Practical Assessment of Secure Multiparty Computation Frameworks

Jakub Wójcik
Advisor: Marcel von Maltitz
Seminar Future Internet WS2017/2018
Chair of Network Architectures and Services
Departments of Informatics, Technical University of Munich
Email: jakub.wojcik@tum.de

ABSTRACT

This paper describes, assesses and compares the secure multi-party computation frameworks FRESCO and Bristol SPDZ in terms of infrastructural differences, practicality and performance. It provides a step by step guide to performing computations with the frameworks. The measurement were performed in a virtual environment.

Bristol SPDZ is faster and uses less memory. It provides an easy way to define computations by writing simple Python scripts. FRESCO is more suited for rapid development by integrating into a Java development process. Its modular design makes it equipped for the future.

1. INTRODUCTION

Secure multi-party Computation is a cryptographic method to perform joint calculations of arithmetical functions by multiple parties without them getting to know each other's input values. Multiple names and abbreviations have been used, such as secure computation (SC) or multi-party computation (MPC), but in the paper the term used will be secure multi-party computation (SMPC).

What makes SMPC different than other forms of cryptography is the fact that it treats the participating parties as adversaries. The mathematical groundwork has been laid in the 90s, but SMPC was just considered theoretically for a long time. An important step was Shamir's secret sharing[16], which is a method to share a secret between multiple parties, so that together they can reconstruct it, while on their own they have only useless information. Equally important were proofs that secure protocols exist, such as in the paper by Ben-Or, Goldwasser and Wigderson[11], which is based on Shamir's secret sharing and is still in use nowadays. Only in recent years, due to better computation power and advancements in SMPC protocols, such as the development of the SPDZ protocol[13], practical implementations have become feasible.

A few actively developed and maintained frameworks have been created. Among them are e.g. Sharemind, FRESCO and Bristol SPDZ. Sharemind[6] is a company selling solutions based on Shamir's secret sharing to businesses wanting to compare themselves to others without releasing their private data. Because this is a typical application of SMPC, this paper will also use statistical functions to assess the frameworks.

This paper looks at two of the frameworks in detail: FRESCO and Bristol SPDZ. In section 2 each framework is described on it's own, while in section 3 they are compared to each other in respect to infrastructural differences, practicality and performance. The frameworks are considered production ready, if it is possible to perform thousands of computations in a negligible timespan, while also hiding the implementation details from the end user. This requirement makes the frameworks ready to use in stock exchanges or social media applications where results have to be provided instantly. Finally section 4 lists some ideas of future work to be done and section 5 offers a conclusion to the comparison of the frameworks.

2. CONSIDERED SMPC FRAMEWORKS

The considered frameworks FRESCO and Bristol SPDZ are both active open-source SMPC solutions. They are currently being developed on GitHub with the latest contributions on both being from November 2017.

The points listed above are all the similarities off the frameworks, virtually everything else about them is different. The frameworks use different tool-sets, follow different paradigms, and have been developed for different reasons.

2.1 FRESCO

The name FRESCO is an abbreviation of "A FRamework for Efficient Secure COmputation"[1]. The framework is an abstraction from the different specific SMPC protocols, that are called protocol suites in the context of FRESCO. Instead it is meant as a foundation which allows for any protocol suite to be used to compute the same functions.

FRESCO implements multiple protocol suites, such as BenOr-Goldwasser-Wigderson (BGW) that is secure for an honest majority. That means that if the majority of the parties doesn't deviate from the protocol, it is guaranteed that no one will obtain any additional information[11], only the computed function value.

Other suites are also implemented, e.g. SPDZ, but in this paper only the BGW suite will be taken into account because SPDZ is currently under active development[2]. For more information see section 4.

FRESCO is developed in Java 8. It is packaged with all required dependencies by Maven in a JAR which can be downloaded from its website[5]. Computations are also meant to be written in Java by extending the `Application` class and implementing the sequence of computation steps to be performed. It can be embedded anywhere in a Java program, but the documentation provides a sensible default application that can be used to start FRESCO from the command line. Providing necessary configuration from the command line arguments, the application is run in one line.

The developer using FRESCO is able to build their own Java programs that call into FRESCO when SMPC is to be performed. They can bundle FRESCO with their program and deploy it as a single Java executable.

2.2 Bristol SPDZ

The SPDZ (pronounced ”speedz”[10]) protocol has been introduced in 2011 and it uses different mathematical concepts than BGW, namely somewhat homomorphic encryption[13][7]. In has been improved twice. The first time in 2012 solving open problems of the original paper[12]. The second time in 2016 when MASCOT, a new protocol for the preprocessing phase, was introduced[15].

The special thing about SPDZ is that it is divided into two phases. The first one is called the offline phase, or rather the preprocessing phase as a connection between the parties is required. This phase generates triples, which are used to perform multiplications, and Message Authentication Codes (MACs), which use asymmetric cryptography to ensure that the values provided by the parties are correct. Both triples and MACs are used later during the online phase. In the online phase the actual computation is performed.

The online phase is designed to be fast, hence the name of the protocol. The computationally heavy stuff is performed during the preprocessing phase which can take place e.g. at night or on weekends, when no computations need to be performed. During the online phase MACs are needed for every step of the computation, while for every multiplication three previously generated triples are consumed. A lot of both has to be prepared beforehand.[15]

Bristol SPDZ is developed in C++ and Python 2. It has to be compiled from source and it depends on a few other libraries that have to be installed manually.[4]

The implementation of the SPDZ protocol is first and foremost a showcasing of the improved preprocessing phase MASCOT which has been proposed by the same people that develop the implementation.[7] As such it does not hide its internals and instead exposes every step as a separate executable or script that can be run.

3. ASSESSMENT OF THE FRAMEWORKS

The frameworks are compared and assessed in respect to practicality and performance.

As they have been developed with different use cases in mind, often no direct comparison is possible. Instead the execution of the chosen solutions is assessed.

3.1 Setup and methodology

In order to assess all relevant differences between the frameworks in a realistic setting, a fresh OS was set up in a virtual environment and all steps needed to achieve the goal of computing some statistical functions were counted towards practicality.

Used statistical functions. As performing statistics with SCMP is a common use case, in this paper statistical functions were used during the practicality and performance tests. The functions were chosen to include all of the basic arithmetical operations as it is possible to compute them using SMPC: addition, subtraction, multiplication and division.

The first one is the average function:

$$\mu = \frac{1}{n}\sum_{i=1}^{n} x_i$$

The second one is variance:

$$\mathrm{Var}(X) = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2$$

Having the variance, it is easy to calculate the standard deviation by taking the square root. It is not yet possible to compute the square root with SMPC.

$$\sigma = \sqrt{\mathrm{Var}(X)}$$

Both frameworks were setup to calculate the functions in the same scenario. There were four parties configured, each of them having a single integer x_i as its secret input value. This section describes the effort it takes to configure the setup and measures the performance of the computations.

Virtual environment. The frameworks were tested in a virtual environment. The virtual environment was created using Vagrant and VirtualBox.

The chosen OS was `ubuntu/trusty64`[8] with 1024 MB of RAM. The VMs were running on a relatively modern processor with eight virtual cores and a clock speed of 1.2 GHz.

The performance of the framework was measured on the VMs with the Linux `time` command. The memory consumption was measured with `valgrind`. Although the values might not represent real conditions, they certainly are adequate to use them in comparisons with each other.

3.2 Key (infrastructural) differences

This section looks at the frameworks from an infrastructural point of view. Considered are the installation process, the dependencies, and the way the computations of the arithmetic functions have to be defined and performed. It also looks into the paradigm behind the frameworks.

The section also provides a step by step guide to performing computations with the frameworks.

Installation and dependencies. Installing FRESCO is very straightforward. An prepackaged jar with all required dependencies can be downloaded from the FRESCO website[5].

Alternatively the framework can be compiled from source using Maven, which is only mildly more complicated. The only necessary dependencies system wise are the Java Development Kit version 1.8 and Maven in order to compile from source. Both of these are available out of the box on modern Linux distributions.

To be able to install Bristol SPDZ, first the C++ compiler `g++`, Python 2 and m4 have to be installed. They are available in the repositories of most modern Linux distributions. Additionally the Multiple Precision Integers and Rationals Library (MPIR) has to be installed. The instructions are available in the manual[14]. Simply run `configure` with the `-enable-cxx` flag to enable C++ compatibility.

To compile and install Bristol SPDZ the source code has to be downloaded from the repository[4]. The git submodule `SimpleOT`, which is necessary for MASCOT to work, has to be downloaded, too. In the `CONFIG` file, the flag `USE_GF2N_LONG`[1] has to be set to 1. Afterwards compilation is as simple as running `make`.

Defining computations. In order to define computations in FRESCO the `Application` class has to be extended and the `prepareApplication` method overridden. In the method body all of the steps required to compute the arithmetic function have to be defined using `ProtocolProducer`s. These steps form the protocol that will be performed with the specified protocol suite when running the application. Inserting the values into the computation and getting the output of the function has to be dealt with. There is a difference between secret and open values represented by types prefixed with an S or O. For example `SInt` and `OInt` are secret and open integer values, respectively. The secret numbers are effectively shared secrets. No party has enough information to know the values. Only together they can reconstruct it to create an open integer.

The order in which the steps will be computed has to be defined as well. Which operations will be performed sequentially and which ones in parallel. The developer of the are responsible for the optimization of the computation. It is possible to define one application as an extension of another.

Additionally the application must be run by calling the method `SCE.runApplication`. This can happen anywhere in the code. To compile the Java program the compiler has to know where it can find the packages of FRESCO. This is typically done by setting the `classpath` argument:

```
javac -cp .:fresco-0.2-jar-with-dependencies.jar
```

To define computations in Bristol SPDZ, a file with the extension `.mpc` inside of the subdirectory `Programs/Source/` has to be created. Inside the file the computation is defined by writing a Python script that operates on values of custom data types.

[1]Enables the use of a 128 bit GF($2n$) field that is needed by MASCOT.

As with FRESCO in Bristol SPDZ there is a difference between secret and open data types. For example there are `sint` and `oint` representing secret and open integers, respectively. Inserting the private values into the computation and revealing the result has to be dealt with, but other than that the standard arithmetical operators can be used.

The definition of every computation has to be compiled by calling `compile.py` from the SPDZ directory with the filename of the program, but without the subdirectory. For the program to work with MASCOT, the flags `-p 128 -g 128` have to be passed. The flags set the sizes of the internally used data types. They are necessary because the compiler and MASCOT use different sizes by default. The compilation will generate multiple files inside of `Programs/`.

Paradigm. FRESCO clearly follows the paradigm of allowing to define computations independently of the protocol suite that will be used to perform the actual computation. Being developed purely in Java, it also can be neatly tied into existing Java applications. It is designed to be used by developers as a Java library to perform SMPC[3].

The paradigm behind Bristol SPDZ seems to be to create a working implementation of the newest improvement to the SPDZ protocol. It is meant as an showcasing of the MASCOT protocol. As such it does not focus on usability and instead gives the user full control over each internal step.

Performing computations. In order to perform a computation with FRESCO, the location of FRESCO has to be specified to the JVM in the `classpath` argument, again. When using the provided default program, which reads the parameters from the command line, additionally the protocol suite and the addresses of all the participating parties have to be specified. The number of the current party has to be specified, as well. Then all of the parties run the program having access only to their own private inputs. Party number one of two would run the following to use the BGW protocol suite:

```
java -cp .:fresco-0.2-jar-with-dependencies.jar App
-s bgw -Dbgw.threshold=1
-p 1:192.168.33.10:9001 -p 2:192.168.33.11:9002
-i 1
```

The program automatically connects to the other parties and performs the computation.

In order to perform an computation with Bristol SPDZ, multiple steps have to executed:

- Prepare the data for the online mode by running `Scripts/setup-online.sh 4 128 128`.
  The 4 in this context is the number of participating parties, while the 128 are the sizes of the used data types. They have to match the values provided to the compiler.
- Distribute the compiled program and the prepared data among the participating parties.

- Prepare a file `HOSTS` with the addresses of the parties. It is needed for MASCOT to work.
- Perform the MASCOT preprocessing phase by every party running `ot-offline.x -N 4 -p 0`. The flag `-N` defines the number of participating parties, while the flag `-p` defines which party one is. With the flag `-n` the number of generated triples can be set.
- Prepare the private input values in a separate file by running `gen_input_f2n.x` or `gen_input_fp.x` on each party. These tools encode the provided input values into a binary format.
- Start a server which will coordinate the communication between the parties by running `Server.x`. The server is only needed to establish a connection between the parties, the communication itself is peer to peer. The players have to be reachable by hostname.
- Run the online phase on every party.
  `Player-Online.x -lg2 128 -pn <server port> -h <server host> <player id>`.

3.3 Practicality

Practicality in this context means how well the framework can be used. This applies to its paradigm, how easy it is to write computations and how well the framework is documented.

3.3.1 FRESCO

FRESCO is designed for rapid development and it mostly fulfills this promise. One major drawback is the lack of up to date documentation.

The framework is undergoing some structural changes, so the documentation does not always line up with the actual implementation. This might change in the near future (cf. section 4). In the meantime it is probably better to learn using the framework by looking at working examples. The developers include a few demos with FRESCO.[1]

Defining computations. From an infrastructural point of view, defining computations in FRESCO works very well. Because of its nature as a Java library, defining new computations can be tied into an existing development process. It is possible to use FRESCO with IDEs such as Eclipse.

On the other hand, FRESCO forces the developer to define all the steps to perform the computation. They have to decide which steps have to be performed sequentially and which ones in parallel. This requires a deeper understanding of underling mechanisms and can get quite verbose. For example in the applications used in this paper the definition of the average computation has 15 lines, while the definition of variance has 30 line of code. This is just to define the steps of the computation. Any other code was not counted.

Additionally any optimization has to be performed manually. A handy feature showcased in the demos is that already defined computations can be reused to build more complicated ones by inheriting from another `Application` class.

Performing computations. This is where FRESCO shines.

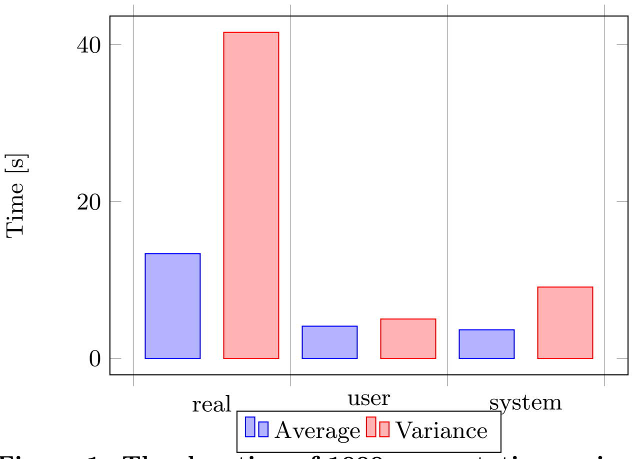

Figure 1: The duration of 1000 computations using FRESCO BGW.

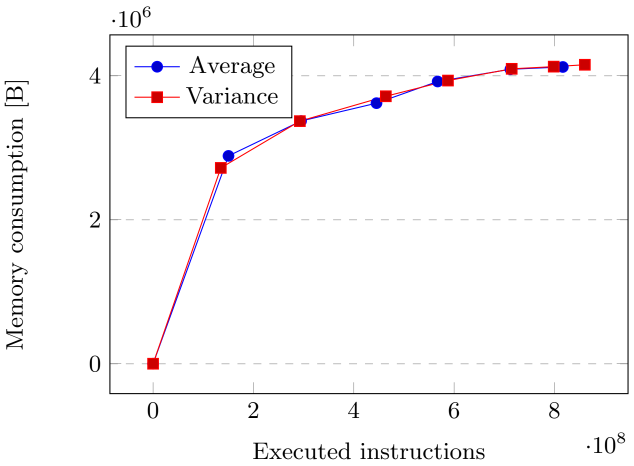

Figure 2: Memory consumption of one computation using FRESCO BGW.

Performing a computation is very straight forward. There is one executable, which takes all its configuration from command line parameters in one place. Only the protocol suite to use has to be chosen, but this offers a lot of flexibility. In the future, it should be even possible to use new protocols that are not implemented yet.[3]

3.3.2 Bristol SPDZ

Bristol SPDZ has been developed to showcase MASCOT. It has great features, but at the same time it is quite cumbersome to use.

Defining computations. This is where Bristol SPDZ shines.

Defining computations is very easy because it is just writing a Python script. The sequence in which the operations are computed and the optimization are inferred automatically.

The only drawback is a lack of a documentation about the custom data types. One can either look into the source code to learn which ones to use, or look at working examples or demos which are included in Bristol SPDZ.[4]

Performing computations. Performing computations with Bristol SPDZ is very cumbersome. Every step and phase has to be run manually and a lot of files are generated in the process. Of course a the process can be simplified by using a shell script, when the desired configuration has been identified. In the repository there is even the script `Scripts/run-common.sh`[4] that can be used for this purpose, but it is not compatible with every setup.

The most annoying thing is that the same options have different names for the tools on different steps. The commands of each step also have their own unique formats. For example MASCOT uses a hosts file to communicate with the other parties while the online phase uses a server.

In order to be reachable by the hostnames, the file `/etc/hosts` has to be edited on every party. There should be a better way of getting a connection.

3.4 Performance

The performance of the frameworks was measured using the Linux `time` command. It returns three values: real, user and system. The value real stands for the actual duration the program was running. The other values show how much the time the processor spent in user and system mode, respectively.

It can be said that in general in user mode the arithmetical operations are performed. In system mode on the other hand it's mostly read and write operations and network communication. The sum of the times spent in both modes is not necessarily equal to the real time because the processor might spend some time on other calculations, such as running different processes. This is especially important considering that the tests were run in a virtual machine.

To get a better picture of the time the computation actually takes, and not to measure the startup time of the program, each computation is performed 1000 times.

In order to measure the amount of memory the programs consumed, the profiler `valgrind massif`[9] was used:
`valgrind -tool=massif -heap=yes -stacks=yes`

This command returns the size of the allocated heap and stack space during the runtime of the program. In the graphs the x-axis shows the number of instructions executed by the processor. This offers a overview over the memory consumption in different stages of the program, although the executed instructions are not necessarily proportional to the runtime of the process.

3.4.1 FRESCO

The performance of FRESCO is quite stable. As can be seen in figure 1 the time it takes to perform a 1000 simple computations is over 10 seconds. One thousand computations of average took 13.36 s, while the computations of variance took 41.57 s. The slightly more complicated computation of variance which has twice as many lines as average took nearly three times as long.

This does not feel like much when performed on its own, but it certainly means that performing a lot of computations in

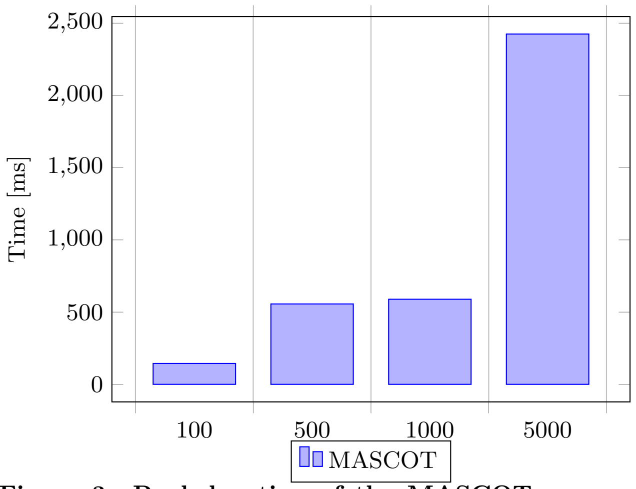

Figure 3: Real duration of the MASCOT preprocessing phase generating n triples.

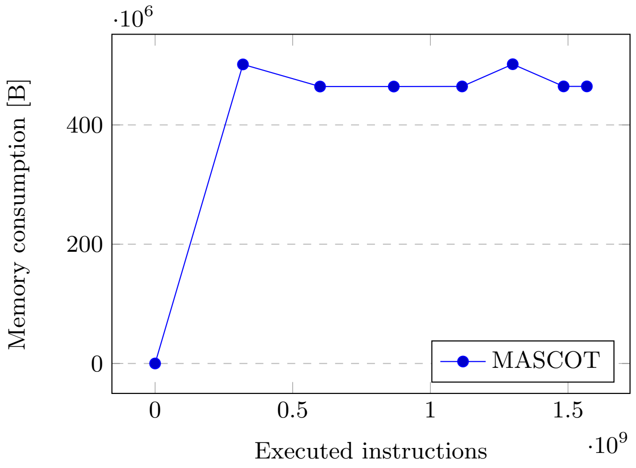

Figure 4: Memory consumption of MASCOT generating 5000 triples.

batch could take hours. It also means that FRESCO is not suitable for application in social media or stock exchanges where rapid response times are very important.

An interesting thing to note is that the computation time reported by FRESCO by enclosing the call to the `runApplication` method and the running time measured by the system are consistently about one second off. This probably means that one second is the startup time of the program and the Java Virtual Machine.

Memory consumption of FRESCO is reasonable. It constantly increased during the computations and reached about 4.0 MiB at the end. The progression can be seen in figure 2.

3.4.2 Bristol SPDZ

In Bristol SPDZ the performance measurement are divided between the preprocessing and online phase because they can be performed independently. The first one might be performed at night and the latter during the day when SMPC in needed.

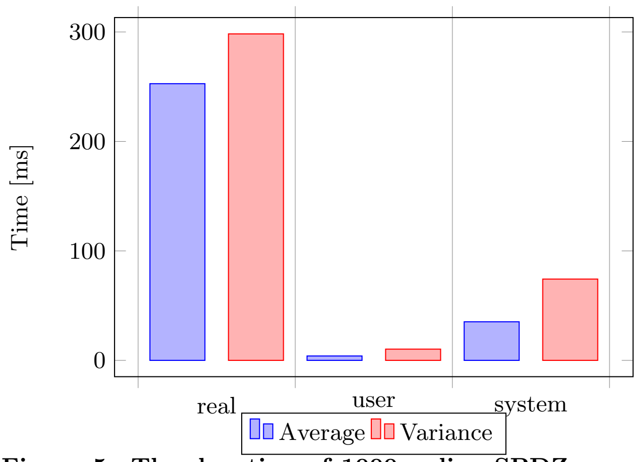

Figure 5: The duration of 1000 online SPDZ computations.

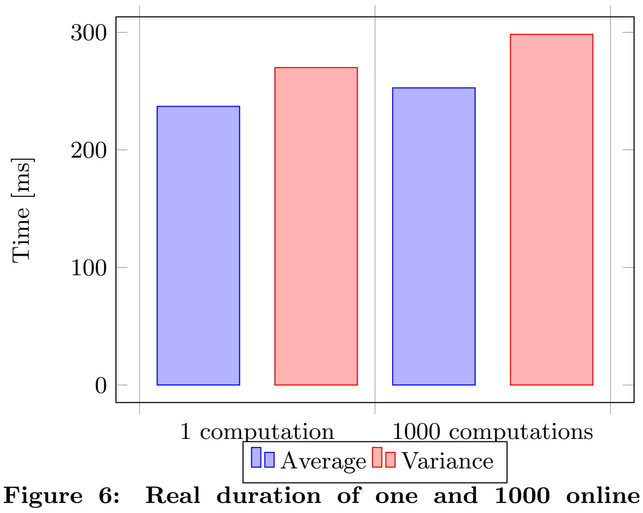

Figure 6: Real duration of one and 1000 online SPDZ computations.

Preprocessing phase. To measure the performance of MASCOT a different number of triples were generated. With more triples the running time rises significantly (see figure 3). It took 2.42 s to generate 5000 triples.

The memory consumption of MASCOT is very stable. While generating 5000 triple the memory usage was about 470 MB which is nearly half of the available RAM on the virtual machine (see figure 4).

Online phase. Compared to MASCOT, the online phase is very fast. One thousand computations of average took 253 ms, while the computations of variance took 298 ms (see figure 5).

Such speeds make it certainly possible to be used in application where speed is important, such as social media or stock exchanges. One has to keep in mind that this speeds are only possible if there are previously generated triples and MACs. When the parties run out of those, further computations become impossible. The generation of additional tripes is much slower (cf. figure 3).

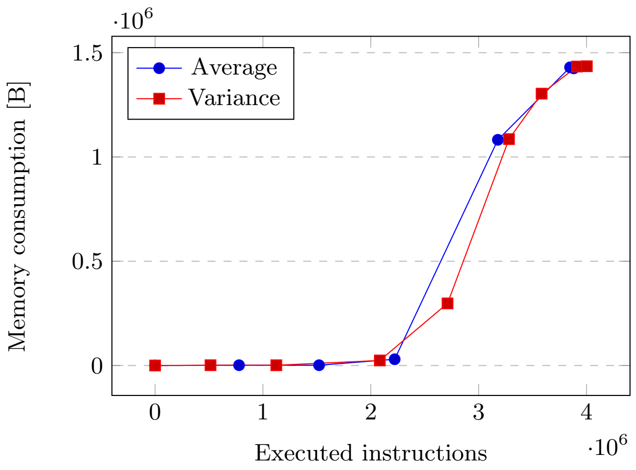

Figure 7: Memory consumption of one online SPDZ computation.

An interesting thing to note is that the duration time of one computation and 1000 computations are nearly identical (see figure 6).

The memory consumption of never exceeds 1.3 MiB (see figure 7). It is nearly zero for a long time, which according to the massif documentation[9] is a normal behavior of short running programs because most of the time they spend loading external libraries. This also explains the little difference between one and 1000 computations.

4. FUTURE WORK

Both of the frameworks are in active development with the latest contributions made in November 2017. This means that the information in this paper will not stay up to date for a long time. It is necessary to follow the development closely.

FRESCO. The current version of the framework FRESCO, version 0.2, is currently undergoing some structural changes. Because of this a lot of the features and protocol suites are not correctly documented.

Additionally a SPDZ protocol suite with the MASCOT preprocessing phase[2] that is used in Bristol SPDZ is currently being implemented by the FRESCO team. As it is not yet ready to be used, the SPDZ protocol suite has not been taken into account in this paper. Having two completely different implementations of the same protocol, it would certainly be interesting to measure the performance differences of both. Once the implementation is done, it will surely be worth to compare it to Bristol SPDZ and assess other differences, such as ease of use.

Actual network. All of the measurements in this paper were performed in a virtual environment. In order to understand how the frameworks perform in real circumstances, it would be a good additional analysis to perform the computations on an actual network.

Other parameters that can be varied are the number of participating parties and the connection speed. It would also be interesting to see how the frameworks work over the Internet with the machines of the different parties being in different physical locations. This would probably be the closest scenario to a computation in real circumstances.

5. CONCLUSION

The frameworks are have been designed with different goals in mind. It is therefore impossible to deem one better then the other. Which one is better suited for a task depends heavily on which paradigm is better aligned with the goal of the task.

In terms of performance Bristol SPDZ clearly wins. It is faster and uses less memory. Even considering the sum of both phases it still is faster than FRESCO. Another place where Bristol SPDZ is strong, is defining the computations, which basically is the straightforward task of writing a Python script. Where it certainly lags behind is its lacks of a cohesive structure to the different execution steps.

FRESCO on the other hand wins when it comes to rapid development. It is easier to integrate into a development process and easier to run the implemented computations. Its modular design with different suites makes it better equipped for the future.

Application developed with both frameworks can be made production ready as defined in the introduction. The developer using FRESCO will have to make sure the number operations does not make their program slow, while the developer using Bristol SPDZ will have to put more effort into making SMPC work seamlessly in the background. Depending on the context both can be adequate compromises.

6. REFERENCES

[1] A framework for efficient secure computation. On GitHub `https://github.com/aicis/fresco`. Accessed: 2017-11-16.

[2] Implement proper spdz preprocessing. `https://github.com/aicis/fresco/issues/112`. Accessed: 2017-11-18.

[3] Introduction - fresco 0.2.0 documentation. `http://fresco.readthedocs.io/en/latest/intro.html`. Accessed: 2017-10-01.

[4] Multiparty computation with spdz online phase and mascot offline phase. On GitHub `https://github.com/bristolcrypto/SPDZ-2`. Accessed: 2017-11-16.

[5] Releases - fresco 0.2.0 documentation. `http://fresco.readthedocs.io/en/latest/releases.html`. Accessed: 2017-09-05.

[6] Sharemind | privacy enhancing technology for data-driven business. `https://sharemind.cyber.ee/`. Accessed: 2017-10-01.

[7] Spdz software | bristol university | department of computer science. `https://www.cs.bris.ac.uk/Research/CryptographySecurity/SPDZ/`. Accessed: 2017-11-18.

[8] Vagrant box ubuntu/trusty64. `https://app.vagrantup.com/ubuntu/boxes/trusty64`. Accessed: 2017-09-04.

[9] Valgrind user manual - massif: a heap profiler. `http://valgrind.org/docs/manual/ms-manual.html`. Accessed: 2017-09-30.

[10] What is spdz? part 1: Mpc circuit evaluation overview. `https://bristolcrypto.blogspot.de/2016/10/what-is-spdz-part-1-mpc-circuit.html`. Accessed: 2017-11-18.

[11] M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness Theorems for Non-Cryptographic Fault Tolerant Distributed Computation. *Proceedings of the 20th Annual ACM Symposium on the Theory of Computing (STOC)*, pages 1–10, 1988.

[12] I. Damgard, M. Keller, E. Larraia, V. Pastro, P. Scholl, and N. P. Smart. Practical covertly secure mpc for dishonest majority – or: Breaking the spdz limits. Cryptology ePrint Archive, Report 2012/642, 2012. `http://eprint.iacr.org/2012/642`.

[13] I. Damgard, V. Pastro, N. Smart, and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. Cryptology ePrint Archive, Report 2011/535, 2011. `http://eprint.iacr.org/2011/535`.

[14] T. Granlund and W. Hart. The multiple precision integers and rationals library. `http://mpir.org/mpir-2.7.2.pdf`, 19 November 2015. Accessed: 2017-09-30.

[15] M. Keller, E. Orsini, and P. Scholl. Mascot: Faster malicious arithmetic secure computation with oblivious transfer. Cryptology ePrint Archive, Report 2016/505, 2016. `http://eprint.iacr.org/2016/505`.

[16] A. Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, Nov. 1979.