D13.4

Prototype Implementations of Key Protocols

Project number:	609611
Project acronym:	**PRACTICE**
Project title:	Privacy-Preserving Computation in the Cloud
Project Start Date:	1st November, 2013
Duration:	36 months
Programme:	FP7/2007-2013
Deliverable Type:	Report
Reference Number:	ICT-609611 / D13.4 / 1.0
Activity and WP:	Activity 1 / WP13
Due Date:	October 2016 - M36
Actual Submission Date:	3rd November, 2016
Responsible Organisation:	BIU
Editor:	Benny Pinkas
Dissemination Level:	PU
Revision:	1.0
Abstract:	This deliverable describes prototype implementations of key secure multi-party computation protocols. In particular, it describes the LibSCPAI software library which is C++ library for secure computation, an extensive set of implementations of oblivious transfer protocols, and a prototype implementation of Valiant's universal circuit construction.
Keywords:	Secure multi-party computation

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

Editor

Benny Pinkas (BIU)

Contributors (ordered according to beneficiary numbers)

Agnes Kiss (TUDA)
Tomas Schneider (TUDA)
Michael Zohner (TUDA)

Executive Summary

This deliverable describes prototype implementations of key secure multi-party computation protocols. All implementations are open source, and the code can be found at github.com.
The first chapter describes the LibSCPAI software library, which is a C++ library for secure computation. This library grew out of the SCAPI library, which was implemented in Java. LibSCPAI implements high performance state-of-the-art protocols for secure multi-party computation.
The second chapter describes an extensive set of implementations of oblivious transfer extension protocols. These protocols enable to perform a small number of preprocessing public key operations which enable to compute a large number of oblivious transfer operations at much lighter cost of symmetric key operations.
The third chapter describes a prototype implementation of Valiant's universal circuit construction. This construction enables to run a secure computation of a function while hiding the function that is being computed.

Contents

List of Figures

Chapter 1

SCAPI and LibSCAPI

LibSCAPI is the C++ high performance version of the SCAPI library (Secure Multiparty Computation API). The library was rewritten to take advantage of vectorized architectures that can process many computation (e.g., many circuits) in parallel. LibSCAPI is developed by Bar Ilan University Cryptography Research Group.

Availability LibSCAPI is an open source project which is available for download at `https://github.com/cryptobiu/libscapi`. The library contains about 50,000 lines of code.

1.1 Goals

The goal of LibSCAPI is to promote research by academia and by industry in the field of secure multi-party computation.
The library was designed and implemented to support the following properties:

- A consistent API over the different components of a secure multi-party computation system. Namely, the primitives, mid-layer protocols, interactive mid-layer protocols and communication channels. The consistent API simplifies the development and evaluation of new protocols.
- Making LibSCAPI easy to build and use.
- Integrating into LibSCAPI the best performance open-source implementations that were done by other academic researchers. As an example, LibSCAPI contains the most up-to-date implementations of oblivious transfer extension, as were developed by Asharov et al. [3] and Keller et al. [9].
- Achieving high performance performance on a standard Linux & Intel x64 Architecture.

  The library uses modern techniques like Intel intrinsics instructions, pipelining, and TCP optimizations. However, it avoids using techniques that are too advanced or not available on common platforms (such as Intel AVX-512 and DPDK, GPGPU etc.).
- Providing a common platform for benchmarking different alogirthms and implementations.

1.2 LibSCAPI Modules

The LibSCAPI library includes the following modules.

Primitives: This module includes the basic cryptographic primitives that are used by all layers of LibSCAPI. These primitives include the following functionalities:

- Discrete-logarithm based cryptographic operations (both modulo a prime, and in elliptic curve groups).
- Cryptographic hash functions, such as functions from the SHA family.
- The HMAC message authentication construction.
- Basic cryptographic constructions for implementing a key derivation function (KDF), pseudorandom functions, pseudorandom permutations, a pseudorandom generator, trapdoor permutations, and a random oracle.

Mid-layer protocols: This module implements more complex cryptographic primitives that are used by higher level modules.
Currently this module includes implementations of the public-key encryption schemes of ElGamal, Cramer-Shoup, and Damgard-Jurik.

Interactive Mid-layer protocols: This module includes interactive cryptographic protocols, mostly for the purpose of enabling entities in higher level modules to prove that they are behaving properly (in order to ensure security against potentially malicious behavior).
This module currently implements Sigma protocols, Zero-Knowledge proofs, and Commitment Schemes.
In particular, the following interactive mid-layer protocols are implemented:

- Sigma protocols:
  - Knowledge of a discrete log.
  - Knowledge of a Diffie-Hellman tuple (DH), and of an extended DH tuple.
  - Knowledge of a Pedersen commitment knowledge, and of a Pedersen committed value.
  - Knowledge of an El Gamal commitment knowledge, of an El Gamal committed value, of an El Gamal private. key and of an El Gamal encrypted value.
  - Knowledge of a Cramer-Shoup encrypted value.
  - Knowledge of a Damgard-Jurik encrypted zero, of a Damgard-Jurik encrypted value, and of a Damgard-Jurik product.
  - Knowledge of an AND (of multiple statements).
  - Knowledge of an OR of two statements, and of an OR of multiple statements.
- Zero Knowledge proofs:
  - A Zero Knowledge proof based on any sigma protocol
  - A Zero Knowledge proof of Knowledge based on any sigma protocol (currently implemented using Pedersen's commitment scheme)
  - Zero Knowledge proof of knowledge based on any sigma protocol, using the Fiat-Shamir heuristic (in the Random Oracle Model)

- Commitments protocols:
  - Pedersen commitment, Pedersen hash commitment, and Pedersen trapdoor commitment.
  - El Gamal commitment, and El Gamal hash commitment.
  - Simple hash commitment
  - Equivoqal commitments

OT Extension: This module includes state-of-the-art protocols for both Semi-Honest and Malicious oblivious transfer extension, based on [3, 9].

Circuits: This module includes implementations of Yao's protocol for secure two-party computation. It includes versions of the protocol for either semi-honest or malicious security, as well as for more efficient amortized performance when a batch of computations needs to be evaluated.

Communication: This module implements a communication channel that is used by all modules.

Chapter 2

Oblivious Transfer Extension

We prototypically implemented several OT extension protocols described in [10, 11] and made them available as open source projects at https://github.com/encryptogroup/OTextension. We give an outline of the architecture in Figure 2.1. The implemented OT extension protocols include:

- The passively secure protocol of [8] (short IKNP)
- The passively secure protocol of [3] (short ALSZ13)
- The passively secure protocol of [13] (short KK)
- The actively secure protocol of [17] (short NNOB)
- The actively secure protocol of [2] (short ALSZ15), which is described in Section 3.2 in D13.1 [10].

Furthermore, we have implemented several base-OT protocols, which can be exchanged dynamically:

- The passively secure protocol of [16] (short NP)
- The actively secure protocol of [18] (short PVW)
- The actively secure protocol of [6] (short CO), which is described in Section 3.1 in D13.1 [10].

The OT extension protocols can be dynamically instantiated with the different OT functionalities, outlined in Section 1.1 in D13.3 [11]. An instruction on how to set up the code as well as how to call the methods is provided on Github.

2.1 Implementation

Implementation of the passive secure OT extension protocol of [3] and the active secure OT extension protocols of [2] and [17]. Implements the general OT (G_OT), correlated OT (C_OT), global correlated OT (GC_OT), sender random OT (SR_OT), and receiver random OT (RR_OT) (Definitions of the functionalities will follow). Implements the base-OTs by Naor-Pinkas [16], Peikert-Vaikuntanathan-Waters [18], and Chou-Orlandi [6]. The code is based on the OT extension implementation of [5] and uses the MIRACL libary [1] for elliptic curve arithmetic.

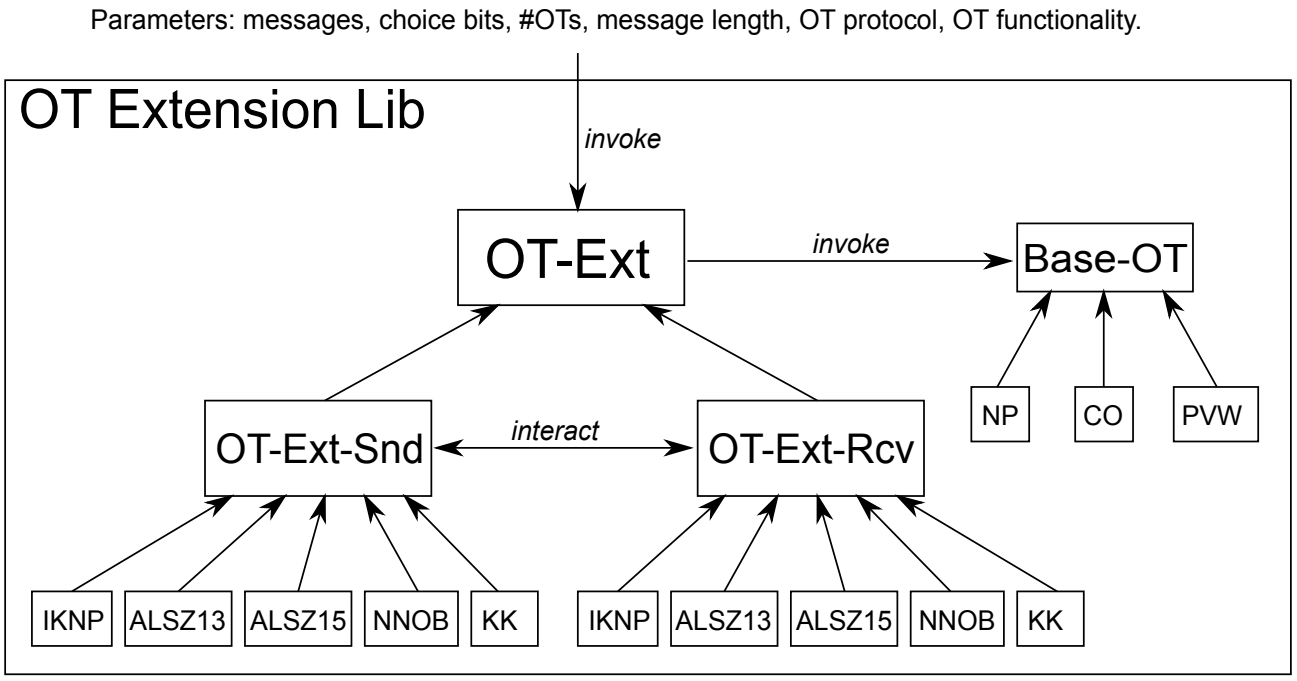

Figure 2.1: Architecture of our open source OT extension library at https://github.com/encryptogroup/OTextension. Upward arrows denote class inheritance.

Update: Implemented 1-out-of-2 OT from the 1-out-of-N OT extension of [7].
Our OT extension implementation is maintained as open source project, available at https://github.com/encryptogroup/OTextension. In the following section we detail how to deploy the code.

2.1.1 Requirements

- A **Linux distribution** of your choice (the OT extension code was developed under Ubuntu).
- **Required packages:**
  - `g++`
  - `make`
  - `libgmp-dev`
  - `libssl-dev`

  Install these packages with your favorite package manager, e.g, `sudo apt-get install package_name`.

2.1.2 Compiling the Framework

1. Clone a copy of the main OTExtension git repository and the Miracl submodule by running:
   `git clone -recursive git://github.com/encryptogroup/OTExtension`
2. Enter the Framework directory: `cd OTExtension/`

3. Call `make` in the root directory to compile all the code and create the corresponding executables.

Using the Framework

To start OT extension, open two terminals on the same PC and call
`ot.exe -r 0`
in one terminal to start OT extension as sender and call
`ot.exe -r 1`
in the second terminal to start OT extension as receiver. This will invoke the passive secure 1-out-of-2 OT extension protocol from [8] for 1 million OTs on 8-bit strings. The result of the OT will be checked for correctness and the times (in ms) for the base-OTs, for the OT extensions, the number of bytes sent and the number of bytes received will be printed on the terminals. A list of all available options can be obtained via
`ot.exe -h`.

Notes

- An example implementation of OT extension can be found in `mains/otmain.cpp`.
- OT related source code is found in `ot/`.
- Different compilation flags can be set in `util/constants.h`.

Chapter 3

Valiant's Universal Circuit Construction

We provide a prototype implementation of Valiant's efficient universal circuit construction [20] with our optimizations from [12]. The implementation is available open source at `https://github.com/encryptogroup/UC`.
The goal of implementing and usign Valiant's efficient universal circuit construction is to perform a secure computation of a function while hiding the function that is being computed. The implementation accepts any Boolean circuit as input in Secure Hardware Description Language (SHDL) format [15, 4], provided that the gates have at most two incoming edges, which can be enforced by using FairplayPF to translate a functionality [14]. It outputs the topology of the UC along with its programming bits corresponding to the input circuit as described in deliverable D13.3 [11].
The output of our universal circuit compiler can then be used as the input to any generic secure computation framework that understands our circuit format. In deliverable D21.3, we discuss how our universal circuit compiler can be used for building applications. In this deliverable we will describe how to deploy our tool and use it to obtain a universal circuit and its programming, given a specified input circuit C.

3.1 Implementation

3.1.1 Requirements

Our UC compiler was developed in C/C++ and therefore requires `g++` using Linux. Its classes are depicted in Figure 3.1.

3.1.2 File System Structure

- `/graphviz/` - Graphviz files if graph debugging is enabled.
- `/circuits/` - Example circuit files.
- `/src/` - Source code.
  - `src/gamma/` - Source of underlying graph of the circuit.
  - `src/uc/` - Source of universal graph, its embedding and printing as universal circuit into file.

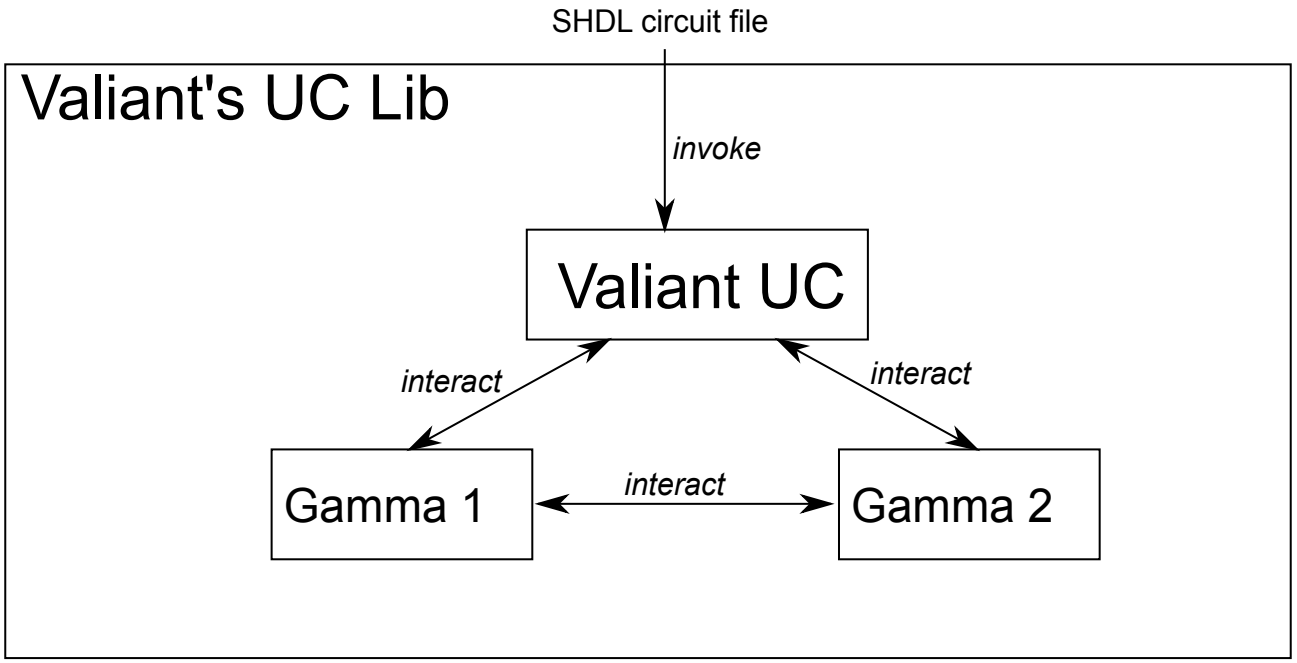

Figure 3.1: Architecture of our open source UC construction implementation at https://github.com/encryptogroup/UC.

– `src/util/` - Source of functions that are used for debugging or reading from the SHDL file.

3.2 Using the Framework

1. Clone a copy of the main UC git repository and its submodules by running:
   `git clone -recursive git://github.com/encryptogroup/UC`
2. Enter the Framework directory: `cd UC/src/`

Testing Circuits

- Our UC compiler is compatible with SHDL format with gates with at most two inputs, i.e., any function generated with the FairplayPF [14] circuit compiler. Examples such as `MobileCode` and `CreditChecking` can be found under `/circuits/`. A circuit described in SHDL format is as follows:

```
0 input
1 input
⋮
u input
⋮
i (output) gate arity 2 table [ * * * * ] inputs [ * * ]
j (output) gate arity 1 table [ * * ] inputs [ * ]
⋮
```

  where any gate can potentially be an output gate and can have 1 or 2 input wires (arity 1 or 2), which are defined at `inputs [ * * ]`. The gates can perform any gate functionality

that is defined using 4 bits for 2 inputs `table [ * * * * ]` and 2 bits for 1 input `table [ * * ]`.

- Our UC compiler is also compatible with the test circuits for basic function from [19], having to use an additional conversion function that translates these to SHDL circuit description as described below.
- If you are using the format of the circuits from [19], you add your circuit under `/circuits/` and set the name of your circuit, e.g., `circuit_name`, within the first parameter of the main function of `bristol_to_SHDL.cpp` and run:
  `g++ -o Bristol bristol_to_SHDL.cpp`
  `./Bristol`
- If you generated your circuit file using FairplayPF [14] or you have completed the previous step, next the fanout-2 gates are eliminated and the UC and its programming are generated using `UC.cpp`. For generating and testing the UC, set the name of your circuit in the main of `UC.cpp` and run: `g++ -o UC UC.cpp`
  `./UC`
- On successful run, two files are generated under `/circuits/`:
  `circuit_name_circ.txt` with the UC topology and `circuit_name_prog.txt` with its programming corresponding to the input circuit. The UC topology is described as a series of inputs, gates and outputs in the following way:
  `C 0 1 ...u` enumerates the client's input wires.
  `X i j k l`
  `Y i j k`
  `U i j k`
  `O x ...z` enumerates the output wires.

  The gates in the middle occur according to the topology of Valiant's construction. An X switching gate described in [12] and in deliverable D13.3 [11] with `i, j` input wires and `k, l` output wires. A Y block and a universal gate U receive two inputs as before but output only one value `k`.
- Some debug output is provided to verify the correctness of the computation.

Chapter 4

Summary

We described prototype implementations of key secure multi-party computation protocols, and of tools for implementing secure multi-party computation protocols. The implementations are all academic prototypes that are available as open source.
The implementations include the LibSCPAI C++ software library for secure computation, implementations of oblivious transfer extension protocols, and an implementation of Valiant's universal circuit construction.

Chapter 5

List of Abbreviations

OT	Oblivious Transfer
UC	Universal Circuit
PFE	Private Function Evaluation

Bibliography

[1] Certivox, multiprecision integer and rational arithmetic cryptographic library (miracl). https://github.com/CertiVox/MIRACL.

[2] G. Asharov, Y. Lindell, T. Schneider, and M. Zohner. More efficient oblivious transfer extensions with security for malicious adversaries. In *Advances in Cryptology – EUROCRYPT'15*, volume 9056 of *LNCS*, pages 673–701. Springer, 2015. Full version: http://eprint.iacr.org/2015/061.

[3] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. More efficient oblivious transfer and extensions for faster secure computation. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 535–548. ACM, 2013.

[4] Assaf Ben-David, Noam Nisan, and Benny Pinkas. FairplayMP: a system for secure multi-party computation. In *ACM CCS'08*, pages 257–266. ACM, 2008.

[5] Seung Geol Choi, Kyung-Wook Hwang, Jonathan Katz, Tal Malkin, and Dan Rubenstein. Secure multi-party computation of boolean circuits with applications to privacy in on-line marketplaces. In *Topics in Cryptology - CT-RSA 2012*, volume 7178 of *LNCS*, pages 416–432. Springer, 2012.

[6] T. Chou and C. Orlandi. The simplest protocol for oblivious transfer. In *Progress in Cryptology – LATINCRYPT'15*, volume 9230 of *LNCS*, pages 40–58. Springer, 2015.

[7] Daniel Demmler, Thomas Schneider, and Michael Zohner. ABY - A framework for efficient mixed-protocol secure two-party computation. In *Network and Distributed System Security Symposium, NDSS'15*, 2015.

[8] Y. Ishai, J. Kilian, K. Nissim, and E. Petrank. Extending oblivious transfers efficiently. In *Advances in Cryptology – CRYPTO'03*, volume 2729 of *LNCS*, pages 145–161. Springer, 2003.

[9] Marcel Keller, Emmanuela Orsini, and Peter Scholl. Actively secure OT extension with optimal overhead. In Rosario Gennaro and Matthew Robshaw, editors, *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part I*, volume 9215 of *Lecture Notes in Computer Science*, pages 724–741. Springer, 2015.

[10] Florian Kerschbaum, Florian Hahn, , Benny Pinkas, 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.

[11] Florian Kerschbaum, Florian Hahn, Benny Pinkas, Thomas Schneider, Michael Zohner, Pille Pullonen, and Claudio Orlandi. PRACTICE Deliverable D13.3: the complete set of new protocols, 2016. Available from http://www.practice-project.eu.

[12] Ágnes Kiss and Thomas Schneider. Valiant's universal circuit is practical. In *EUROCRYPT (1)*, volume 9665 of *Lecture Notes in Computer Science*, pages 699–728. Springer, 2016.

[13] V. Kolesnikov and R. Kumaresan. Improved OT extension for transferring short secrets. In *Advances in Cryptology – CRYPTO'13*, volume 8043 of *LNCS*, pages 54–70. Springer, 2013.

[14] Vladimir Kolesnikov and Thomas Schneider. A practical universal circuit construction and secure evaluation of private functions. In *fc 08)*, volume 5143 of *LNCS*, pages 83–97. Springer, 2008. Code: http://encrypto.de/code/FairplayPF.

[15] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay - secure two-party computation system. In *USENIX Security Symposium 2004*, pages 287–302. USENIX, 2004.

[16] M. Naor and B. Pinkas. Efficient oblivious transfer protocols. In *Symposium on Discrete Algorithms (SODA'01)*, pages 448–457. ACM/SIAM, 2001.

[17] J. B. Nielsen, P. S. Nordholt, C. Orlandi, and S. S. Burra. A new approach to practical active-secure two-party computation. In *Advances in Cryptology – CRYPTO'12*, volume 7417 of *LNCS*, pages 681–700. Springer, 2012.

[18] C. Peikert, V. Vaikuntanathan, and B. Waters. A framework for efficient and composable oblivious transfer. In *Advances in Cryptology – CRYPTO'08*, volume 5157 of *LNCS*, pages 554–571. Springer, 2008.

[19] Stefan Tillich and Nigel Smart. Circuits of basic functions suitable for MPC and FHE, 2015.
http://www.cs.bris.ac.uk/Research/CryptographySecurity/MPC/.

[20] Leslie G. Valiant. Universal circuits (preliminary report). In *ACM Symposium on Theory of Computing (STOC'76)*, pages 196–203. ACM, 1976.