achieved using SFE, while privacy of the mobile agent's code can be guaranteed with PFE. Privacy-preserving *credit checking* using garbled circuits is described in [11]. Their original scheme cannot represent any policy, though by evaluating a UC, their scheme can be extended to more complicated credit checking policies. Privacy-preserving evaluation of *diagnostic programs* was considered in [6], where the owner of the program does not want to reveal the diagnostic method and the user does not want to reveal his data. In the protocol presented in [6], the diagnostic programs are represented as binary decision trees or branching programs which can easily be converted into a Boolean circuit representation and evaluated using PFE based on universal circuits.

We validated the practicality of Valiant's universal circuit construction with an efficient implementation. We ran our experiments on two Desktop PCs, each equipped with an Intel Haswell i7-4770K CPU with 3.5 GHz and 16 GB RAM, that are connected via Gigabit-LAN and give our benchmarks in Table 3.3. We show the real practicality of UCs through experimental results proving the efficiency of our implementation of PFE with the ABY framework [10].

Chapter 4

Summary

In this report we have demonstrated how the platform for secure computation can be used from the two distinct perspectives; the protocol developer, developing new secure computation techniques, and the application developer, developing academic prototypes or real world application somehow utilizing secure computation.

It was shown how protocol developers can implement and integrate new secure computation techniques into the platform and use existing application and test code to verify the correctness of the new protocol.

A number of prototypes were described to show how the platform has been used to provide secure computation solutions to a wide array of real world problems.

Bibliography

[1] David W. Archer, Dan Bogdanov, Benny Pinkas, and Pille Pullonen. Maturity and performance of programmable secure computation. *IEEE Security and Privacy*, 14(5):48–56, Sept 2016.

[2] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 784–796, New York, NY, USA, 2012. ACM.

[3] Dan Bogdanov, Marko Jõemets, Sander Siim, and Meril Vaht. How the estonian tax and customs board evaluated a tax fraud detection system based on secure multi-party computation. In *Financial Cryptography and Data Security - 19th International Conference, FC 2015, San Juan, Puerto Rico, January 26-30, 2015, Revised Selected Papers*, volume 8975 of *LNCS*, pages 227–234. Springer, 2015.

[4] Jonas Bohler, Florian Hahn, Raad Bahmani, Daniel Demmler, Agnes Kiss, Thomas Schneider, Michael Stausholm, Reimo Rebane, Jose Bacelar Almeida, Manuel Barbosa, Hugo Pacheco, Vitor Pereira, and Bernardo Portela. PRACTICE Deliverable D22.3: software development kit and tools prototype, 2015.

[5] Jonas Bohler, Florian Hahn, Daniel Demmler, Agnes Kiss, Thomas Schneider, Michael Zohner, Kasper Damgaard, Karl Tarbe, Reimo Rebane, Ville Sokk, Jose Bacelar Almeida, Manuel Barbosa, and Hugo Pacheco. PRACTICE Deliverable D22.4: software development kit and tools prototype (final version), 2016.

[6] Justin Brickell, Donald E. Porter, Vitaly Shmatikov, and Emmett Witchel. Privacy-preserving remote diagnostics. In *ACM CCS'07*, pages 498–507. ACM, 2007.

[7] Niklas Buescher, Peter Nordholt, Dan Bogdanov, Roman Jagomägis, Jaak Randmets, José Bacelar Almeida, Bernardo Portela, and Hugo Pacheco. PRACTICE Deliverable D12.3: formal verification requirements, 2015. Available from `http://www.practice-project.eu`.

[8] Christian Cachin, Jan Camenisch, Joe Kilian, and Joy Müller. One-round secure computation and secure autonomous mobile agents. In *International Colloquium on Automata, Languages and Programming (ICALP'00)*, volume 1853, pages 512–523, 2000.

[9] Ivan Damgård, Kasper Damgård, Kurt Nielsen, Peter Sebastian Nordholt, and Tomas Toft. Confidential benchmarking based on multiparty computation. Technical report, Cryptology ePrint Archive, Report 2015/1006, 2015.

[10] Daniel Demmler, Thomas Schneider, and Michael Zohner. ABY – a framework for efficient mixed-protocol secure two-party computation. In *Network and Distributed System Security (NDSS'15)*. The Internet Society, 2015. Code: `http://encrypto.de/code/ABY`.

[11] Keith B. Frikken, Mikhail J. Atallah, and Chen Zhang. Privacy-preserving credit checking. In *ACM Electronic Commerce (EC'05)*, pages 147–154, 2005.

[12] Isabelle Hang, Ferdinand Brasser, Niklas Buescher, Stefan Katzenbeisser, Ahmad Sadeghi, Kai Samelin, Thomas Schneider, Jakob Pagter, Peter Sebastian Nordholt Janus Dam Nielson, Kurt Nielsen, Johannes Ulfkjaer Jensen, Dan Bogdanov, Roman Jagomägis, Liina Kamm, Jaak Randmets, Jaak Ristioja, Reimo Rebane, Jaak Ristioja, Sander Siim, Riivo Talviste, Manuel Barbosa, Bernardo Portela, Rui Oliveira, Stelvio Cimato, and Ernesto Damiani. PRACTICE Deliverable D22.1: tools: State-of-the-art analysis, 2013. Available from `http://www.practice-project.eu`.

[13] Yan Huang, David Evans, and Jonathan Katz. Private set intersection: Are garbled circuits better than custom protocols? In *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012*. The Internet Society, 2012.

[14] Florian Kerschbaum, Florian Hahn, Thomas Schneider, Michael Zohner, Pille Pullonen, and Claudio Orlandi. PRACTICE Deliverable D13.1: a set of new protocols, 2015. Available from `http://www.practice-project.eu`.

[15] Ágnes Kiss and Thomas Schneider. Valiant's universal circuit is practical. In *Advances in Cryptology – EUROCRYPT'16*, 2016.

[16] Vladimir Kolesnikov and Thomas Schneider. A practical universal circuit construction and secure evaluation of private functions. In *Financial Cryptography and Data Security (FC'08)*, volume 5143, pages 83–97, 2008. Code: `http://encrypto.de/code/FairplayPF`.

[17] Tobias Mueller, Niklas Buescher, Hiva Mahmoodi, Janus Dam Nielsen, Peter S. Nordholt, Dan Bogdanov, Manuel Barbosa, Johannes U. Jensen, and Kurt Nielsen. PRACTICE Deliverable D22.2: Tools design document, 2014.

[18] Peter Nordholt. PRACTICE Deliverable D14.1: protocol implementations, 2015. Available from `http://www.practice-project.eu`.

[19] Leslie G. Valiant. Universal circuits (preliminary report). In *Proceedings of the Eighth Annual ACM Symposium on Theory of Computing*, STOC '76, pages 196–203, New York, NY, USA, 1976. ACM.