FRESCO Documentation

Release

Dec 06, 2018

Contents

FRESCO is a FRamework for Efficient and Secure COmputation, written in Java and licensed under the open source MIT license. The project aims to ease the development of prototype applications based on secure computation.

These pages briefly document the project. If you have questions not answered here please ask using our issue tracker on GitHub or by email at fresco@alexandra.dk.

The FRESCO source code is available at https://github.com/aicis/fresco.

CHAPTER 1

Introduction

FRESCO is a FRamework for Efficient Secure COmputation. It is designed to make it easy and efficient to write prototype applications based on secure computation.

1.1 What is Secure Computation?

Secure computation (also known as Multi-Party Computation (MPC) or Computation on Encrypted Data (CoED)) is an emerging cryptographic tool that allows a number of parties to securely collaborate based on private data. More specifically, secure computation allows to jointly compute functions of private data from multiple parties, without revealing the underlying private data.

As an example consider the classic *Millionaires Problem*: Two millionaires meet on the street and want to decide who is the richest of the two. However, they fear embarrassment if they find that one millionaire is much poorer than the other. The millionaires can solve this problem by comparing their fortunes using secure computation to guarantee that they learn *only* who is the richest and no additional data is revealed. Furthermore, they can do so directly between each other, without having to involve any third parties.

In general any computable function can be computed privately using secure computation. To give a few examples the FRESCO framework has been used in prototypes to

- Compute statistical data from surveys without revealing the individual survey answers (in PRACTICE).
- Benchmark the financial and energy performance of companies while keeping private the performance data of the individual company (in the PRACTICE and Big Data by Security projects respectively).
- Let banks credit rate potential customers without revealing the private data of the customers *or* the private credit rating functions of the bank (in the Big Data by Security project).

For more information on secure computation see Wikipedia.

1.2 Main Features of FRESCO

The FRESCO framework aims to support the development of both new applications using secure computation, and the development of new secure computation techniques (referred to as *protocol suites* in FRESCO) to be used as the backend for those applications. In some sense FRESCO can be thought of as a *hub* that provides the infrastructure to connect applications with protocol suites. The framework puts focus on the following main features:

- **Rapid and simple application development**. With FRESCO you can write applications that use secure computation without being an expert in cryptography. You only need to specify which data to "close" and which data to "open". FRESCO provides a *standard library* of many commonly used secure functionalities. These can be easily combined in order to quickly achieve new complex functionalities for use in applications. Once you have written your application, you can run it using different kinds of protocol suites. This is important, since each suite comes with its own specific security level and performance, and you may not even know which kind of security is required at the time you write your application.
- **Rapid and simple protocol suite development**. FRESCO provides a collection of reusable patterns and components that allows protocol suites to be developed with minimal effort. Once you have developed your protocol suite, you immediately get the benefit that many existing applications (and tests) can run on top of your new suite.
- **Open and flexible design**. FRESCO provides great freedom regarding the way you implement your applications and protocol suites. Applications can, e.g., be specified in Java, or as a textual representation of a circuit. Protocol suites have full freedom and control over things such as thread scheduling and networking. It is even possible, using JNI, to write your protocol suite in C/C++ and still get the benefit of access to many existing applications written using FRESCO.
- **Support for large and efficient computations**. FRESCO supports techniques such as parallelization and preprocessing that enable scaling to large computations.

1.3 Contact

If you have any comments, questions or ideas, feel free to contact the FRESCO development team either by dropping a mail to fresco@alexandra.dk or by using our issue tracker at GitHub.

1.4 Related Projects

For further projects related to secure computation we refer to the Awesome-MPC list.

CHAPTER 2

Installation

FRESCO is designed to run on Linux, MacOS, and Windows. The following installation guide is tested on Linux and MacOS.

2.1 Building FRESCO from Source

The preferred way to install FRESCO is by building it from the latest source from GitHub. This way you get all the latest additions to FRESCO. To do this, make sure you have installed git, Java 8, and Maven.

Then in a terminal run:

```
git clone https://github.com/aicis/fresco.git
cd fresco
mvn install
```

This will download the FRESCO source code and dependencies, compile all the FRESCO modules, and run the test suite. On a successful build Maven should install the FRESCO modules on your system and a JAR file can now be found in the `./target` directory of each corresponding module, as well as in your local Maven repository. Note, that the test suite executed on `mvn install` can take several minutes. To skip the tests and only run the build, use `mvn install -DskipTests`.

If you use Maven for your project you can then use a FRESCO module by adding it as a dependency in your projects POM file. E.g., to use the `core` module add the dependency

```
<dependency>
  <groupId>dk.alexandra.fresco</groupId>
  <artifactId>core</artifactId>
  <version>1.0.1-SNAPSHOT</version>
</dependency>
```

possibly incrementing the version number to the current version.

In order to use one of the *protocol suites* in your project, you cat add it as a dependency as well. For instance, if you want to use the SPDZ protocol suite, your POM file will need to include:

```
<dependency>
  <groupId>dk.alexandra.fresco</groupId>
  <artifactId>spdz</artifactId>
  <version>1.0.1-SNAPSHOT</version>
</dependency>
```

2.2 Using the Latest FRESCO Release

If you prefer to install a released version of FRESCO you can get the source from the release site https://github.com/aicis/fresco/releases, and run `mvn install` as described above.

Alternatively If your project uses Maven you could just add the dependency to your projects POM file and have Maven download the dependency from the Central Repository. E.g., to use a release version of the `core` and `spdz` modules add the dependencies

```
<dependency>
  <groupId>dk.alexandra.fresco</groupId>
  <artifactId>core</artifactId>
  <version>1.0.0</version>
</dependency>

<dependency>
  <groupId>dk.alexandra.fresco</groupId>
  <artifactId>spdz</artifactId>
  <version>1.0.0</version>
</dependency>
```

possibly adjusting the version tag to the desired version.

2.3 FRESCO in a Docker Container

If you use Docker and would prefer to work with FRESCO in a Docker container, we have made a docker image available which you can run using

```
docker run -it frescompc/fresco
```

If you would like to build the docker image yourself we have included the *Dockerfile* in the root of the repository. To build the image simply clone the repository (as above) and run

```
docker build -t fresco .
```

To run the container interactively using the image run

```
docker run -it fresco
```

CHAPTER 3

Quickstart

This section gives a brief introduction on how to start working with FRESCO. If you have further questions please get in touch using our issue tracker or by email at fresco@alexandra.dk.

The best place to start is to browse the demos bundled with the FRESCO repository at https://github.com/aicis/fresco/tree/master/demos.

The following demos are currently included:

- Sum - computes the sum of a number of integers input by a single party.
- Distance - computes the distance between two points provided by two different parties (in Euclidean two dimensional space).
- Aggregation - computes the aggregation of a hard-coded list. The list consists of pairs of (key,value). The demo aggregates all values where the keys match.
- AES - computes an AES encryption of a block of plain-text provided by one party under a key provided by an other party.
- Private Set Intersection - computes the intersection of the private sets of two parties.

Each demo includes instructions on how to build and run them directly on the command line.

The demos should hopefully give you a sense of how secure computation is specified in FRESCO. To get started on your own applications you should also have a look at the various classes implementing the `ComputationDirectory` interface which gathers various generic functionality implemented in FRESCO which can be combined to realize more complex functionality. Specifically consider `Numeric` and `AdvancedNumeric` for arithmetic and `Binary` and `AdvancedBinary` for Boolean based secure computation.

3.1 A Simple Example

In this example we demonstrate how to use the FRESCO framework in your own application. FRESCO is a flexible framework intended to be used in your own software stack, so start by adding the dependency to fresco in your own project.

This example is based on the `DistanceDemo` class implementing the **Distance** demo outlined above. However, essentially any FRESCO application could be substituted for `DistanceDemo` in the following.

```java
DistanceDemo distDemo = new DistanceDemo(1, x, y);
Party me = new Party(1, "localhost", 8871);
DummyArithmeticProtocolSuite protocolSuite = new DummyArithmeticProtocolSuite();
SecureComputationEngine<DummyArithmeticResourcePool, ProtocolBuilderNumeric> sce =
    new SecureComputationEngineImpl<>(
        protocolSuite,
        new BatchedProtocolEvaluator<>(new BatchedStrategy<>(), protocolSuite));
BigInteger bigInteger = sce.runApplication(
    distDemo,
    new DummyArithmeticResourcePoolImpl(1, 1),
    new KryoNetNetwork(new NetworkConfigurationImpl(1, Collections.singletonMap(1,
        me))));
double dist = Math.sqrt(bigInteger.doubleValue());
```

Here we take the existing application, `DistanceDemo`, and run it with a single party using the dummy protocol suite. This can run directly in your own tests.

Congratulations on running your first FRESCO application!

If you want to see this run with multiple parties, the above example can be modified to include two parties running on the same machine.

```java
DistanceDemo distDemo = new DistanceDemo(1, x, y);
Party partyOne = new Party(1, "localhost", 8871);
Party partyTwo = new Party(2, "localhost", 8872);
DummyArithmeticProtocolSuite protocolSuite = new DummyArithmeticProtocolSuite();
SecureComputationEngine<DummyArithmeticResourcePool, ProtocolBuilderNumeric> sce =
    new SecureComputationEngineImpl<>(
        protocolSuite,
        new BatchedProtocolEvaluator<>(new BatchedStrategy<>(), protocolSuite));
HashMap<Integer, Party> parties = new HashMap<>();
parties.put(1, partyOne);
parties.put(2, partyTwo);
BigInteger bigInteger = sce.runApplication(
    distDemo,
    new DummyArithmeticResourcePoolImpl(myId, 2),
    new KryoNetNetwork(new NetworkConfigurationImpl(myId, parties)));
double dist = Math.sqrt(bigInteger.doubleValue());
```

3.2 A Little Explanation

Let's have a look at each part of the example above.

A FRESCO application, in this case `DistanceDemo`, implements the `Application` interface. To run an `Application` we must first create a `SecureComputationEngine`. This is a core component of FRESCO that is the primary entry point for executing secure computations through the computation directories and the active protocol suite.

The `SecureComputationEngine` is initialized with a `ProtocolSuite` and a `ProtocolEvaluator` (defining the secure computation technique and strategy for evaluating the application respectively). In this case we are using the `DummyArithmeticProtocolSuite` with the `BatchedProtocolEvaluator`.

To run an `Application`, we also need a `ResourcePool` and a `Network`. A `ResourcePool` is controlled by you, the application developer and is a central database of resources that the suite needs. The `Network` is the interconnected parties participating in the secure computation. By default FRESCO uses a `Network` implementation based on KryoNet as the network supplier, but you can create your own and use that if this matches your application better.

When we call `runApplication` the `SecureComputationEngine` executes the application and returns the evaluated result directly in a `BigInteger` - here the distance between the two points.

Notice how our `Application` is created. Implementing `Application` signals that our `DistanceDemo` class is a FRESCO application. An application must also state what it outputs as well as what type of application this is i.e. are we creating a binary or arithmetic application. This is seen in the interface

```
public interface Application<OutputT, Builder extends ProtocolBuilder> extends␣
↪Computation<OutputT, Builder>
```

The output type can be anything you want. In our case it is a `BigInteger`. The builder type we use here is a numeric type since the `DistanceDemo` computation works with numeric protocol suites. Since the `Application` interface extends the `Computation` interface, this requires us to implement the method

```
DRes<BigInteger> buildComputation(ProtocolBuilderNumeric producer)
```

This is the method that defines how our FRESCO application is built. The `DRes` return type represents a deferred result for the output (modeling that everything in FRESCO is evaluated “later”).

CHAPTER 4

Protocol Suites

Various techniques for secure computation are currently known. In the literature these are referred to as secure computation *protocols*. However, as these usually consist of a number of sub-protocols in FRESCO we use the term *protocol suites* to avoid ambiguity. I.e., in FRESCO a protocol suite is taken to be a set of sub-protocols that as a collection implements general secure computation.

FRESCO is designed to work with multiple interchangeable protocol suites and aims to support the development of new protocol suites. FRESCO also comes with a few protocol suites already implemented. The following table gives a rough comparison of the currently included protocol suites.

Suite	Parties	Adversary	Model of Computation	Reactive
Dummy Boolean	1+	none	Boolean	yes
Dummy Arithmetic	1+	none	Arithmetic	yes
TinyTables	2	semi-honest	Boolean	yes
SPDZ	2+	malicious	Arithmetic	yes
SPDZ2k	2+	malicious	Arithmetic	yes

Here the *Parties* column describes the number of parties that can be involved in secure computation using the given protocol suite. *Adversary* describes the type of adversary the protocol suite tolerates. The *Model of Computation* describes how the protocol represent the computation to be securely computed. Currently, protocol suites in FRESCO are tied to a single model of computation, i.e., the SPDZ suite only supports Arithmetic computations and does not support Boolean computations. Finally, a protocol suite being *reactive* means that it allows intermediate values to be opened, and further secure computation may continue on closed values that depend on the values opened so far.

Below we will describe the protocol suites in a little more detail.

4.1 The Dummy Boolean and Arithmetic Protocol Suites

The dummy suites do all computations in the clear and thus provide *no security at all*. These suites are intended for testing and debugging. Contrary to other protocol suites they can run with only one party.

The dummy suites are also useful for benchmarking: The overhead of the dummy suite can be seen as the *baseline* overhead of FRESCO when no security is applied.

4.2 The TinyTables Protocol Suite

The *TinyTables* protocol suite is based on work by Damgård *et al.* *[DNNR17]*. This protocol suite works in the Boolean setting, with exactly two parties and the original protocol comes in versions that provide security against both a semi-honest and malicious adversary. The version currently implemented in FRESCO, however, only implements security against semi-honest adversaries.

TinyTables uses a simple technique to preprocessing the function to be evaluated before the input is known. This preprocessing involves creating a small table of values for each *AND* gate involved, hence the name TinyTables. Online evaluation is reduced to a lookup into such a table for each *AND* gate with minimal communication overhead. As with other Boolean protocol suites, TinyTables evaluates XOR's locally without communication.

4.3 The SPDZ Protocol Suite

The SPDZ suite is based on another work of Damgård *et al.* *[DPSZ12]*. This protocol suite works over a finite field of size at least 2^s where s is a *statistical security parameter*. I.e., it works in the arithmetic setting. SPDZ allows for two or more parties to participate in the secure computation and is secure against malicious adversaries.

SPDZ is based on additive secret sharing over the given finite field. It requires a preprocessing step to produce so called *Beaver Triples* which will be used online to evaluate multiplications. Contrary to TinyTables, SPDZ preprocessing is not directly dependent on the function to be evaluated online beyond the number of multiplications to be performed. In the online evaluation, SPDZ uses the preprocessed data to evaluate each multiplication with a small amount of communication, whereas addition can be done locally.

4.4 The SPDZ2k Protocol Suite

The SPDZ2k protocol suite is based on a variant of SPDZ due to Cramer *et al.* *[CDESX18]*. In contrast to the regular SPDZ protocol which works over a field SPDZ2k works over the ring Z_{2^k} for some k. This is an advantage as working in such a ring more closely resembles how arithmetic on the integers behaves in normal programming languages and it allows for various optimizations compared to working over a field.

The current implementation supports all numeric native protocols, however there are higher level computations (for instance equality and comparison) which it is not yet compatible with. Support for those is forthcoming.

4.5 References

[DNNR17]:

Ivan Damgård, Jesper Buus Nielsen, Michael Nielsen and Samuel Ranellucci

The TinyTable Protocol for 2-Party Secure Computation, or: Gate-Scrambling Revisited

CRYPTO 2017

[DPSZ12]:

Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl and Nigel P. Smart
Practical Covertly Secure MPC for Dishonest Majority – Or: Breaking the SPDZ Limits
ESORICS 2013

[CDESX18]:
Ronald Cramer, Ivan Damgård, Daniel Escudero, Peter Scholl, and Chaoping Xing
SPDZ_2^k: Efficient MPC mod 2^k for Dishonest Majority
Unpublished

CHAPTER 5

Contributing

We will be happy to accept contributions to the FRESCO framework. Contributions could include:

- **Feedback** - Whether you have a *question* on FRESCO, a *suggestion* for new additions or noticed a *bug* in the framework, any feedback is a valuable contribution.
- **Documentation** - Any improvements or additions you may have to the documentation found on these pages are very welcome. For more on how to work on the documentation, see *here*.
- **Code** - We will be very happy to include well-written code that is compliant with the overall FRESCO design. This could be bug fixes, new protocol suites, generic functionality fitting the FRESCO standard library or improvements to the core framework etc. For more on how to work with the FRESCO code see *here*.

We encourage you to use our issue tracker to provide feedback or discuss any other contributions you would like to make. Alternatively, you can contact us at fresco@alexandra.dk.

5.1 Pull Requests

The easiest way to contribute code or documentation is to send us a pull request on GitHub. Please follow these steps:

1. Create an issue related to your planned contribution on the issue tracker.
2. Create a fork of the FRESCO repository (find directions at http://github.com/aicis/fresco).
3. Make your changes to the new fork.
4. Make a *Pull Request* at GitHub.

Before being merged into the project your pull request will be reviewed by the FRESCO team. To ease this process, please keep your pull requests focused on the issue being solved. Also, please give the pull request an informative title and description, preferably with references to the issues related to the request (see https://help.github.com/articles/autolinked-references-and-urls/#issues-and-pull-requests).

CHAPTER 6

For Developers

In this section we give some tips and guidelines for developing and contributing code to FRESCO.

6.1 Directory Structure

The FRESCO root directory (see https://github.com/aicis/fresco) contains a number sub-directories containing multiple sub-projects for FRESCO. Here we describe the most important directories:

- core - contains the core FRESCO framework including the application interface, library of generic functionality and dummy suites.
- demos - contains a number of demos demonstrating how FRESCO can be used. Each demo has its own sub-project.
- doc - contains the source of the documentation for this site.
- suite - contains the protocol suites implemented in FRESCO each as its own sub-project.
- tools - contains various tools used in FRESCO each as its own sub-project.

We use Maven to manage FRESCO, and within each sub-project we use the standard Maven directory structure.

6.2 Developing using an IDE

The FRESCO team (mostly) uses Eclipse to develop FRESCO. To develop using Eclipse, first check out a fork of FRESCO from GitHub. Then to import FRESCO into Eclipse choose

```
File > Import... > Maven > Existing Maven Projects
```

and select the FRESCO root directory.

To help conform to the code style used in FRESCO, as described in the *Code Style* section, we recommend installing the Checkstyle plugin for Eclipse and configuring it to use the Google style. This plugin can also be used to generate a code formatter for Eclipse. To ensure imports are ordered correctly, you may also need to go to

```
Eclipse > Preferences... > Java > Code Style > Organize Imports
```

and delete all groups in the list displayed (as the Google style dictates that all imports must be in a single block).

To help fulfill the code coverage goal described in *Testing* we also recommend installing the Jacoco plugin. This eases checking code coverage on your changes locally.

Alternatively some IntelliJ support is also present - look for the .idea files in the root of the repository.

6.3 Code Style

To keep the code style consistent we use the style defined by Google. We prefer to keep the code from generating compile warnings, using the `@SuppressWarnings` annotation sparingly in case of unavoidable warnings.

6.4 Testing

We use JUnit4 for testing. We use the Travis tool to continuously check that all code committed to the repository compiles and passes all tests. We strive for 100% test coverage of the FRESCO code and use the Codecov tool to automatically check that new patches have 100% coverage and do not decrease the overall test coverage.

For each sub-project tests are located in the source code folder named `test` separated from the main code, as per the standard Maven directory structure. When writing tests for something in package `x.y.z` the test should belong to the same package. This way, methods that are *package private* and therefore not exposed in the FRESCO API can also be tested.

We work with two classes of tests:

- Regular tests. These should be fast and not rely on any external dependencies such as a server already running. I.e., it should always possible to check out the code and just run these tests with only meeting the requirements seen in the *install* section.
- Integration tests. These are tests that for example rely on external databases being set up, or involve deployment to different hosts. You can mark a test class or test method as integration test by using the `@Category` annotation like so:

  ```
  @Test @Category(IntegrationTest.class) public void testSomething() { // Your test␣
  ↪goes here. }
  ```

Integration tests are ignored when you the FRESCO tests suite using the Maven command

```
mvn test
```

but are included when you run

```
mvn integration-test
```

A few good practices regarding tests:

1. Write tests.
2. Don't delete, comment, or `@Ignore` tests unless you really know what you are doing.
3. Make sure that tests are independent of each other.
4. Tests should be deterministic. Use a pseudo-random generator with a fixed seed if you need randomness.
5. Working tests should be silent when they work. Use log level `Level.FINE` if needed.

6.4.1 Writing Tests for a Protocol Suite

If you are developing a new protocol suite you may want to write tests in the same way as the tests for suites that are already included in FRESCO. Consider, e.g., the SPDZ suite. A helper method is made:

```
protected void runTest(TestThreadRunner.TestThreadFactory f, EvaluationStrategy␣
↪evalStrategy,
      NetworkingStrategy network, PreprocessingStrategy preProStrat, int noOfParties)␣
↪throws Exception
```

The first argument to `runTest` is a `TestThreadFactory` which defines which logic should be tested. It is a factory that provides threads for each party in the test. If the protocol to test is symmetric, each thread is identical. The test framework makes sure that each thread has access to its own `partyId` so if the test requires the parties to do different things, they can branch on their partyId.

The rest of the arguments to `runTest` are parameters over which you want your tests to vary. For example this could be the number of players and evaluation strategy. But it can also include parameters specific to your suite. The `runTest` should set up the remaining parameters for your test – those parameters that should remain fixed in all your tests.

Then create a number of small tests, like the following:

```
@Test
public void test_MultAndAdd_Sequential() throws Exception {
  runTest(new BasicArithmeticTests.TestSimpleMultAndAdd(), EvaluationStrategy.
↪SEQUENTIAL,
    NetworkingStrategy.KRYONET, PreprocessingStrategy.DUMMY, 2);
  }
```

It is fine to let the name reflect the specific parameters used in the test. Note how we use a generic test here: The test `BasicArithmeticTests.TestSimpleMultAndAdd` can be used to test multiplications and additions for any protocol suite that supports basic arithmetic operations, so there is no need to rewrite such tests. Only write your own specific tests if you need to test some specific functionality of your suite that no other suite has, otherwise consider making the test generic such that it can be reused by others.

Writing many small tests like this makes it easy to decide later which of the tests to include. The "unit" test suite should be relatively quick and not require external setup. If it depends on such things, mark it with `@Category(IntegrationTest.class)`.

6.5 Building the Documentation

The documentation will be built automatically and uploaded to fresco.readthedocs.org when new changes are pushed to the repository. Before committing changes to the documentation, it is a good idea to build the documentation locally and check that it looks ok. This can be done as follows.

Building the docs requires Sphinx to be installed. A good way to do this is by using *virtualenv*. Using virtualenv installs Sphinx in a local folder that can be easily removed, and it ensures that the installation does not have any side effects: Go to the `doc` folder. Then create a new virtual environment:

```
virtualenv env
source ./env/bin/activate
pip install -r requirements.txt
```

If the install fails, you might have to update pip. Just follow the directions pip gives you. This only needs to be done once. When done, you can activate the virtual environment just by doing:

```
source ./env/bin/activate
```

Once activated, you can build documentation with:

```
make html
```

On Mac OS X you may need to set the following environment variables:

```
export LC_ALL=en_US.UTF-8
export LANG=en_US.UTF-8
```

You can enter the two lines directly in your terminal or to add them to your `~/.bash_profile`.

Once built, you can view the result, open the file `doc/build/html/index.hmtl` with a web browser.