

A Performance and Resource Consumption Assessment of Secure Multiparty Computation

Marcel von Maltitz and Georg Carle
Technische Universität München, Department of Informatics
Chair for Network Architectures and Services
85748 Garching b. München, Germany
{vonmaltitz | carle}@net.in.tum.de

Abstract—In recent years, secure multiparty computation (SMC) advanced from a theoretical technique to a practically applicable technology. Several frameworks were proposed of which some are still actively developed.

We perform a first comprehensive study of performance characteristics of SMC protocols using a promising implementation based on secret sharing, a common and state-of-the-art foundation. Therefore, we analyze its scalability with respect to environmental parameters as the number of peers, network properties – namely transmission rate, packet loss, network latency – and parallelization of computations as parameters and execution time, CPU cycles, memory consumption and amount of transmitted data as variables.

Our insights on the resource consumption show that such a solution is practically applicable in intranet environments and – with limitations – in Internet settings.

Index Terms—Cryptography, Secure Multiparty Computation, Privacy, Performance, Resource Consumption, Measurement

I. INTRODUCTION

While the foundations for secure multiparty computation (SMC) were laid about forty years ago [1], the topic experienced a revival in the last decade: Starting as mere theoretic considerations, improvements in hardware performance made practical implementations and productive use of SMC possible. In consequence, a number of SMC frameworks emerged and its practical application was considered in research [2]–[5].

Their use cases have in common that they focus on singular events of orchestrated or manually triggered computations. With Smart Buildings and the Internet of Things (IoT), a new type of use case for privacy-preserving data processing becomes relevant: Regular and automated processing of data streams will be carried out on commodity or even low-end hardware. Here, SMC can be the distributed system of choice for performing privacy-preserving aggregation of distributed data. But this is only the case when the environmental constraints do not render its application infeasible: Host nodes will only have a low amount of memory and a constrained CPU in terms of frequency and number of cores. Furthermore, communication might happen via wireless LAN or even between different regions over the Internet. Then, network

performance is restricted in terms of limited transmission rate or high network latency and presence of packet loss.

It is hence vital to understand the resource requirements of a productively usable SMC solution. Our work provides these insights by performing a thorough performance evaluation of a selected SMC framework based on secret sharing assessing the influence of a multitude of parameters on variables quantifying host (CPU and memory), network (transmitted data) and user resources (time) alongside with the identification of critical scaling behavior.

The remainder of the paper is structured as follows: In Section II, we give an overview of SMC in general and argue for the framework we select for further examination. Section III presents the related work regarding practical evaluation of SMC. We present preliminary theoretical performance considerations for round based SMC protocols in Section IV. Section V contains the description of our evaluation setup; the results are presented and discussed in Section VI. We elaborate the practical implications in Section VII and conclude our paper with Section VIII.

II. SECURE MULTIPARTY COMPUTATION

Secure multiparty computation enables multiple communicating parties to collaboratively compute a function while being able to keep their respective input value completely confidential. Yao initiated this field of research by presenting the Millionaire’s Problem and the idea of Secure Function Evaluation [1][6]. While many single purpose protocols were proposed, the main interest was in the creation of a general purpose framework which allows the computation of arbitrary functions. Basic concepts were identified which allowed approaching this aim, most notably *garbled circuits* [6], *homomorphic encryption* [7] and *secret sharing schemes* [8] [9]. Its theory flourished early in the 80’s (cf. [9]–[14]) while implementations have only been developed in the last decade. Among them, many have been proposed as proof of concept but were not publicly available [2] or have not been developed further since then [3] [15] [16]. Currently, Sharemind [17], SPDZ-2 [18] [19] and FRESKO [20] constitute the state-of-the-art of actively developed SMC frameworks¹.

¹There are further frameworks for the special two-party case, but they are not applicable in this multiparty context.

This work has been supported by the German Federal Ministry of Education and Research, project DecADE, grant 16KIS0538.

a) *Sharemind*: Sharemind [17] was mainly developed by Bodganov [21]. It is implemented in C++ and uses an additive secret sharing scheme in the ring $\mathbb{Z}_{2^{32}}$. Its focus was to provide a real-world suitable framework with appropriate performance. They therefore opted to prevent only passive corruption – which is less computationally expensive – and to constrain their solution to strictly three computation parties, while allowing an arbitrary number of input parties. Their argument is that further computation parties increase the communication overhead. Sharemind is now part of the services that Cybernetica [22] provides. It is under active development but partially closed-source.

b) *SPDZ-2*: SPDZ-2 [18] [19] is currently developed by the University of Bristol. The software is mainly written in C++ while the protocols can be written in Python. It features the SPDZ protocol, which follows the current research direction of using additive secret sharing and performing a (computationally expensive) preprocessing phase in order to gain highly efficient protocol executions.

c) *Framework for Efficient Secure Computation*: FRESKO [20] is developed by the Alexandra Institute in Denmark, a non-governmental organization for IT innovation and IT research. It aims for being a non-prototypical, productively applicable generic SMC framework written in Java. I.e. it is desired that the framework provides an abstraction from specific SMC primitives so that protocol specification can be performed independently. The benefits are that primitives can be switched afterwards while the specified protocol does not have to be changed. This especially enables simple incorporation of newest research results on SMC.

Currently, they support the Ben-Or–Goldwasser–Wigderson (BGW) [9] protocol based on secret sharing using polynomials and the computation (“online”) phase of SPDZ [18] which has been successfully applied in [23]. A full support of SPDZ is work in progress.

All of these solutions are secret sharing based. Hence, a similar performance behavior depending on the investigated parameters can be expected. However, for our use case we need a solution which is able to support computations with a theoretically arbitrary number of participants. This is not given by Sharemind. Furthermore, Sharemind is closed-source which further obstructs assessment. SPDZ-2 is currently still work in progress on a level of fundamental changes and consequently not ready for a thorough performance measurement. Our choice is therefore FRESKO, which aims for production-ready application.

III. RELATED WORK

The newly gained interest in SMC during the last years resulted in a multitude of publications, which propose successive improvements or applications of established approaches. By contrast, the body of research is missing thorough performance measurements of SMC solutions.

Most of these publications do not provide performance data or merely a single result for their exact setting of application [2] [5] [24] [25]. Others typically only evaluate overall

execution time and to some lower degree transmitted bytes measured while at most varying the number of parties and the amount of input data [3] [4] [17] [26]–[30]. Only few include further parameters like the transmission rate [19] and technology-dependent factors like circuit size and depth [16] and evaluate further parameters e.g. throughput.

For understanding whether SMC is also feasible in distributed systems, fog computing and the Internet of Things, it is necessary to perform more thorough measurements including further factors. It is vital to understand the influences of the network characteristics and to further examine the impact on host resources, i.e. CPU utilization and memory consumption.

We aim to provide the necessary insights by assessing the parameters *number of peers*, *transmission rate*, *network latency*, *packet loss*, and *input data parallelization* while measuring the variables *execution time*, *CPU cycles*, *heap memory consumption*, and *transmitted bytes*.

IV. PRELIMINARY EXECUTION TIME CONSIDERATIONS

A possible theoretic computation model for secret sharing based SMC protocol foundations like BGW “is a complete synchronous network of n processors” [9]. The protocol itself, common to all processors, is dissected into rounds. “In one round of computation each of the players can do an arbitrary amount of local computation, send a message to each of the players, and read all messages that were sent to it at this round” [9]. In secret sharing based protocols, such a message typically contains a share of a private local value – e.g. a polynomial in the BGW protocol – held by the sender.

Considering the aforementioned rounds as a time factor, the protocol becomes an alternating sequence² of local computation and network communication:

$$comp_1, comm_1, \dots, comp_{m-1}, comm_{m-1}, comp_m \quad (1)$$

Furthermore, the communication steps are synchronization points for the players, as they typically need the shares of the other participants in order to proceed with the next round. We denote the costs in terms of time for a step $comp_i$ as $cost_{comp_i}$. The message sent from player P_k sent to P_l during $comm_i$ is referred to as $msg_{i,k \rightarrow l}$.

Two phases are typically common to all SMC protocols: During the *input phase* the own private input is transformed into shares and distributed among the participants. This takes one round. In the *output phase* the shares of the computed result are exchanged among all participants, so that each is able to recombine them and to obtain the plaintext result. In FRESKO this also takes a round³.

Regarding the basic arithmetic operations BGW provides, the round complexity varies. Addition is “free” as it does not

² We consider recombining the shares to be the last step $comp_m$. Hence, there are only $m - 1$ communication steps.

³ Some solution perform a resharing in order to make the final shares independent from the shares obtained in the computation. This is, e.g., necessary when the shares should be reused to perform further calculation. Then, another round becomes necessary during this phase.

need any communication. Multiplication requires rerandomization of the polynomial and the reduction of its degree [9]. This involves a step of communication between the participants, and hence, requires a round.

Theoretically, the communication cost of the i th round $cost_{comm_i}$ depends on the number of messages sent during the round. As every participant sends an individual share of a polynomial to every other participant during communication steps, the overall number of shares sent is $\mathcal{O}(n^2)$. Furthermore, every participant p_i typically contributes its own input v_i for the computation. Hence, when a single multiplication step is specified in the protocol, this means that the product of all input values should be computed: $\prod_{i=1}^n v_i$. In such a case, $n - 1$ single multiplication rounds are necessary; consequently the costs for such an array multiplication are $\mathcal{O}(n^3)$.

These theoretical costs assume a sequential execution of each communication. However, inspection of the FRESKO code and the analysis of its behavior show that sending and receiving for every participant can happen in parallel⁴: Sending is a non-blocking action for the computation layer which hands over the messages to be sent to the communication layer of FRESKO. Receiving is actually blocking on the computation layer, however, the communication layer is nevertheless able to receive all available messages simultaneously. In other words, waiting times for receiving multiple shares are not strictly additive.

When a host has sent out every share and it has received all other participants' shares, the next computation step can be performed. So, in spite of the aforementioned theoretical complexity, due to parallelization the overall communication cost per round mainly depends on the pair of hosts, where communication takes longest:

$$cost_{comm_i} = \max_{1 \leq k, l \leq n} cost_{msg_{i,k \rightarrow l}} \quad (2)$$

While every round is practically performed in constant time, the number of rounds per array multiplication increases linearly.

A further approximative simplification of the communication costs can be made: Communication between two peers is always identically structured and bears shares as content. We could verify this claim using the FRESKO code, which specifically only sends instances of the single class which represent the shares. Hence, we can simplify that

$$\forall i \in \{1, \dots, m - 1\} : cost_{comm_i} = cost_{comm} \quad (3)$$

Note that Equation 3 does not hold for computation steps, as each phase performs different tasks.

Combining Equations 1 and 3 the overall costs of time can be estimated by

$$cost_{overall} = \sum_{i=1}^m cost_{comp_i} + (m - 1) * cost_{comm} \quad (4)$$

⁴One exception is the initial input sharing phase. Here, sending of shares is only performed by a single host at a time.

Using the model of the alternating sequence, two types of influences on the duration become visible: The computation performance depends in the properties of the participants, the communication performance depends on the properties of their network links. Due to the synchronizing behavior of rounds, the costs of both sides add up to the overall costs.

In the first part of our measurements (Section VI-A to VI-C), we focus on how the overall costs are influenced by an increasing number of participants as well as network parameters, namely network latency, transmission rate and packet loss. Besides duration measurements, we also assess the memory footprint, the CPU utilization and the amount of data transferred. In the second part (Section VI-D), we examine whether the described sequential processing and the resulting additivity of communication and computation costs can be circumvented. Therefore, we regard cases, where multiple individual computation sessions can be parallelized instead of being executed sequentially. We show that the amortized costs per session decrease as a consequence.

Performance Comparison

Conceptually, SMC replaces a Trusted Third Party (TTP) by providing a secure protocol implementation. Canetti [31] used this understanding to propose a now well-established method to prove secrecy and correctness of an SMC approach.

We can also apply this understanding to assess the performance penalty that SMC introduces. The ideal world which uses a TTP for computation can also be used as a performance baseline. In fact, in today's productively used systems, TTP solutions are the established standard; hence, the comparison with a TTP is also practically relevant.

In order to do so, we align the necessary actions when using a TTP with the phases of an SMC computation. In a TTP setting, the input phase can be understood as providing the input data to the TTP. The output phase, in turn, comprises sending the result from the TTP to the participants. Computation steps themselves can be directly adapted. The whole comparison applied to the BGW protocol is shown in Table I.

Presenting our results in section VI, we add – where applicable and foreseeable – an estimation how a TTP solution would perform. In these cases we approximate the communication performance as described before while neglecting the comparatively low influence of the computation steps.

V. EVALUATION SETUP

In the following we describe our test setup. The use case explains which computations have been carried out via SMC. Here, we refer to a real-world use case performed at our lab; the functionality is however similar to other real-world systems. In the second part, the methodology, we document the measurement environment in terms of used software, hardware and measurement tools.

A. Use Case

Our use case is inspired by MearDroid [32], a smartphone app which allows users to gain insights in the sensor data of

Phase	Computation per host	SMC		TTP	
		Communication (overall)	Communication (overall)	Computation on TTP	Communication (overall)
Close	Generation of polynomial, calculation of n shares	$n^2 - n$ messages	—	—	n messages
Addition	$n - 1$ additions	—	—	$n - 1$ additions	—
Multiplication	$n - 1$ multiplications, Comp_{Close} , Comp_{Open}	$n^2 - n$ messages	—	$n - 1$ multiplications	—
Open	Lagrange interpolation	$n^2 - n$ messages	—	—	n messages

Table I: Performance comparison SMC vs. TTP

their smartphones and allows comparison to other users. We assume a set of moving devices. One property of interest is the summed and averaged travel distance over the set of devices.

Insecurely and without SMC, the functionality is realized as follows: Each client is able to derive a stream of distances from the raw GPS coordinates. They can connect to a common central trusted server. Upon each connection the client transmits the travel distance since its last connection. These distances are collected as a running average. At any given point in time the overall average distance can then be computed by the server.

In order to apply FRESKO, the input has to be organized in synchronous sessions. In every session, each device contributes its distance since the last session, whereas the statistics server inputs the current value of the running sum (starting with 0). The result of each round is saved by the statistics server.

Knowing that communication between the peers is the typical bottleneck for SMC [4] [5] [26], the choice of the use case is beneficial for our performance measurement: The computational part is comparatively low so that performance effects caused by communication and their relationship to the named parameters become clearly visible. This allows better assessment of the communication bottleneck of SMC based solutions with negligible influence by the local computations.

Input Data: We used real world data retrieved from MeasrDroid, yielding five traces consisting of 20000 GPS tuples each, being collected in intervals of 15 to 20 minutes depending on the individual configuration per device. The utilization of more nodes for some measurements made it necessary to provide more input data to be used for the computation. The data itself does in no way influence the performance of the system. Hence, without loss of application and closeness to reality we took our data from the original 5 donors and duplicated the inputs until every used node had an own list of GPS input tuples.

B. Methodology

We evaluated FRESKO in the following setting:

1) *Hardware and Host Setup:* For our tests we had 15 physical hosts available. Each host has an Intel Xeon CPU with eight cores at 2.50GHz and a cache size of 8192KB. They have 15.780MB of RAM each and a 1 Gbit networking interface. They are arranged in the shape of a star topology, all hosts are connected via a single switch. The default link latency is around 0.18ms and there is no packet loss. The test hosts use Debian Jessie (8.5) and a 3.16 Linux kernel. The source code is compiled to a Java application which is in turn executed by the Java VM from the OpenJDK 1.8.0_111. For some tests simulating intranet, Internet and mobile Internet

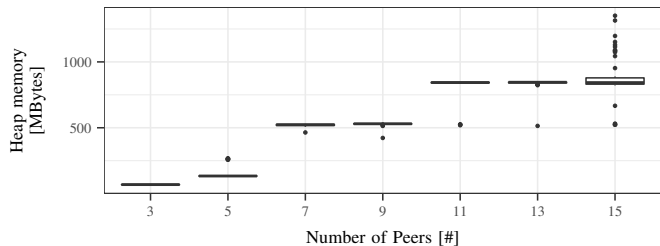


Figure 1: Impact of the number of peers on the maximum allocated heap memory

settings we added an artificial delay of 16 ms, 50 ms, 200 ms and 500 ms to the communication round-trip equally distributing the delay to both hosts of each link using τ_c . For this purpose, τ_c delays every outgoing packet by the half of the desired additional delay. Packet loss is also simulated via τ_c .

2) *Software Setup:* We use an orchestration layer which configures all client-side parameters (latency, transmission rate, packet loss, ...) and then starts the target application. The Java application itself locally loads GPS coordinates in order to perform 1000 executions of the protocol per measurement. This repetition makes the measurement results more robust against random performance fluctuations during single computations. Furthermore, each measurement itself has been repeated 50 times if not noted otherwise.

3) *Measurement Software:* Profiling is performed using *perf* from the linux-tools (version 3.16+63) for counting CPU cycles, *BTrace* (version 1.3.8.3 (20160926)) for assessing memory consumption and execution time and *tshark* (version 2.2.4) from wireshark for collecting the raw transmitted data.

VI. RESULTS

In the next subsection we focus on the host resources *heap memory* and consumed *CPU cycles*. Afterwards we analyze the amount of *transmitted data* representing the network resource. Then the *execution duration*, most directly affecting the user, is discussed. As the last subsection, we analyse the influence of parallelization of computations on the named resources.

A. Host Resources

1) *RAM:* In our context, RAM is separated in stack and heap memory. Our measurements showed that stack memory always ranged from 16 MBytes to 20 MBytes. We consider these variations to be negligible. Therefore, we focus completely on heap memory consumption in the following.

Our baseline execution with 3 peers and a setup as described in Section V-B1, the standard memory consumption is around

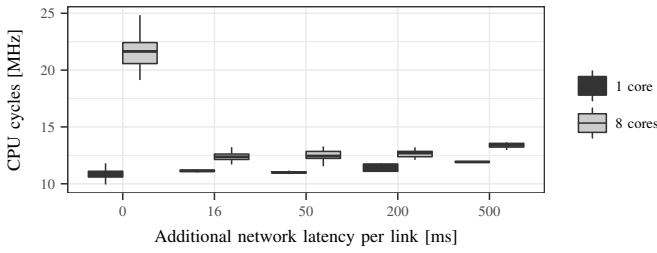


Figure 2: Impact of network latency on the CPU cycles

69 MBytes. This is only negligibly influenced by networking parameters. The delay in execution caused by a lower transmission rate, higher network latency or packet loss typically influences the speed of memory allocation and in consequence the garbage collector. This yields variations up to ± 3 MBytes.

We identified a strong correlation when scaling the number of peers. Increasing this parameter, heap memory consumption gradually diverges step-wise (cf. Figure 1). The FRESKO application uses around 70 MBytes during a computation with 3 to 5 peers, which increments to 530 MBytes for 7 to 9 peers and increases again to 840 MBytes for 11 to 15 peers. This is expectable as data about current connections as well as intermediate results like the shares of all other participants are stored on the heap. We deduce a linear trend from Figure 1 where the notable amount of outliers at $x=15$ already foreshadows the next step of heap increment. In any case, this factor rapidly becomes critical: With 15 peers, FRESKO already starts exceeding the memory resources of a Raspberry Pi [33] 3 B (1 GByte RAM) and uses a considerable amount of the memory of a current smartphone, where typically 2 GByte to 4 GByte are available for the whole system and all concurrently running applications.

2) *CPU Cycles*: During the CPU measurements we noticed that there is a major difference in the number of consumed CPU cycles when comparing a fixed node (in our case, node 2) with the last node (having the highest ID) in the set of participants. This difference is not an effect of the actual computation, but the reason is rather found in the setup phase of FRESKO. The initial step before coordinating the computation, the hosts have to establish connections with every other participant. This is achieved by every application listening for incoming connections and performing own connection attempts to other hosts in parallel, driven by busy waiting.

During our measurements, the application was started on all hosts with increasing ID, always having a little delay between the invocations. Due to this reason, that phase exhibits a specific pattern: The first application starts to poll for all other hosts which are not yet listening for incoming connections. This requires a notable number of CPU cycles. When the second application comes up, it immediately connects to the first host due to one of its connection attempts. From this point in time, both hosts poll in order to connect to all other hosts.

In consequence, the first hosts performs most polling while waiting for not yet started participants, while the last host needs only a comparatively small amount of TCP SYN at-

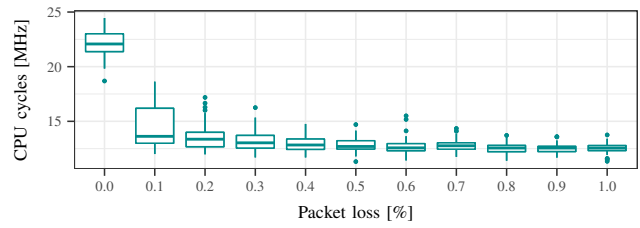


Figure 3: Impact of packet loss on the CPU cycles

tempts before all other hosts connected to it, wasting much less CPU cycles. This understanding is necessary to interpret our results.

Our baseline is around $21.5 \cdot 10^9$ cycles for the first peer node and $16.5 \cdot 10^9$ cycles for the last node. When reducing network performance as described in Section VI-A1, consumption drops to approximately $12.5 \cdot 10^9$. This effect is best depicted in the cases of network latency (Figure 2) and packet loss (Figure 3) and can be attributed to the previously explained startup phase: Impeded transmission adds another constrain on the polling which in turn becomes slower and less CPU intensive.

Additionally Figure 2 shows a slight increase in CPU cycles when increasing the network latency further. As the number of instructions did not increase during the same measurements, we expect this effect to be caused by IO waiting time during the delayed protocol execution.

On the side of number of participating nodes, the number of consumed CPU cycles depends strictly linear on it. For the first node we get (MSE⁵: 2.9451)

$$(5.16 + 5.83556 * n) * 10^9$$

and for the last node (MSE: 1.74056)

$$(15.263 + 0.69823 * n) * 10^9$$

We see that the amount of CPU cycles used in the startup phase heavily outweighs the increase of participating nodes.

B. Network Resources

Our baseline of transmitted data for three peers is 5.35 MBytes per peer.

We identified that the amount of transmitted data per peer varies around 400 KBytes upon network changes. By package inspection a common reason could be found in the network communication behavior of FRESKO:

The communication layer of FRESKO on the host of sender s receives and buffers a serialized object $o_{r,1}$ from the computation layer to be sent to recipient r . The actual transmission of $o_{r,1}$ happens in the moment when r is prepared to accept the data. However, this action does not block on the sender side. I.e. if the sender itself does not have to wait for any further incoming data from other peers, it can proceed with the next computation step immediately. Here, it can already

⁵Mean squared error

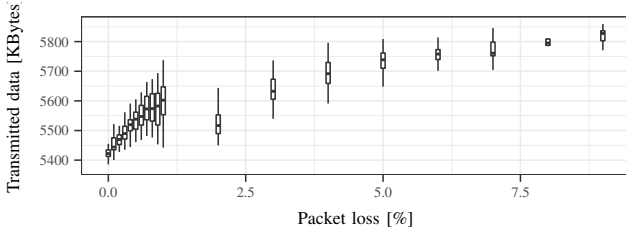


Figure 4: Impact of packet loss on the transmitted KBytes

prepare the next step of communication, including to create and prepare the next object $o_{r,2}$ to be sent to the same recipient. Given the recipient did not request $o_{r,1}$ at this point in time, the communication layer will combine $o_{r,1}$ and $o_{r,2}$ into a single message, which is then sent to r when its request happens. Combination of packets results in a reduction of the overall amount to be sent by reducing the absolute number of necessary packet headers. It is coincidence that this effect is most useful in environments with constrained transmission, where it also naturally happens most often.

The measurements of two network parameters reflect this behavior up to some degree: When reducing the transmission rate to 1 MBit, a drop to 5.10 MBytes can be detected. A similar behavior occurs when adding artificial network latency, however, without a distinct trend.

While these deviations undercut the baseline, packet loss yields an increase of the amount of transmitted data (cf. Figure 4). This behavior is expected as packet loss requires retransmissions. With a maximum of 10% packet loss, transmitted data was increased by approximately 400 KBytes.

Regarding the number of peers, the number of messages to be exchanged between all peers depends quadratically on it. Our measurements support this by showing that the amount of transferred bytes between a pair of hosts increases linearly. In our setting the increase follows the following regression line (MSE: 0.03332):

$$(-2.743 + 2.69419 * n) \text{ MBytes}$$

In other words, for each peer approximately 2 MByte of additional data is transmitted *per host*.

C. User Resource: Time

The computation duration is the most interesting variable from the user perspective. Our baseline is 5.35 seconds for 1000 calculations, i.e. each computation costs around 5 ms, whereas the startup of the Java Virtual Machine is not included.

We can see that time is heavily and differently influenced by the evaluated parameters: The increase in time is strictly linear when adding more participants. At first, this might surprise as the exchanged messages between all participants increase quadratically in their number. However, in Section IV we already elaborated how parallel execution of communication can reduce the complexity by n . As a regression function (cf. Figure 5) we yield (MSE: 0.24894):

$$(-1.086 + 2.01883 * n) \text{ ms}$$

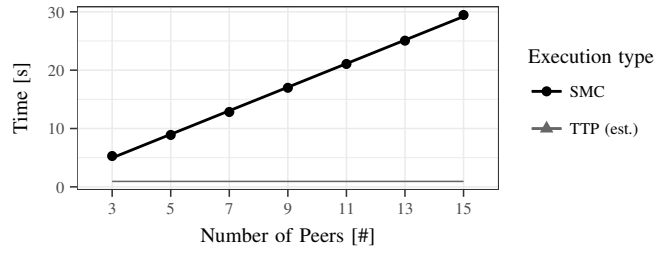


Figure 5: Impact of number of peers on the execution time

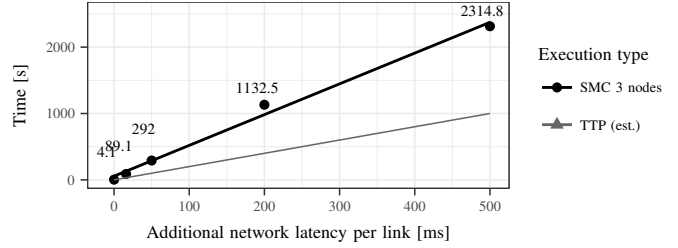


Figure 6: Impact of network latency on the execution time

As comparison, the communication delay of a TTP solution does not notably depend on the number of participants, given that sending and receiving messages can happen in parallel.

Network latency also causes a linear increase. This directly corresponds with the intuition that every message is delayed by a constant factor. However, the influence is notably stronger in absolute terms. The following regression function (cf. Figure 6) holds for three participating peers (MSE: 15415.50432):

$$47.327ms + 4.61851 * network \ latency$$

Execution inside an intranet takes around 4 seconds for 1000 sequential computations. When communicating via the Internet (50 ms to 300 ms), the computations already cost 5 to 25 minutes. The magnitude of the duration can be roughly estimated as follows: During the input phase with $n = 3$ participating hosts, $n * (n - 1) = 6$ messages have to be exchanged. Each participant sequentially waits for $n - 1 = 2$ messages from the other parties. The performed addition operation is free of communication. During the output phase, again 6 messages have to be exchanged, but this time waiting is performed in parallel⁶. Hence, as an estimate in our setup, every participant sequentially waits for $n = 3$ messages, which can consist of one to two packets each. A message of one packet costs a single network delay. A message of two packets costs three times the network delay as the second packet is only sent when the sender has received an acknowledgement message from the recipient. In consequence, we gain an interval of $[n * network \ latency, 3n * network \ latency]$ per protocol execution.

Utilizing a basic TTP solution, all hosts send their data during a single network delay. The computation itself is performed locally. At the end another network delay is added for sending the results to all participants (in parallel). While it

⁶Using Equation 2 we count this as a single message.

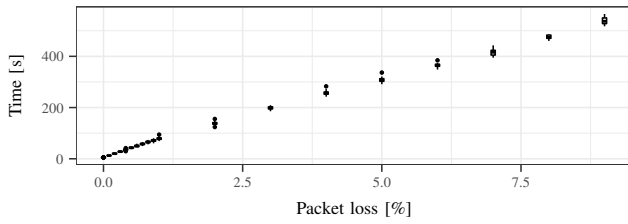


Figure 7: Impact of packet loss on the execution time

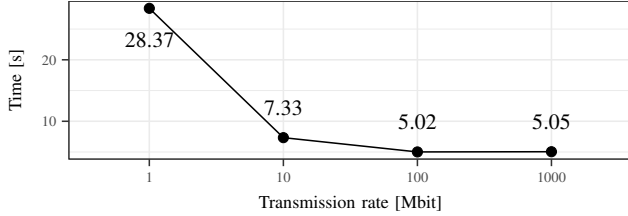


Figure 8: Impact of transmission rate on the execution time

seems that the performance of the SMC solution is acceptably worse in comparison, it is important to note that the duration of the SMC session scales proportionally with increasing number of peers, while the TTP does not depend on this factor (cf. Figure 5).

When packet loss occurs, repeated retransmissions become necessary. Due to this, we expect the execution time (Figure 7) to constitute a geometric row and to increase hyperbolically in the interval $[0, 1]$ with increasing packet loss probability p_{loss} . One would expect the same characteristics for the execution time. However, the steep increase only happens very late when p_{loss} is near 1. The analyzed interval from 0% to 10% is at the beginning of the function’s domain, where only a linear increase becomes visible. The sessions started failing due to timeouts at a packet loss rate of 10%.

Comparatively weak constraints are given by the transmission rate (cf. Figure 8). A very low rate of 1 MBit does influence execution time negatively, but already between 10 MBit and 100 MBit all rate-induced impediments are resolved.

Inspection shows that a transmission consists of sending a share from one host to another. This encompasses one to maximally two packets each having only a length between 100 and 1000 Bytes. This is the reason why network latency has stronger influence than the transmission rate.

In conclusion, each single computation has a low duration; the overall duration increases linearly with the number of peers. While this influence is comparatively small, the network parameters have the highest influence on the execution time. In the ranges of the practically relevant intervals we saw that the transmission rate can influence the execution time by factor 5, packet loss has an influence up to a factor of approximately 110 and network latency can slow down the computation even by factor 550. These impediments already occur at network configurations which are realistic on the Internet or on the mobile Internet at least.

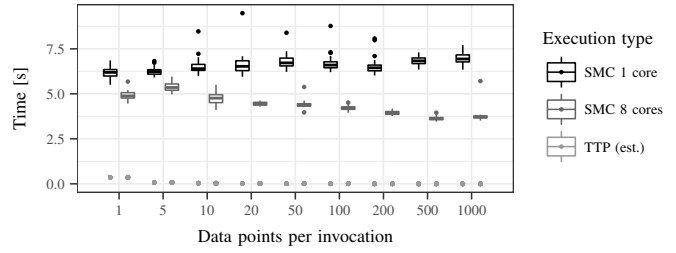


Figure 9: Impact of network latency on the execution time depending on parallelization with 0ms additional latency

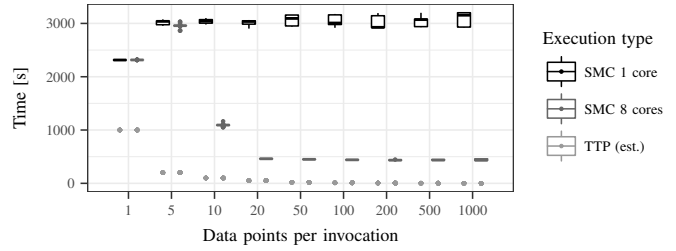


Figure 10: Impact of network latency on the execution time depending on parallelization with 500ms additional latency

D. Parallelization

Two potential bottlenecks can exist (cf. Section IV): The computation steps are constrained by the host and the communication steps by the links between them. Due to their strict sequential execution, the occurring delays are additive.

This situation can be generically improved in certain scenarios: In our test setup we performed 1000 calculations sequentially. When a given number of input data is known in advance, multiple computations can be performed at the same time. FRESKO provides a `ParallelProtocolProducer` which allows combining subprotocols so that they are executed in parallel using individual threads. Parallelization was parametrized with the values $\{1, 5, 10, 20, 50, 100, 200, 500, 1000\}$.

Our initial finding is that the benefit of parallelization depends on the size of the delay introduced by host or network parameters. Figure 9 shows that in our default setting parallelization reduces an initial duration of 4.910 seconds with $pf == 1$ in average to 3.631 with a $pf == 500$, a reduction by 26%. This improvement depends on the availability of multiple CPU cores. Otherwise, parallelization leads to a slight rise in execution time due to its organizational overhead.

However, when examining cases with e.g. higher network latency (Figure 10), a higher reduction can be achieved. An initial duration of 2315 seconds is reduced to 435 seconds when $pf == 200$, a reduction by 81%.

Investigation of the code shows that the network layer does not combine messages of different but simultaneously performed sessions; they are sent via different “channels”. Instead, this layer processes the messages to be sent in a strict sequential fashion. This means that the measured improvements result from the time-multiplex utilization of the networking layer: During sequential execution of subsequent computations, all other parties often wait for a single party

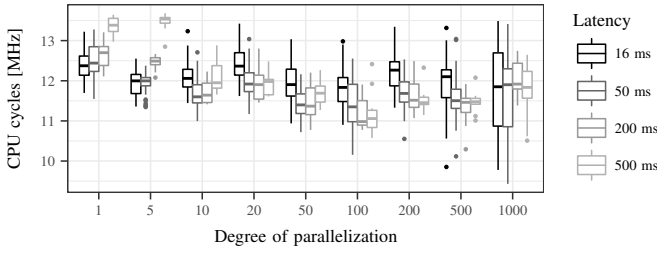


Figure 11: Impact of parallelization on the CPU cycles

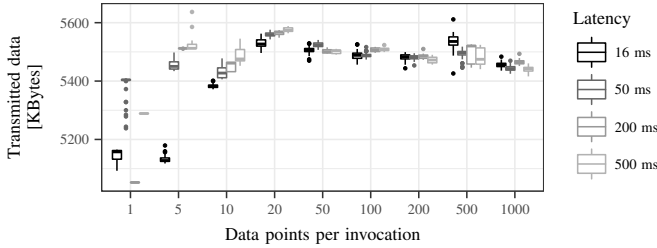


Figure 12: Impact of parallelization on the transmitted bytes

to perform its computation and sending of shares. With parallelization, we achieve that every participant is kept busy on the communication part – the bottleneck – all the time.

When employing parallelization on a basic TTP solution a reduction of the *actual communication overhead* is possible. Input data of multiple computations can be combined into a single (or a low amount of) packets. In consequence, the impact of network latency can be highly reduced.⁷

Figure 11 shows the consumed CPU cycles depending on the degree of parallelization with non-zero additional network latency and 8 available cores per host. Parallelization does not impose notable penalties on the hosts’ CPU. Correspondingly, as the execution duration is significantly reduced, the actual degree of CPU utilization during that time increases.

Regarding transferred data, an increasing degree of parallelization does not lead to a corresponding increase (cf. Figure 12). Instead, it is still bounded by approximately 5.7 MBytes.

Figure 13 shows that parallelization leads to a late and not completely clear trend of memory increase when $pf \geq 50$. This could extend to higher degrees of parallelization. However, the benefits of parallelization are achieved with a much lower pf than where memory consumption starts to increase gradually. That means a sweet spot of beneficial parallelization without memory penalties should be generally identifiable. Nevertheless this is a trend where further investigations could provide more insights on whether memory consumption is bounded or not.

In conclusion, we could show that parallelization is able to reduce the computation duration approximately by factor 5. Therefore, computation of 20 items in parallel is sufficient and already exploits the full parallelization potential. Further increase of the parallelization factor did not yield notable

⁷In Figure 10 we show an estimation of the TTP performance while neglecting that a high parallelization rate makes it necessary to split the packets again for transmission.

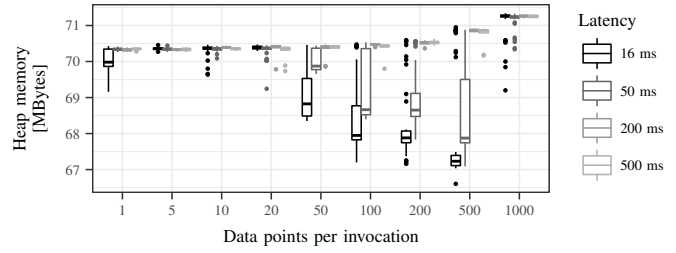


Figure 13: Impact of parallelization on the maximum allocated heap memory

improvements. However, different types of computations yield protocols which differ in their ability to be parallelized. Hence, the parallelization factor should be newly determined every time a new protocol is used. Finally, while parallelization yields a notable improvement on execution cycles, it does not cause any changes regarding the CPU cycles. This shows that parallelization can effectively be used to utilize all available CPU resources. Concomitantly, the former sequential execution, creating two alternating bottlenecks – the CPU and the network – is changed to a parallel execution of both phases (with respect to different computations). In consequence, only the stricter of both bottlenecks constitutes the limiting factor instead of their sum. In any case, multi-core hosts are necessary in order to leverage parallelization advantages at all.

VII. PRACTICAL IMPLICATIONS

Our results show that FRESKO as an implementation of SMC possesses a performance and resource utilization behavior which allows practical application: In the setting of an intranet, computations are efficiently performed. The execution time is around 2 to 3 ms per session and peer. This allows batch processing of data and interactive use cases. Performance might, however, not be sufficient for the realization of real-time applications depending on the computation to be carried out. Regarding the hosts systems, multiple cores are necessary when parallelization can be utilized. In other cases, secure computation should also be feasible with weaker devices. Memory consumption can become critical when a multitude of peers participates in the computations. This must be considered upon productive use. However, regarding all identified performance results, we deem the memory consumption to be more related to Java than to secret sharing or SMC in general. Having a setting of memory constrained devices, a more economical programming language would be more appropriate.

In wide area networks as the Internet and possibly mobile Internet, network latency is the most influential constraining factor. Execution time degrades strongly with increasing latency. In these contexts, we currently only see batch processing as a use case: Given it is acceptable to wait several minutes for a computation result, SMC can be utilized. However, in this context it is more likely that parallelization can be applied, which decreases the latency penalties to some degree. Further improvement of the situation would require to reduce the amount of transmitted packets. This could be possible

by stricter orchestration of computations running in parallel, where packets between different peers would be used for multiple sessions simultaneously. On contrary, our current solution applies parallelization which does not enforce message combination, but only enabled waiting times per host to be used for further computations.

VIII. CONCLUSION

This paper presents the results of thorough measurements to assess the fundamental practical applicability of secure multiparty computation (SMC) in real-world contexts.

We show how SMC sessions can be understood with regard to performance as a alternating sequence of local computation and communication between participating peers. This yields practical implications, bearing in mind that both types have their own individual bottleneck: Typically, their delay is strictly summative during a single execution.

In our measurements, we examine how network latency, transmission rate and packet loss, as well as the number of peers influence the execution time, the CPU utilization, memory allocation and the amount of transmitted data. Furthermore, we analyze whether parallelization of formerly consecutive sessions can overcome the additivity of delays.

Interpreting our findings, we conclude that SMC is practically applicable with weak limitations in intranet settings. Here, requirements for participating host systems are in ranges of today's commodity hardware. Furthermore, SMC seems to be applicable to some (lesser) degree in Internet settings. Here, network latency has the biggest negative influence on performance. However, as performance of SMC protocols continues to increase, we expect that feasibility of SMC over the Internet will also improve in the next years.

IX. ACKNOWLEDGEMENTS

We would like to thank Daniel Raumer and Florian Wohlfart for their valuable feedback on the initial versions of the paper.

REFERENCES

- [1] A. C. Yao, "Protocols for secure computations," in *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*. Washington, DC, USA: IEEE, 1982, pp. 1–5.
- [2] P. Bogetoft, D. L. Christensen, I. Damgård, M. Geisler, T. Jakobsen, M. Krøigaard, J. D. Nielsen, J. B. Nielsen, K. Nielsen, J. Pagter, M. Schwartzbach, and T. Toft, "Secure multiparty computation goes live," in *Lecture Notes in Computer Science*, vol. 5628 LNCS, 2009, pp. 325–343.
- [3] M. Burkhart, M. Strasser, D. Many, and X. Dimitropoulos, "SEPIA: Privacy-preserving Aggregation of Multi-domain Network Events and Statistics," *Proceedings of the 19th USENIX Conference on Security*, p. 15, 2010.
- [4] M. Zanin, T. T. Delibasi, J. C. Triana, V. Mirchandani, E. Álvarez Pereira, A. Enrich, D. Perez, C. Paşaoğlu, M. Fidanoglu, E. Koyuncu, G. Guner, I. Ozkol, and G. Inalhan, "Towards a secure trading of aviation CO2 allowance," *Journal of Air Transport Management*, vol. 56, pp. 3–11, 2016.
- [5] D. Bogdanov, R. Talviste, and J. Willemson, "Deploying secure multiparty computation for financial data analysis," *Financial Cryptography*, pp. 57 – 64, 2012.
- [6] A. C. Yao, "How to generate and exchange secrets," in *Proceedings of the 27th IEEE Symposium on Foundations of Computer Science*. IEEE Computer Society Press, 1986, pp. 162–167.
- [7] R. L. Rivest, L. Adleman, and M. L. Dertouzos, "On Data Banks and Privacy Homomorphisms," *Foundations of Secure Computation*, pp. 169–180, 1978.
- [8] A. Shamir, "How To Share a Secret," *Communications of the ACM (CACM)*, vol. 22, no. 11, pp. 612–613, 1979.
- [9] 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)*, pp. 1–10, 1988.
- [10] O. Goldreich, S. Micali, and A. Wigderson, "How to play ANY mental game," in *Proceedings of the Nineteenth Annual ACM Conference on Theory of Computing – STOC '87*. New York, NY, USA: ACM, 1987, pp. 218–229.
- [11] D. Beaver, S. Micali, and P. Rogaway, "The Round Complexity of Secure Protocols," in *Proceedings of the 22nd Annual ACM Symposium on the Theory of Computing*, 1990, pp. 503–513.
- [12] D. Chaum, I. B. Damgård, and J. van de Graaf, "Multiparty Computations Ensuring Privacy of Each Party's Input and Correctness of the Result," in *Advances in Cryptology*, C. Pomerance, Ed. Berlin Heidelberg: Springer-Verlag, 1987, pp. 87–119.
- [13] D. Chaum, C. Crépeau, and I. Damgård, "Multiparty Unconditionally Secure Protocols," *Proceedings of the 20th Annual ACM Symposium on Theory of Computing*, pp. 11–19, 1988.
- [14] T. Rabin and M. Ben-Or, "Verifiable secret sharing and multiparty protocols with honest majority," *Proceedings of the 21st Annual ACM Symposium on Theory of Computing*, pp. 73–85, 1989.
- [15] M. Geisler, "Cryptographic Protocols: Theory and Implementation," Ph.D. dissertation, Aarhus University, 2010.
- [16] A. Ben-David, N. Nisan, and B. Pinkas, "FairplayMP: A System for Secure Multi-Party Computation," *Proceedings of the 15th ACM Conference on Computer and Communications Security*, pp. 257–266, 2008.
- [17] D. Bogdanov, S. Laur, and J. Willemson, "Sharemind: A framework for fast privacy-preserving computations," in *IACR Cryptology ePrint Archive*. Springer, 2008, no. October, p. 289.
- [18] I. Damgård, V. Pastro, N. Smart, and S. Zakarias, "Multiparty computation from somewhat homomorphic encryption," in *Lecture Notes in Computer Science*, vol. 7417, 2012, pp. 643–662.
- [19] M. Keller, E. Orsini, and P. Scholl, "MASCOT," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 830–842.
- [20] "A FRamework for Efficient Secure Computation." [Online]. Available: <https://github.com/aicis/fresco>
- [21] D. Bogdanov, "Sharemind: programmable secure computations with practical applications," Ph.D. dissertation, 2013.
- [22] "Cybernetica," 2017. [Online]. Available: <https://www.cyber.ee>
- [23] I. Damgård, K. Damgård, K. Nielsen, P. S. Nordholt, and T. Toft, "Confidential benchmarking based on multiparty computation," *Lecture Notes in Computer Science*, vol. 9603 LNCS, pp. 169–187, 2017.
- [24] C. Thoma, T. Cui, and F. Franchetti, "Secure Multiparty Computation Based Privacy Preserving Smart Metering System," *44th North American Power Symposium (NAPS)*, pp. 1–6, 2012.
- [25] P. Bogetoft, I. Damgård, T. Jakobsen, K. Nielsen, J. Pagter, and T. Toft, "A Practical Implementation of Secure Auctions Based on Multiparty Integer Computation," *Financial Cryptography*, vol. 4107, pp. 142–147, 2006.
- [26] K. Bonawitz, V. Ivanov, B. Kreuter, A. Marcedone, H. B. McMahan, S. Patel, D. Ramage, A. Segal, and K. Seth, "Practical Secure Aggregation for Privacy Preserving Machine Learning," *IACR Cryptology ePrint Archive*, vol. 2017, p. 281, 2017.
- [27] B. Pinkas, T. Schneider, N. P. Smart, and S. C. Williams, "Secure two-party computation is practical," *Lecture Notes in Computer Science*, vol. 5912 LNCS, pp. 250–267, 2009.
- [28] D. Bogdanov, M. Niitsoo, T. Toft, and J. Willemson, "High-performance secure multi-party computation for data mining applications," *International Journal of Information Security*, vol. 11, no. 6, pp. 403–418, 2012.
- [29] F. Kerschbaum, D. Biswas, and S. De Hoogh, "Performance comparison of secure comparison protocols," *Proceedings - International Workshop on Database and Expert Systems Applications, DEXA*, no. October, pp. 133–136, 2009.
- [30] F. Kerschbaum, D. Dahlmeier, A. Schröpfer, and D. Biswas, "On the practical importance of communication complexity for secure multiparty computation protocols," *Proceedings of the 2009 ACM symposium on Applied Computing - SAC '09*, pp. 2008–2015, 2009.

- [31] R. Canetti, "Security and Composition of Multi-party Cryptographic Protocols," 1999.
- [32] Chair of Network Architectures and Services; TUM, "MeasrDroid." [Online]. Available: <http://www.droid.net.in.tum.de>
- [33] "Raspberry Pi Models." [Online]. Available: <https://www.raspberrypi.org/products/>