DS-06-2017: Cybersecurity PPP: Cryptography

PRIViLEDGE
Privacy-Enhancing Cryptography in Distributed Ledgers

D4.5 – Report on Tools for Privacy-Preserving Applications on Ledgers

Due date of deliverable: 30 June 2021
Actual submission date: 29 June 2021

Grant agreement number: 780477
Start date of project: 1 January 2018
Revision 1.0

Lead contractor: Guardtime AS
Duration: 36 months

Project funded by the European Commission within the EU Framework Programme for Research and Innovation HORIZON 2020

Dissemination Level	
PU = Public, fully open	X
CO = Confidential, restricted under conditions set out in the Grant Agreement	
CI = Classified, information as referred to in Commission Decision 2001/844/EC	

D4.5

Report on Tools for Privacy-Preserving Applications on Ledgers

Editor
Ahto Truu (GT)

Contributors
Maria Iliadi (GRNET)
Foteinos Mergoupis-Anagnou (GRNET)
Sven Heiberg (SCCEIV)
Berry Schoenmakers (TUE)
Markulf Kohlweiss (UEDIN)
Daniele Friolo (UNISA)
Ivan Visconti (UNISA)

Reviewers
Panos Louridas (GRNET)
Marko Vukolic (IBM)

29 June 2021
Revision 1.0

The work described in this document has been conducted within the project PRIViLEDGE, started in January 2018. This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 780477.

Executive Summary

This document presents the final report on three toolkits and three use-case application prototypes that have emerged from the PRIViLEDGE project. The toolkits include a production-grade implementation of the Snarky Ceremonies protocol for zero-knowledge proofs, a toolkit adding ledger-based communication support and verifiability to multi-party computations, and a toolkit to support transparent access to both on-ledger and off-ledger data from applications. The use-case prototypes include a verifiable privacy-preserving voting system, verifiable privacy-preserving reports for health data, and privacy-preserving verification of academic degree documents.

Contents

1 Introduction **1**

2 Toolkits **2**
2.1 Toolkit for Zero-Knowledge Proofs . . . 2
2.1.1 Overview . . . 2
2.1.2 Components . . . 2
2.1.3 Optimizations . . . 3
2.1.4 Security Considerations . . . 3
2.1.5 Availability and Licensing . . . 3
2.2 Toolkit for Multi-Party Computation . . . 3
2.2.1 The Ledger Proxy . . . 4
2.2.2 The Generic Proxy Component . . . 4
2.2.3 The Ledger Component . . . 4
2.2.4 Recommended Features . . . 8
2.2.5 MPyC Framework . . . 8
2.2.6 Example Applications . . . 10
2.2.7 Verifiability Enhancements . . . 10
2.2.8 Availability and Licensing . . . 10
2.3 Toolkit for Data Storage . . . 10
2.3.1 Overview . . . 10
2.3.2 Interface Outline . . . 11
2.3.3 The DIPLOMATA Record Ledger Use Case . . . 11
2.3.4 Availability and Licensing . . . 11

3 Use-Cases **12**
3.1 Prototype Application for i-Voting . . . 12
3.1.1 Overview . . . 12
3.1.2 Actors and Use-Cases . . . 12
3.1.3 Components . . . 13
3.1.4 Implementation . . . 13
3.1.5 Disclaimer . . . 13
3.1.6 Availability and Licensing . . . 14
3.2 Prototype Application for Health Insurance . . . 14
3.2.1 Overview . . . 14
3.2.2 Architecture . . . 15
3.2.3 Availability and Licensing . . . 15
3.3 Prototype Application for Diplomata . . . 15
3.3.1 Overview . . . 15
3.3.2 Frontend Server . . . 16

3.3.3 API Server 17
3.3.4 Cryptographic Library 18
3.3.5 Ledger Library 18
3.3.6 Availability and Licensing 18

4 Summary **19**

Chapter 1

Introduction

This document presents the final report on three toolkits and three use-case application prototypes that have emerged from the PRIViLEDGE project. Please see [PRI20b] and [PRI20c] for earlier iterations of the report.

Section 2.1 introduces a production-grade implementation of the Snarky Ceremonies protocol in the Rust programming language. The new implementation includes a number of significant performance and security enhancements over the reference implementation that has been available previously.

Section 2.2 reports on a toolkit to support multi-party computations. The toolkit consists of two modules. The first one provides a generic mechanism for parties in a two-party of multi-party computation to use a ledger as a communications channel instead of point-to-point connections. This can be useful for providing an audit trace of the protocol execution or enabling multi-party computations among parties who are not guaranteed to be online simultaneously. The second module adds support for verifiable MPC to the existing multi-party computation framework MPyC. The module also includes an implementation of a secure group scheme that could in principle allow secret-sharing based secure computations over any finite group.

Section 2.3 presents a toolkit to support access to both on-ledger and off-ledger data in a fashion that is transparent to the application accessing the data.

Section 3.1 describes a prototype implementation of the Tiviledge voting system. The system uses a public bulletin board to allow independent auditors to verify that all accepted votes were correctly stored, forwarded, and tabulated. The audit mechanism also protects voter privacy.

Section 3.2 outlines a proof-of-concept implementation of verifiable multi-party computation to support outcome-based contracts in health insurance. The application supports verifiable reporting on patient responses to treatments, to allow performance-based pricing of medications without exposing private data of the patients to the pharmaceutical companies.

Section 3.3 introduces DIPLOMATA, a system for verifying academic degrees in a privacy-preserving manner. A holder of a diploma can request the issuer to certify the diploma for a specific recipient, and only the designated party can verify the diploma.

The toolkits and applications reported on in this document are relevant mostly to the applications layer of the distributed ledger technology stack. Please see [PRI21c] for report on the PRIViLEDGE software related to the underlying ledger systems themselves.

Chapter 2

Toolkits

2.1 Toolkit for Zero-Knowledge Proofs

2.1.1 Overview

SNARKY is a Rust implementation of the Snarky Ceremonies protocol [KMSV21] over the BLS12-381 elliptic curve. Its intention is not to provide a reference implementation (a previous implementation in Python[1] serves better for this purpose), but to be the first experimental version of a production-grade library for running the protocol in real-life applications. As such, the library's source code deviates from the sequential logic (and also the notation) of the protocol as presented in the original publications.. This is, roughly speaking, due to the following needs:

(a) Parallelization of loops (speed): Loops should be appropriately adjusted, so that tools from the Rust ecosystem (mainly rayon) become applicable.

(b) Constant time operations (security): In order to mitigate timing attacks, constant time operations were applied wherever possible.

Correctness of code is guaranteed by tests, originally written against a separate implementation closely following the paper (without any optimizations). That does not obviate the need for a thorough security review from a third party.

2.1.2 Components

The library consists of the following self-contained subpackages (Rust crates).

util: Provides various helpers for internal usage (parsing, errors, random generation etc.).

backend: This is a wrapper around the `https://github.com/arkworks-rs/curves/tree/master/bls12_381` bilinear-pairing library. For the purpose of rapid development, the selected wrapping mechanism consists entirely of Rust macros. Refactoring with structs and traits can make it possible to parametrize the bilinear-pairing upon instantiation, that is, choose different curves from the arkworks ecosystem.

polynomials: This is a thin wrapper around the polynomial infrastructure provided by arkworks, intended to adapt it for the needs of the **circuits** subpackage.

circuits: Provides infrastructure for quadratic arithmetic programs (QAPs). Its main purpose is to guard against the creation of QAPs with incompatible dimensions. The underlying data structure is currently a collection of Rust vectors containing polynomials. The usage of available crates for building circuits generically over a scalar field type should be examined as a possibility.

[1] `https://github.com/grnet/snarky-prototype`

protocol: The transaction layer of the protocol, exposing also the library's public API. Provides the updatable SRS infrastructure along with a proof generation/verification mechanism for operating on it.

2.1.3 Optimizations

Originally, `https://github.com/zkcrypto/bls12_381` was used as the bilinear-pairing backend. Although this library is mature and maintained by the active ZK community of Rust, `https://github.com/arkworks-rs/curves/tree/master/bls12_381` turned out to be significantly faster with respect to pairing, scalar exponentiation and the G2-group operation. Plugging it along with the polynomial infrastructure of the arkworks ecosystem improved performance roughly as follows:

- phase 1 SRS update: 13 times faster;
- phase 2 SRS update: 3 times faster;
- verification: 2 times faster (referring to the part which is independent of the number of updates).

It should be stressed that the arkworks is a work in progress, so an eye should be kept on it for further optimizations.

A further performance boost was given by a standard application of data-parallelism (rayon), which dynamically adapts the workload of iterators, so that maximum performance is achieved under account of runtime environment:

- phase 1 SRS update: 2 times faster;
- phase 2 SRS update: 2 times faster;
- verification: 2 times faster.

Finally, verification became 4 times faster after applying the technique of "batching" for verifying a large number of bilinear-pairing equations.

2.1.4 Security Considerations

Unsafe Rust is nowhere used for the moment, so that memory-safety is generally guaranteed by the borrow-checker, but this might change in the future for the sake of further optimizations.

Considerable effort has been spent to mitigate timing attacks by applying constant-time operations, which included elimination of short-circuit evaluations and early returns. Note however that this leaves out the backend; in particular, no constant-time check has been implemented for equality between algebraic objects, assuming instead that this should be provided by the pairing library itself.

2.1.5 Availability and Licensing

The library is available at `https://github.com/grnet/snarky` and licensed under AGPLv3.

2.2 Toolkit for Multi-Party Computation

The toolkit for ledger-oriented two/multi-party computation focuses on upgrading known libraries for two/multi-party computation so that they can leverage features of ledgers. The toolkit consists of two parts: 1) the first module generically allows libraries from two/multi-party computation to use a ledger as a communication channel instead of point-to-point connections; 2) the second module is an upgrade of MPyC to add the possibility of verifying computations.

2.2.1 The Ledger Proxy

This component provides a proxy between an MPC protocol library and a ledger. Messages from the MPC library are automatically posted to the ledger by the proxy, and when a new message is posted on the ledger from another party, the proxy forwards it to the MPC library. The MPC library remains untouched, and thus the feature of redirecting the communication through the ledger is transparent. The main requirement to make this possible consists of properly setting the IP addresses and port numbers typically used when configuring the library with classical TCP connections so that instead of connecting the parties directly, each party is connected to the proxy and the proxy can capture the messages sent through such TCP channels. The proxy encapsulates messages sent to each of these ports as ledger transactions and broadcasts them to the network by contacting the corresponding peers.

The proxy is designed to be generic also in the way it uses the ledger, and thus it is composed of two main parts: the generic proxy component and the specific ledger component. The proxy component takes care of the communication with the MPC library, and the ledger component is an interface that can be implemented in multiple ways depending on the desired ledger. We have successfully configured the proxy so that it works with:

- MPyC, an MPC library tolerating a dishonest minority of passively corrupt parties.
- EMP-ag2pc, a two-party computation library based on authenticated garbling and achieving efficient maliciously secure two-party computation.
- An ad-hoc natural multi-party coin tossing protocol tolerating a dishonest majority of actively corrupt parties.

We implemented the ledger component to work with Ethereum and Hyperledger Fabric. A more detailed discussion of all components follows.

2.2.2 The Generic Proxy Component

The generic proxy component communicates with the MPC libraries by relying on TCP sockets, needed to emulate communication channels between the party P_i, impersonated by the library, and all the other parties $P_{j\neq i}$ involved in the MPC protocol.

The proxy, for an n-party computation, creates $n-1$ sub-instances. Each sub-instance emulates the connection between the local player P_i and a remote player P_j.

On the MPC library side, each sub-instance (i, j) opens a TCP socket on a specific port to listen to messages coming from the MPC library (from P_i to P_j), and another socket connected to a different port to write messages to the MPC library (from P_j to P_i).

On the ledger side, the proxy keeps polling the ledger component to receive messages from P_j to P_i, which it will forward to the TCP socket connected to the port for writing to the MPC library. When the MPC library sends a new message destined to P_j to the TPC socket connected to the listening port, the proxy triggers the ledger component's writing function to encapsulate and post such message.

Due to differences between ledgers, a setup phase is required before running the protocol. At the very beginning, for each instance, the proxy triggers a setup function provided by the ledger component, setting the channel (i, j) and configuring the ledger libraries.

2.2.3 The Ledger Component

The ledger component's interface consists of the following functions (since all the components are implemented with Java, we will use the Java notation):

- `setUp(T setupData, byte[] sid, int localParty, int remoteParty)`
  Must be run by the proxy at the beginning of the execution. It takes as inputs the protocol session ID,

the local party that the library is impersonating, and the remote party he wishes to communicate (P_i is the local party in our example). To keep the most generic approach, the value `setupData` can be instantiated with any possible object (defined in Java with the generic type `T`). As we will see, in our implementations, we will usually use a string type indicating the path of the ledger configuration file.

- `readData(int msgNum, boolean quickness)`
  Takes as inputs a message number and a boolean value indicating whether the protocol execution must be done in quick mode or not, and outputs the message with index `msgNum` meant for the player P_i and coming from P_j.

- `writeData(byte[] msg, int msgNum)`
  Takes as inputs a message and a message number, and writes such message to the ledger by putting P_j as a receiver.[2]

We realized the ledger component implementing the interface above with a dummy implementation, an Ethereum implementation, and a Hyperledger Fabric implementation. In the following, we will give a brief description of how we implemented our interfaces.

Dummy Blockchain. We implemented a dummy ledger that naively emulates the ledger functions. To keep the component simple, the ledger's persistence and liveness properties are simulated by the server by storing the sequence of messages in a local file. The dummy ledger's configuration file contains the IP address and the ports needed to communicate with the dummy ledger server. The functions are realized in the following way:

- `setUp(String configFile, byte[] sid, int sender, int receiver)`
  Stores the session ID, the sender ID and the receiver ID, and instantiates a TCP socket needed to communicate with a dummy server.

- `readData(int msgNum, boolean quickness)`
  Reads from the dummy server the message with index `msgNum` via the TCP socket instantiated when the proxy has triggered the `setUp` function. Due to the instant finalization of the dummy ledger messages, the function ignores the boolean `quickness`.

- `writeData(byte[] msg, int msgNum)`
  Sends to the server the message `msg` with index `msgNum`. The server stores the pair $(\texttt{msg}, \texttt{msgNum})$ in a file working in append-mode.

Ethereum Blockchain. We realized the Ethereum component with the aid of the web3j Java library. The library offers an automated interface to communicate with the Ethereum peers through JSON RPC APIs. Such APIs are provided by most of the commonly known services: to test our protocols in the Ethereum Ropsten testnet, we relied on the Infura network drivers.[3]

Before starting the protocol execution, each player should own an Ethereum address and the corresponding secret key. Moreover, a smart contract with the following functions should be deployed on the network:

- `readMessage`
  Takes as inputs the session ID, the sender ID, the receiver ID, and the message number. It outputs the message and the block in which the message is stored.[4]

[2] Note that the value `msgNum` does not correspond to the entire round message, but to an index assigned by the proxy based on the internal logic of the message management (e.g. in TCP the socket could receive a round message in multiple packets).

[3] Check `https://infura.io/` for more info.

[4] The block number can be inferred by properly modeling the RPC query destined to the Ethereum peer. To avoid a considerable increase in the code complexity of the Ethereum component, we ask the contract function to output the block together with the message. Since this function serves only as a reading interface, querying such a function is equivalent to asking the peer to read the ledger and extract the state information.

- `writeMessage`
  Takes as inputs the session ID, the message, the sender ID, the receiver ID, and the message number. This function must store the message in the ledger such that the `readMessage` function can correctly retrieve the message when the smart contract state changes.

- `setParticipants`
  Takes an array of addresses and a session ID as inputs. This function sets and stores the authorized set of parties for a specific session ID. After storing the set of participants, the `writeMessage` function must check if the party who triggered the `writeMessage` function corresponds to the wallet address with the correct index in the array, i.e., if `writeMessage` is triggered by P_i (i is the sender's index), then the wallet address should be the one in position i of the array stored in the contract.

The contract code is not relevant as long as the contract functions correctly.[5] As we will describe below, before starting the execution, the proxy must feed the ledger component setup function with a JSON format configuration file containing the following parameters:

- the local player wallet secret key;
- the JSON API RPC peer's URL (the Infura driver's address in our setting);
- a value indicating the maximum transaction size allowed by the network;
- the confirmation time in terms of number of blocks;
- the address of the contract that has been deployed on the network;
- an array containing the ordered list of the participants' address, i.e. each position corresponds to the player's index.

We implemented the interface functions as follows:

- `setUp(String configFile, byte[] sid, int localParty, int remoteParty)`
  Reads the JSON RPC URL of the peer and starts the communication with the peer. A pre-determined player (the one with the ID 1) triggers the contract function `setParticipants` to link the set of addresses written in the configuration file with the session ID `sid`.

- `readData(int msgNum, boolean quickness)`
  Triggers the contract function `readMessage` described above giving as inputs the session ID, the value `remote party` as the contract sender, the value `local party` as the contract receiver, and the message number `msgNum`. If `quickness` is set to `false`, waits the minimum number of blocks that are necessary to ensure confirmation of the message, and then re-reads the message by triggering `readMessage`. If `quickness` is set to `true`, it takes the message number of all the messages present in the unconfirmed bucket, i.e. a locally updated bucket of messages that are not still confirmed in the ledger, and re-reads such messages in the ledger. If at least one of such messages is different, meaning that an adversary tried to exploit forks to send different messages, it raises an exception, provoking a protocol abort. Otherwise, it adds the message with index `msgNum` to the unconfirmed bucket and flushes the bucket by removing the confirmed ones.

- `writeData(byte[] msg, int msgNum)`
  Triggers the smart contract function `writeMessage` giving as inputs the values session ID, the sender as the local party, the receiver as the remote party, the message, and the corresponding message number.

[5] As a working example, we deployed a smart contract in the Ropsten testnet network at `https://ropsten.etherscan.io/address/0x4C50a188d772F1Fade9b2892A3070c9818037528#code`.

Hyperledger Fabric. Since Hyperledger Fabric (HLF), similarly to Ethereum, relies on smart contracts (named chaincodes in HLF terminology), the implementation of the interface follows a similar route. The configuration file is a JSON containing the following fields:

- the path of the wallet of the user;
- the user ID (integer from 1 to N);
- the name of the channel;
- the name of the chaincode;
- the path of the network configuration YAML file;
- for user 1, a JSON array containing the MSP ID and the user ID of each involved party; the parties have to be listed in ascending order of user ID.

We provide a default chaincode implementing the following functions:

- `createSession`
  Similarly to the Ethereum `setParticipants` function, assigns the identities given as JSON input to a particular session ID. This function will be triggered during the setup phase by the player with ID 1.
- `createMessage`
  Takes as inputs the session ID, the message, the sender ID, the receiver ID, and the message number. This function stores the message in the ledger such that the `queryMessage` function can correctly retrieve the message when the smart contract state changes in the ledger.
- `queryMessage`
  Takes as inputs the session ID, the sender ID, the receiver ID, and the message number. It outputs the message formatted as a Base64 string.[6]

As in Ethereum, the contract code is not relevant as long as the functions correctly perform their tasks. The HLF Java library communicates with the peers via JSON RPC APIs, using the Java HLF SDK. The interface is implemented as follows:

- `setUp(String configFile, byte[] sid, int localParty, int remoteParty)`
  Instantiates the communication with the remote party over the ledger according to the information contained in the configuration file. A pre-determined player (the one with the ID 1) triggers the contract function `CreateSession` to link the set of identities written in the configuration file with the session ID `sid`.
- `readData(int msgNum, boolean quickness)`
  Triggers the contract function `QueryMessage` described above giving as inputs the session ID, the value `remoteParty` as the contract sender, the value `localParty` as the contract receiver, and the message number `msgNum`. Due to the instant confirmation of HLF transactions, the `quickness` parameter is ignored.
- `writeData(byte[] msg, int msgNum)`
  Triggers the smart contract function `CreateMessage` giving as inputs the session ID, sender as the local party, receiver as the remote party, the message and the corresponding message number.

[6] Unfortunately, the Java HLF SDK cannot manipulate raw byte arrays as inputs to smart contracts.

2.2.4 Recommended Features

For production deployment, the toolkit would benefit from the following improvements.

Restart. The toolkit's current version requires the proxy and the MPC library to be always online to complete the protocol execution successfully. By accurately tweaking the message encapsulation mechanism adopted by the proxy, the need to be always online can be removed. In a toolkit with restart, a player can shut down the protocol at any time and restart the execution from that point later on. Indeed, the ledger's persistence and liveness ensure that such messages will never be removed from the ledger.

Moreover, to enable also the communication with MPC libraries that do not natively support the restart functionality, the toolkit can emulate it by restarting the MPC protocol from scratch using the same randomness of the past execution and feeding the latter with all the messages exchanged before (they can be fetched directly from the ledger) to reach the same point of the last execution.

Broadcast. By introducing a special player (say P_0), the toolkit can be optimized for MPC protocols relying on broadcast. By tweaking the proxy messages' encapsulation mechanism and introducing some modification on how the smart contracts deal with the communication channels between parties, the broadcast functionality can be supported. As in the standard execution, the proxy instantiates $n-1$ sub-instances, where each of them indicates a channel from P_i to P_j. Since in the broadcast setting a single message from P_i is destined to all the other players $P_{j\neq i}$, P_i sends a single message to the special party P_0 instead of all the other $n-1$ parties. When, in a broadcast sub-instance, the proxy polls for a new message from P_i, this triggers a special broadcast function that queries the smart contract for messages sent by P_i to the special player P_0. In such a case, the number of messages stored in the ledger drops from $n-1$ to 1 for each round and each communication channel.

2.2.5 MPyC Framework

MPyC is a secure multi-party computation (MPC) framework implemented in Python.

The proxy component described above has been tested with the MPyC. In addition, we have also developed enhancements of MPyC to support verifiable MPC. These verifiability enhancements complement other enhancements of the basic MPyC framework that have also been developed during the course of the project.

The first public release of MPyC was version v0.3 launched on May 30, 2018 during the TPMPC workshop, followed by four major updates made available via the PyPI package repository:

- version 0.3, June 1, 2018;
- version 0.4, October 1, 2018;
- version 0.5, March 10, 2019;
- version 0.6, December 31, 2019;
- version 0.7, October 31, 2020.

The security model currently supported by the MPyC runtime assumes an honest majority, tolerating passive corruptions. In this security model, the implemented protocols in principle provide information-theoretic security. The exception here is due to the use of pseudorandom secret sharing (PRSS) in some of the protocols. The use of PRSS is however not essential and can be avoided at the cost of increased interaction between the parties for some of the basic protocols. As long as the PRF used in the implementation for PRSS is quantum secure, the overall MPyC protocols are in fact quantum secure as well.

The verifiability enhancements prevent actively corrupt parties from compromising the correctness of MPyC computations. We have implemented these enhancements using classically secure zero-knowledge proofs (based

on the discrete log assumption). If these zero-knowledge proofs are replaced by their post-quantum counterparts, the overall MPyC protocols including the verifiability enhancements will be quantum secure.

MPyC is implemented as a Python package, currently consisting of these 11 modules (all in pure Python):

1. gmpy: some basic number theoretic algorithms (using GMP via Python package gmpy2, if installed);

2. gfpx: polynomial arithmetic over arbitrary prime fields;

3. finfields: arbitrary finite fields, including binary fields and prime fields;

4. thresha: threshold Shamir (and also pseudorandom) secret sharing;

5. sectypes: types SecInt/Fxp/Fld/Flt for secret-shared integer/fixed-point/finite-field/floating-point values;

6. asyncoro: asynchronous communication and computation of secret-shared values;

7. runtime: core MPC protocols (mostly hidden by Python's operator overloading);

8. seclists: secure lists with oblivious access and updates;

9. mpctools: reduce and accumulate with log round complexity;

10. random: securely mimicking Python's random module;

11. statistics: securely mimicking Python's statistics module.

The modules are listed in topological order w.r.t. internal dependencies. Modules 1–3 are basic modules which can also be used outside an MPC context. Modules 1–8 form the core of MPyC. Modules 9–11 are small libraries on top of the core.

Unlike many other MPC frameworks that restrict the use of Shamir secret sharing to the case of prime fields, the MPyC framework actually supports finite fields of arbitrary order. Hence, next to prime fields, also binary fields and arbitrary extension fields are supported.

The use of Shamir secret sharing over finite fields, however, remains entirely hidden at the application level through the use of rich range of secure types. The secure types currently supported are SecInt, SecFxp, SecFld, and SecFlt. To implement the verifiability enhancements we also support SecGrp (for secure group operations, e.g., for the group of points on an elliptic curve). Due to use of Python's operator overloading mechanism, a rich set of arithmetic and relational operations can be used with almost no effort. The underlying protocols are implemented efficiently to yield satisfactory performance for many applications.

For the secure types SecInt and SecFxp (and SecFlt because this is built from SecInt and SecFxp) a sufficiently large prime field is used to implement Shamir secret sharing and the required multiparty protocols. For the secure SecFld the field used for Shamir secret sharing will be identical to the secure field, except when the order of the secure field is too small compared to the number of parties (in which case we need to do Shamir secret sharing over a sufficiently large extension of the desired secure field).

The secure types are parametrized with the size of the (number) range demanded. This allows for secure numbers of arbitrary size, which can be tailored to any application, and moreover, can be set at runtime (even depending on the sizes of the inputs and datasets to be processed).

Next to the basic arithmetic and relational operations available for these secure types, an extensive API mimicking many well-known Python primitives is implemented. Examples in this category are functions like `all()`, `any()`, `min()`, `max()`, `sum()`, `prod()`, `sorted()`, `argmin()`, `argmax()`. More specific support is included in the modules 9–11 listed above. The protocols have been implemented carefully to provide adequate performance for these functions.

The MPyC framework internally uses a quite advanced approach to provide synchronous communication between the parties. The most visible functions in this respect are input() and output() for handling secret-shared values (MPyC secure types). Also, there is the function transfer() which can be used for exchanging public values (arbitrary Python types).

2.2.6 Example Applications

A range of demos has been included in the MPyC repository on GitHub, of which several are accompanied by research papers. In addition, more code and applications have been developed in other contexts.

Most notably, the demo `ridgeregression.py` implements secure ridge regression, which led to follow-up work on the secure Moore–Penrose pseudoinverse. The demo `bnnmnist.py` builds on an alternative protocol for secure comparison based on the Legendre symbol, and demonstrates a binarized neural network. This provides a faster alternative for the demo `cnnmnist.py` which relies on a more standard type of secure comparison to build a convolutional neural network. The demos `lpsolver.py` and `lpsolverfxp.py` implement a secure version of the Simplex algorithm for linear programming; this demo is particularly suited for verifiable MPC as verification can be done via the solution to the dual problem. The demos `aes.py` and `onewayhashchains.py` implement a threshold version of Lamport's identification scheme—here verification comes for free as the preimages on a hash chain can be verified publicly. Further applications included are `id3gini.py` for generating ID3 decision trees, and `kmsurvival.py` for performing Kaplan-Meier survival analysis, both in a privacy-preserving manner of course.

Common to all these demos is that the Python code is rather compact and relatively simple given the methods implemented. Algorithmic variations can be investigated easily, to see if these have any impact on the performance.

2.2.7 Verifiability Enhancements

The main ingredient for verifiability support is the generation of zero-knowledge proofs in MPyC. Since MPyC can also be run with a single party, verifiable MPyC thus constitutes a true generalization of zero-knowledge proofs. The generalization comprises the replacement of a single prover by a set of $m \geq 1$ provers where at most $t < m/2$ provers may be corrupt.

To expedite the development of verifiable MPyC we have designed and implemented an extensive secure group scheme. The secure group scheme lets us operate easily and efficiently on secret-shared group elements. In principle, any finite group can be made secure this way. We have included simple examples like small permutation groups S_n, but we mostly focus on groups that are used for cryptographic purposes such as elliptic curves.

In terms of these secure groups, we have implemented a range of zero-knowledge proof systems.

2.2.8 Availability and Licensing

The first module allowing two/multi-party computation libraries to use the ledger as a communication channel is available from `https://github.com/danielefriolo/ledgerMPC` under the MIT license.

The MPyC framework is available from `https://github.com/lschoe/mpyc` under the MIT license. The package can also be installed via the PyPI repository.[7] The verifiable MPC extension to MPyC is available from `https://github.com/toonsegers/verifiable_mpc` under the MIT license. It requires the secure groups extension, which is available from `https://github.com/toonsegers/sec_groups` under the MIT license.

2.3 Toolkit for Data Storage

2.3.1 Overview

The toolkit for data storage is a ledger-oriented solution for bridging data residing on a blockchain with those stored in an off-chain database. It is responsible for coordinating interaction between the backend ledger, storage,

[7] `https://pypi.org/`

and the cryptographic library of an API service and the first effort to implement the Privacy-Preserving Ledger Storage Toolkit outlined in Section 7.3 of [PRI20b]. It should be thought of as "glue" between:

i a ledger, with the property of returning some kind of transaction ID upon appending data;

ii a cryptographic engine (e.g., an asymmetric or hybrid cryptosystem, along with a corresponding PKI), the output of which can be both appended to the ledger or stored off-chain (and potentially privately);

iii an involved party's offchain-storage mentioned in (ii), capable among others of storing transaction IDs so that ledger entries can be retrievable by other parties.

By coordinating the interaction of the above components, the toolkit is essentially a generic tool for executing protocols where quantities produced at each step must be locally stored for later use and/or appended to a ledger for public verifiability. In particular, the toolkit is agnostic of the internals of the components; any ledger, cryptosystem or database should be pluggable, provided that they conform to the corresponding API enforced by the toolkit (e.g., by means of appropriate wrappers). The toolkit provides thus the unified language, under which the various components communicate independently of their peculiarities.

2.3.2 Interface Outline

Roughly speaking, this unified language consists of the following operations:

i READ/WRITE operations on the ledger: The toolkit is responsible for submitting small data (like the "tags" described in 3.3.4 and 3.3.5) produced during protocol execution to the ledger, receiving back a transaction ID for that action, and forwarding this receipt to the storage.

ii READ/WRITE operations on the storage: Auxiliary quantities produced during protocol execution, which need not necessarily be published on the ledger but are needed for later use, must be (persistently or temporarily) stored in the user's database. The ledger data, being the detached signatures of the protocol computations, are thus strongly correlated to the data residing in the user's storage, so that a reciprocal guarantee of legitimacy is achieved among them.

iii ENCRYPT/DECRYPT/COMMIT/SIGN calls to the cryptographic component: The toolkit sets the cryptosystem in motion and receives back the result of the computation, which it stores off-chain, appends to the ledger, or immediately forwards to another party.

2.3.3 The DIPLOMATA Record Ledger Use Case

In the DIPLOMATA record ledger application (Sec. 3.3), the toolkit serves as the backbone of a user's API service, ensuring that the ledger, cryptographic and storage libraries interact in a robust and safe fashion. It does so by enforcing consent and notification upon the execution of any operation that is associated with the user's data.

2.3.4 Availability and Licensing

The toolkit is available at `https://github.com/grnet/db-chain-bridge` and licensed under AGPLv3.

Chapter 3

Use-Cases

3.1 Prototype Application for i-Voting

The online voting use case has developed a prototype online voting system Tiviledge that covers all major functions for an online voting system, focusing on making election data available for independent audits of integrity under the condition of secret ballot.

3.1.1 Overview

Verifiable voting systems are typically based on distributed architectures. System state is distributed between multiple parties, integrity is achieved when a certain threshold of these parties is behaving according to the protocol. A primitive often used to underline the need for such a distributed architecture is a bulletin board that is publicly readable, with authenticated append-only messages [CGS97].

The purpose of a distributed bulletin board is integrity. Any tampering attempt must—at least—be detectable early in the process. Actual election organisers must be capable of convincing the general public that parties hosting the bulletin board are not maliciously co-operating. If the whole election infrastructure is run by a single entity—which is usually the case—then the trust into the integrity of an election comes down to the question of trust towards that single entity.

The Tiviledge online voting system uses the general-purpose permissioned blockchain Hyperledger Fabric (HLF) as the bulletin board technology, implementing the online voting auditing functionality as a smart contract (*chaincode* in HLF terminology).

Tiviledge provides secure, usable and transparent online voting by using a protocol that makes it possible, in addition to voter verifiability, to prove to an independent auditor, in a voter privacy preserving manner, that all accepted votes were stored, sent to the tabulation according to the election rules, and decrypted/tabulated correctly.

The notion of the data-audit is used to ensure that the published voting result corresponds to encrypted preferences sent by eligible voters to the digital ballot box and the bulletin board. We mitigate the need to prove correctness of the software and its operation by demonstrating that according to the public protocol, the correct election outcome was calculated based on the given public inputs. Moreover, we emphasize the importance of long-term voter privacy over the long-term integrity of the election result.

The underlying cryptographic protocol is based on verifiable homomorphic aggregation of rerandomized encrypted votes created with a commitment-consistent encryption (CCE) scheme. The protocol allows to publish rerandomized commitments to the public ledger, providing third-party auditability and receipt-freeness. Unconditionally hiding commitment scheme is used.

3.1.2 Actors and Use-Cases

Tiviledge is concerned with the following actors:

- Voters can use Tiviledge to vote and to audit the election data.
- Election organizers can set up and execute an online voting event from determining the list of eligible voters and available candidates, up to the point of calculating the final tally and publishing auditable data.
- Auditors can participate in the bulletin board (HLF blockchain) and perform all auditing calculations on the published data.

3.1.3 Components

Tiviledge prototype consists of several web applications, online services and offline applications.

- Voter application VoteApp is a web application that allows casting votes and verifying election processes. This application is used by both voters and auditors.
- Election manager ElectionManager is a web application with corresponding backend service which is responsible for supporting election management processes.
- Vote collector service VoteCollector is a backend service responsible for storing online votes. It is under the control of election organizers, and commits to the ledger for auditability.
- Pylon service Pylon is a proxy to mediate communication between all infrastructure components, including voter authentication and ledger communication.
- Blockchain network Ledger serves as a bulletin board where public records are being published.
- An offline Key application is responsible for CCE keys generation and aggregation/decryption of votes from VoteCollector.
- An offline Onekey application is used for creating pseudonymized identites for voters and administrators which are used in the Tiviledge system. It generates a public identity with an ECDSA private key and certificate for a real person.

3.1.4 Implementation

The Tiviledge backend and HLF chaincode has been implemented in the Go programming language, the voting, auditing and administrative functionality has been written in TypeScript, using the well-known Angular framework for single-page application development.

Tiviledge uses the MIRACL Core Cryptographic Library for CCE encryption implementation, rerandomization and private key generation. BLS12461 and NIST521 curves are used.

The deployment is simplified by using containerized approach with Docker. It is possible to have single-machine local test setups or fully distributed production-like deployments.

3.1.5 Disclaimer

The current Tiviledge implementation is a research prototype, its intended use is experimentation with the technology, not organizing an actual legally binding online voting event. There are following major areas of interest, important for any production-grade online voting system, that haven't received thorough attention in the development of Tiviledge prototype:

- Compatibility with legal requirements of any particular jurisdiction—the Tiviledge prototype implements 1-of-N ballot structure as a proof-of-concept, but is generalizable to M-of-N ballot-style, supports revoting and multi-channel elections. However, dedicated gap analysis is required in any instance of potential application.

- Voter eligibility verification—the Tiviledge prototype implements a simple credentials scheme, where voters authenticate and sign messages using one-time credentials that are stretched into ECDSA private keys. In practice the procedure and technology of eligibility verification is important cornerstone of election security to avoid e.g. ballot-box stuffing or breach of voter privacy and the simplistic PKI of Tiviledge is not applicable.
- Election private key protection—while implementing the Tiviledge prototype we've considered the election private key protection largely a solved problem, not requiring special attention in this project. Threshold decryption (with different dealer setups) and hardware security modules are the topics of interest here, since the file-based private keys with no access control, that we use in Tiviledge are not suitable for real elections.
- Voter verification—the Tiviledge prototype does not come with a verification application for cast-as-intended property, whereas both Benaloh-challenge and Estonian-style verification would be possible here.

3.1.6 Availability and Licensing

The Tiviledge is a research prototype and there is no roadmap to make it publicly available as such. The Tiviledge protocol may be integrated into the TIVI online voting system.

3.2 Prototype Application for Health Insurance

The goal of this prototype is to test the feasibility of verifiable multi-party computations for privacy-preserving reporting on real-world data sizes, in support of outcome-based contracting in medical insurance.

3.2.1 Overview

As detailed in [PRI20b], the application needs to support the following actors and functions:

- Healthcare providers update patient records as they interact with the patients (performing tests, diagnosing conditions, administering medications, etc.) and periodically post commitments of the current state of those records.
- Patients can review their own records and receive proofs that those views are consistent with the posted commitments. This way, the patients essentially act as auditors for the factual correctness of the medical records underlying the commitments.
- Payers (sick funds or insurers) and pharmaceutical companies can receive aggregate reports on care results, accompanied by proofs that the reports are consistent with the data under the posted commitments (the data that the patients have at least occasionally reviewed to be factually correct).

The verifiable reports can support at least two kinds of outcome-based contracting in a privacy-preserving manner. First, it is possible for the healthcare providers and payers to report on efficacy of the medications and receive rebates for the cases where the patients did not respond to treatments. According to our market research, there is already considerable demand for such performance-based pricing of medications. Second, in the longer run, it would also be possible for the healthcare providers to receive payments based on more general health outcome improvements in the population under their care.

3.2.2 Architecture

As outlined in [PRI20b], for the purpose of the multi-party computation, the patient data is modeled as a stream of events, with each event represented as a tuple of integers. In each tuple, the first value is a patient ID, the second value represents the event type, and the remaining ones are type-specific attribute values.

This is not the usual representation of medical records in the information systems of healthcare providers and therefore the prototype follows a three-layer approach:

- the source data in FHIR format, managed in an HAPI FHIR server instance for testing purposes;
- a mapping layer to read the FHIR records and map them into integer tuples;
- a verifiable MPC layer, built using the Verifiable MPC extension of the MPyC framework, as described in Section 2.2 of the current document and more extensively in Chapter 8 of [PRI21b].

More details on the requirements of the use-case and evaluation criteria are given in [PRI20a]. As reported in [PRI21a], the functionality of the Verifiable MPC package is sufficient to implement the required computations, and preliminary testing results give hope that in the next iteration of the framework and the prototype, also performance may reach levels of practical usability for some specialized markets.

3.2.3 Availability and Licensing

The application is an internal research prototype and there are no plans to make it externally available in its current form. The experience gathered from the experiments on the prototype will be used in developing future versions of products of Guardtime Health.[1]

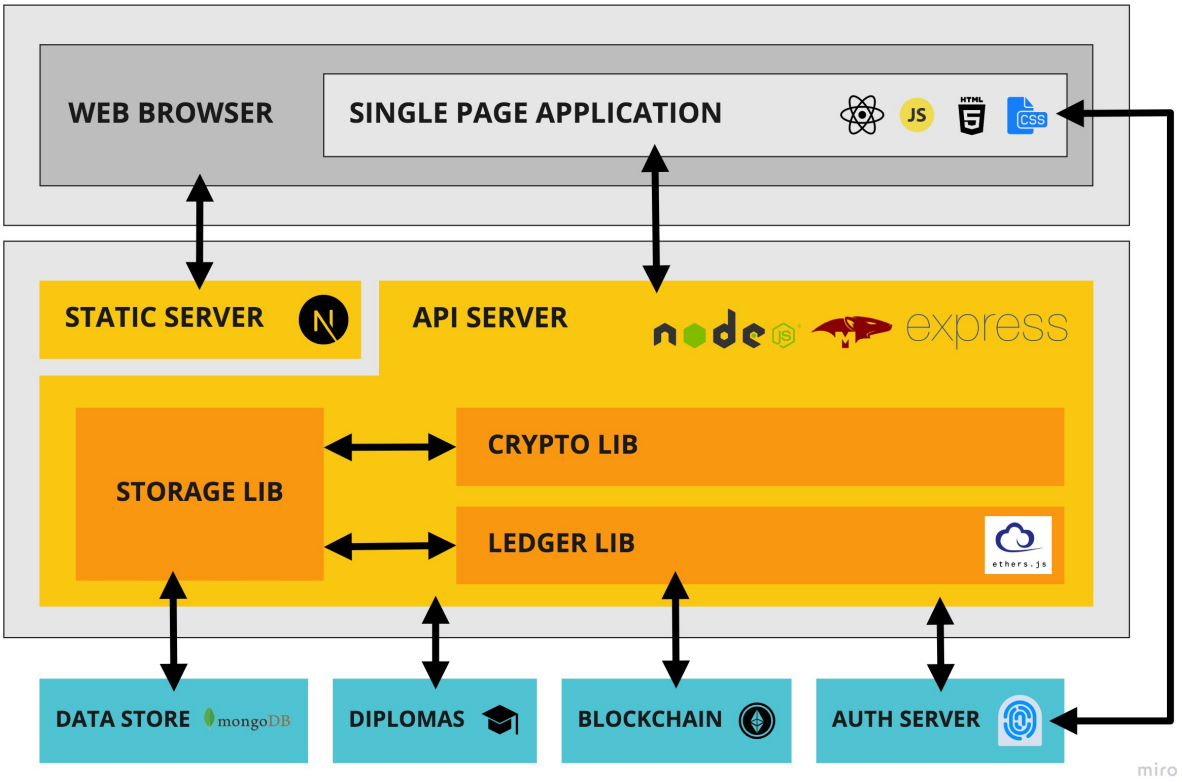
3.3 Prototype Application for Diplomata

3.3.1 Overview

The present prototype is an implementation of the University Diploma Record Ledger, or DIPLOMATA, which is outlined in Chapter 4 of [PRI20b], and conforms to the World Wide Web Consortium's Verifiable Credentials Data Model. It consists of the following components:

- traditional SPA with REST API backend;
- a separate service for each role;
- modular user authentication and diplomas registry.

[1] `https://guardtime.com/health`



e-Diplomata: Software Components Architecture

The DIPLOMATA system enables the verification of academic degrees in a privacy-preserving manner. It does so by employing a public-key cryptosystem along with a distributed ledger for immutably recording transactions, so that involved parties can be held accountable. Except for implementing the previous components, the main purpose of this prototype is to provide a service that is responsible for key management, communication between involved parties and anything else that is not covered by the cryptographic protocols per se (e.g. authentication).

In particular, a separate API service has been created for each of the protocol roles (ISSUER, HOLDER, VERIFIER). It should be stressed that, while being a solution to the Verifiable Credentials Data Model, this service guarantees enhanced privacy of the title under verification, since the title needs not be given to any third party beyond the VERIFIER (this is due to the cryptographic properties of the DIPLOMATA protocol).

3.3.2 Frontend Server

The frontend server is a Next.js server using the digigov React.js library for view components. A web application exists for each of the user types (HOLDER, ISSUER, VERIFIER). The server supports server-side rendering that allows the applications to run fast on older hardware.

Issuer Web Application

The Issuer web application is comprised of the following pages:

- Login Page
- List and Filter Issuer Diplomata Page
- View Diploma Page

In the View Diploma Page an `Award` action is available by which the process of publishing an award commitment to the ledger begins.

Holder Web Application

The Holder web application is comprised of the following pages:

- Login Page
- List and Filter Holder's Awarded Diplomata Page
- View Diploma Page

In the View Diploma Page a `Request Publication` action is available by which the process of publishing a request commitment to the ledger begins. A verifier entity must be selected to which the publication is intended.

Verifier Web Application

The Verifier web application is comprised of the following pages:

- Login Page
- List and Filter Issuer's Published Diplomata Page
- View Diploma Page

In the View Diploma Page a `Verify Diploma` action is available by which the process of verifying an acknowledge commitment to the ledger begins. A document must be uploaded for which the verification is intended.

3.3.3 API Server

The API server is comprised of an Express.js based REST API server and a MongoDB database.

Storage

This component is used for off-chain storing quantities which are produced during the execution of the cryptographic protocol and needed for later use, while also maintaining a relation between them and the tags appended to the ledger by means of transaction IDs. It is basically accessed using the storage toolkit described in the section 7.3 of [PRI20b], which serves as bridge between the ledger, the crypto component and the storage itself. The present storage is implemented with MongoDB, a fast and production-ready NoSQL database, on top of which the Mongoose ODM (Object Document Model) is used for validating data. The latter allows for conveniently splitting the server application into multiple isolation services.

REST API

The Express.js based API is implemented using the REST pattern. Exchange of off-chain data between the entities is done using the REST API endpoints and not directly using the database. This allows the server application to be split into multiple isolated services.

Interoperability

The API service is meant to interconnect with an external authentication service.

3.3.4 Cryptographic Library

In order to participate in the DIPLOMATA protocol, involved parties must have access to an El-Gamal cryptosystem. This is provided by a dedicated cryptographic library, responsible for the cryptographic primitives of the protocol, which should be thought of as one of the independent backends of the API service (along with the ledger and storage libraries), having also direct access to the user's PKI (Public Key Infrastructure). More specifically, all involved parties agree on a common underlying cryptosystem and generate corresponding keys, by means of which they can sign ledger entries, produce and verify ZK proofs, symmetrically encrypt messages etc. The security parameters of the cryptosystem are exposed in the *Cryptographic Security* section [PRI21a, PRI20b]; here we briefly give an outline of the library's architecture.

Internal Viewpoint

The library consists of three self-contained layers (from lower to higher):

- Basic cryptographic layer: the actual cryptosystem along with the corresponding prover and verifier engines, responsible for the cryptographic primitives of the protocol.
- Transaction logic layer: responsible for the protocol execution as described in the specification, leaving aside the details of the primitives.
- Presentation layer: appropriately adapts the transaction logic layer (flattening, serializations etc.), so that interfacing with other subsystems of the application becomes possible (see *External Viewpoint* below).

Each layer is as much agnostic as possible of its underlying layers' internals for the purpose of pluggability, maintenance and testing.

External Viewpoint

The cryptographic component is intended to interact with both the ledger and the user's storage.

- Interaction with the ledger: the detached signatures which are produced during protocol execution ("tags") are appended to the ledger. In turn, a transaction ID is returned, by means of which each tag is retrievable for later use.
- Interaction with the storage: some other quantities, which are produced during the protocol execution, must also be persistently stored for later use. In turn, the cryptographic component should be also able to load these quantities from the storage and adapt them appropriately, so that they become again amenable to cryptographic operations.

3.3.5 Ledger Library

The server communicates with a ledger to record all transactions. To accomplish this communication, the server uses a library that implements functions in order to interact with the blockchain network and publish the ledger data entries. Each ledger entry is identified with a tag s. The server uses one of the library's methods to publish s to the ledger. Synchronously the ledger returns an identifier that can be used to track the transaction and retrieve s. When it is asked, the server uses another library method to retrieve s.

3.3.6 Availability and Licensing

The prototype is available at `https://github.com/grnet/e-diplomata` and licensed under AGPLv3.

Chapter 4

Summary

We have presented three toolkits and three use-case application prototypes developed in the PRIViLEDGE project. The toolkits include a production-grade implementation of the Snarky Ceremonies protocol for zero-knowledge proofs, a toolkit adding ledger-based communication support and verifiability to multi-party computations, and a toolkit to support transparent access to both on-ledger and off-ledger data from applications. The use-case prototypes include a verifiable privacy-preserving voting system, verifiable privacy-preserving reports for health data, and privacy-preserving verification of academic degrees.

We believe all these software components represent clear advances over the state of the art available prior to the work done in the PRIViLEDGE project.

Bibliography

[CGS97] Ronald Cramer, Rosario Gennaro, and Berry Schoenmakers. A secure and optimally efficient multi-authority election scheme. *European Transactions on Telecommunications*, 8(5):481–490, 1997.

[KMSV21] Markulf Kohlweiss, Mary Maller, Janno Siim, and Mikhail Volkhov. Snarky ceremonies. Cryptology ePrint Archive, Report 2021/219, 2021. `https://eprint.iacr.org/2021/219`.

[PRI20a] Validation Criteria (D1.2). Technical report, PRIViLEDGE project, 2020.

[PRI20b] Report on Architecture for Privacy-Preserving Applications on Ledgers (D4.2). Technical report, PRIViLEDGE project, 2020.

[PRI20c] Final Report on Architecture (D4.3). Technical report, PRIViLEDGE project, 2020.

[PRI21a] Use Case Validation (D1.3). Technical report, PRIViLEDGE project, 2021.

[PRI21b] Revision of Extended Core Protocols (D3.3). Technical report, PRIViLEDGE project, 2021.

[PRI21c] Report on Tools for Secure Ledger Systems (D4.4). Technical report, PRIViLEDGE project, 2021.