Optimal Transport Layer for Secure Computation*

Markus Brandt[1], Claudio Orlandi[2], Kris Shrishak[3], Haya Shulman[4]

[1] markus.brandt@sit.tu-darmstadt.de, TU Darmstadt and Fraunhofer SIT, Germany
[2] orlandi@cs.au.dk, Department of Computer Science, DIGIT, Aarhus University
[3] kris.shrishak@sit.tu-darmstadt.de, TU Darmstadt, Germany
[4] haya.shulman@sit.fraunhofer.de, TU Darmstadt and Fraunhofer SIT, Germany

Abstract. We explore two central issues in the performance of Secure Two-Party Computation (2PC): (1) the transport layer in 2PC and (2) evaluation of 2PC implementations.

Transport layer. Although significantly improved, the performance of 2PC is still prohibitive for practical systems. Contrary to the common belief that bandwidth is the remaining bottleneck for 2PC implementation, we show that the network is under-utilised due to the use of standard TCP sockets. Nevertheless, using other sockets is a nontrivial task: the developers of secure computation need to integrate them into the operating systems, which is challenging even for systems experts. To resolve this issue, and break the efficiency barrier of 2PC, we design and develop a framework, we call `Transputation`, which automates the integration of transport layer sockets into 2PC implementations. The goal of `Transputation` is to enable developers of 2PC protocols to easily identify and use the optimal transport layer protocol for the given computation task and network conditions.

We integrated selected transport layer protocols into `Transputation` and evaluated the performance for a number of computational tasks. As a highlight, even a general purpose transport layer protocol, such as SABUL, improves the run-time of 2PC over TCP on EU-Australia connection for circuits with $> 10^6$ Boolean gates by a factor of 8.

Evaluations of 2PC. Evaluations of 2PC implementations do not reflect performance in real networks since they are typically done on simulated environments and even more often on a single host. To address this issue, we setup a testbed platform for evaluation of 2PC implementations in real life settings on the Internet.

1 Introduction

Secure two-party computation (2PC) is a cryptographic tool that allows two remote parties to jointly compute any function on their inputs and obtain the result without leaking any other information. As more and more interaction is performed online between parties that do not trust (or only partially trust) each other, the need for secure and practically efficient 2PC

* This work was supported by: the Concordium Blockhain Research Center, Aarhus University, Denmark; the European Research Council (ERC) under the European Unions's Horizon 2020 research and innovation programme under grant agreement No 803096 (SPEC); the Danish Independent Research Council under Grant-ID DFF-6108-00169 (FoCC); the German Federal Ministry of Education and Research and the Hessen State Ministry for Higher Education, Research and Arts within their joint support of the National Research Center for Applied Cybersecurity ATHENE; the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation): GRK 2050/251805230 and SFB 1119/236615297.

solutions is increasing, and there are plenty of applications that secure 2PC could facilitate. The classic setting where 2PC is needed is that of two parties holding some data and gaining extra utility by combining their data with the data of the other party. However, the parties might not want (or might not be allowed) to share their data with each other due to privacy concerns, competition (e.g., financial) or legislation. In these settings 2PC is a powerful tool that enables collaborations which were previously infeasible. Examples of such applications include key management for digital currencies [2], auctions [10], tax-fraud detection [9], private set intersection [16,18,35,50,60,61] and even prevention of satellite collision [38,33].

Implementations are still not practical. Despite its huge potential, aside from initial attempts and scant success stories, 2PC still remains the focus of theoretical research. Most prototype implementations are meant to demonstrate feasibility [30]. The implementations are evaluated in simulated environments or on a single host without taking into account realistic network conditions nor presence of other processes [40]. Current 2PC implementations are not generally usable and the users have to tradeoff their privacy with efficiency by resorting to third parties for performing computations for them instead of running 2PC with the target service.

Computation is optimal. While the idea of 2PC is now more than 30 years old, the first public implementation of 2PC was released in 2004 [42]. Since then, huge progress has been made in terms of protocol design and implementation engineering. The most efficient implementations of 2PC protocols are based on the protocol proposed in 1986 by Yao [58] which is built from garbled circuits [8] and oblivious transfer (OT) [44]; see background on 2PC, garbled circuits and OT in Appendix A. What has changed in recent years is that (thanks to protocol optimisations, hardware support for cryptographic operations and the use of multiple cores) the computation overhead of 2PC protocols has been reduced drastically and there are indications that the current constructions of 2PC based on garbled circuits have reached the theoretical lower bound [59]. It is now widely believed that the *bandwidth* and not the *computation*, is the remaining bottleneck in 2PC [3,59,30].

Communication is yet to be explored. Although there has been work on reducing the communication complexity and limiting the amount of data that needs to be exchanged between the two parties, no research has been done on the networking layer of 2PC. Indeed, evaluations of 2PC implementations on a single machine now result in practical performance [7,52]. Nevertheless, when 2PC is evaluated on separate hosts, the latency overhead is prohibitive for practical applications [46,54]. Evaluations of 2PC disregard the issues faced by practical deployment of 2PC: many implementations are not evaluated in real life setups where diverse network conditions, such as packet loss, latency, other traffic and other processes can negatively affect performance [40,30].

In this work we explore the transport layer in 2PC, focusing both on the transport layer in secure computation and on evaluation of 2PC implementations in realistic network setups.

Automated Transport Layer All the secure computation implementations use the standard TCP socket supported in popular operating systems (OSes). We demonstrate through our evaluations that, in contrast to other transport protocols, TCP sockets result in high performance penalty and do not fully utilise the bandwidth. TCP connections not only suffer from poor performance on paths with high latency and packet loss but they also fail to adjust to rapidly changing network conditions and incur high latency with buffer bloat. Even in stable network conditions, TCP does not provide optimal performance, and is not suitable for different types of applications. In the recent years different variants of TCP as well as other transport protocols, tailored to different applications and network setups, have been proposed.

> *Why then are other more efficient transport layer protocols not used in implementations of secure computation?*

The cause is the difficulty of integrating new transport layer sockets. As a result, the developers use the default option of TCP. In order to use other transport layer sockets the developers have to integrate them into the OS kernel: this is a challenging task even for systems and networking experts. Hence, 2PC developers resort to using the default TCP sockets, supported in the available cryptographic implementations and operating systems.

> *Which transport layer protocol is optimal for 2PC implementations?*

To answer this question we perform evaluations with popular transport protocols. Our results demonstrate that there is no general transport protocol that can provide optional performance for all 2PC implementations. For optimal performance the transport protocol has to be selected as a function of the 2PC implementation, the size of the inputs, and the network conditions (e.g., packet loss, latency).

In this work we develop a transport layer framework for secure computation that we call `Transputation`, which automates the usage and integration of transport layer protocols into secure computation implementations. `Transputation` can be continually extended with new transport layer protocols. During the computation, `Transputation` automatically identifies the most suitable transport protocol (among those that were integrated into `Transputation`) for a given computation task and network conditions. This adaptive behaviour on the transport layer allows achieving consistently high performance over different and complex real world network conditions.

Evaluations of 2PC Due to the difficulty and overhead of setting up real life testbed environments, 2PC developers typically evaluate the implementations on a single host or on simulated environments [40,30]. The performance in those setups is not representative of real networks, e.g., the developers cannot foresee behaviour during execution with other traffic and in diverse network conditions. Although there are other platforms for running experiments in distributed setups, such as PlanetLab [15], they are not specifically tailored for 2PC evaluations and do not have the support for the various transport layer protocols, nor enable automated usage of transport protocols in 2PC implementations. Hence the 2PC developers would themselves have to setup and install the requirements, including upload binaries, install libraries and dependencies, configure and integrate transport layer protocols, measure latencies and packet loss. We create a platform with automated and real life network setups for evaluation of 2PC implementations. Our platform is setup on `Vultr` cloud instances in different physical locations, over networks with varying latencies and packet loss and other concurrent traffic. By using our preinstalled implementations or by uploading binaries of their implementations, 2PC developers can perform real life evaluations and receive immediate results without the need to install or use any traffic monitoring tools.

Contributions. In this work we explore the obstacles in making secure computation practical for real life systems. We identify two central issues which need to be resolved: (1) there should be an easy way for 2PC developers to integrate and experiment with various transport protocols in their implementations of 2PC and (2) there should be an automated way to evaluate and compare the performance of 2PC implementations in different network setups.

We use our framework `Transputation` to demonstrate the performance improvements of other (even mainstream) protocols over TCP, and to motivate the importance of enabling automated and easy integration of new transport protocols into implementations of 2PC. In our experimental evaluations with `Transputation`, we consider the complexity of the Internet and test `Transputation` in large scale real world networks. For our evaluations we have integrated three transport protocols into `Transputation`: UDP, TCP and SABUL[5]. We explain the choice of the protocols in Section 4. The benefits of using versatile network protocols will be even more significant after transition to 5G networks, which guarantee 99.99% reliability (i.e., no packet loss) and high transmission rates (at least 20Gp/s), [48,53]. In such cases using TCP as the transport will miss out on the performance benefits of 5G. Nevertheless, `Transputation` will enable usage of versatile transport protocols tailored for the needs of specific applications and hence will facilitate adoption of 2PC for different systems.

Our evaluations with the selected transport protocols demonstrate that even general purpose protocols already provide significant performance improvement over standard TCP sockets. For instance, for large circuit sizes in WAN setting, SABUL performs $8x$ better than TCP (Figure 1). Of course, transport protocols tailored for specific tasks would further improve efficiency. There is a large body research of showing performance improvements over general purpose transport protocols when tailoring protocols to specific tasks and engineering patches for specific network conditions, e.g., [22,11,28,47,57,1]. Preliminaries on the transport layer protocols are given in Appendix B.

For automating the evaluations of 2PC, we develop and set up an environment which provides a user friendly GUI, that allows the users to select the network conditions as well as setup and run their 2PC implementations over the Internet with the required latency, packet loss and concurrent traffic.

Organisation. In Section 2 we introduce the `Transputation` framework and its layers. The optimizations and implementations on secure computation layer and the transport layer are explained in Section 3 and 4 respectively. In Section 5 we present the design and implementation of the `Transputation` framework that we propose. In Section 6 we present the evaluation testbed environment and in Section 7 we report on our evaluation and simulation results of `Transputation` on our testbed environment. In Section 8 we review related work and conclude this work in Section 9. Appendix A and B provide the necessary preliminaries on secure computation and transport layer.

2 Transputation Framework

In this section we provide an overview of `Transputation`, explain how it can be used by developers and describe our front end implementations which provides online access to `Transputation`.

[5] Though we use the most updated version of SABUL, also commonly known as UDT or UDP-based Data Transfer Protocol, in this paper we use the former name to stress the difference with “regular” UDP.

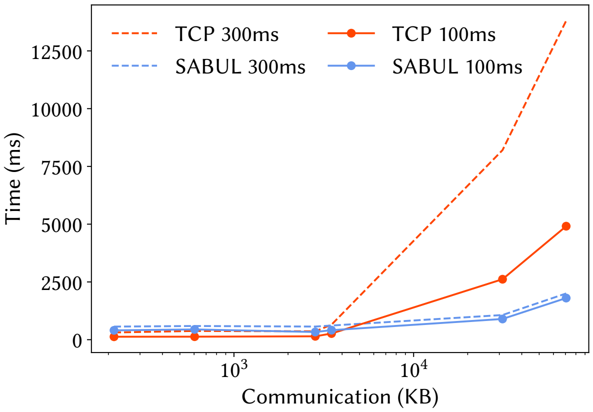

Fig. 1. TCP vs. SABUL as garbled circuit size increases.

2.1 Components

The goal of `Transputation` is to identify which transport layer protocol is optimal for a given computational task at hand, the input sizes, the network setup and buffer sizes. The functionality of `Transputation` is split between two layers: the *secure computation* layer and the *transport* layer.

Secure Computation Layer, composed of secure computation implementations, accepts the function or application to be evaluated securely and the size of the circuit representing the function. In Section 3, we motivate our choices for 2PC implementations that we integrated into `Transputation`. We identify the parameters that impact the data volume and communication pattern, and deploy optimizations relevant for performance improvements on the transport layer.

Transport Layer is responsible for identifying an optimal transport layer protocol for a given computation task. To that end, it determines the latency, bandwidth and packet loss on the network via live experimental probes. Transport layer is also responsible for avoiding overflow at the sender and the receiver. To avoid overflow, given the function to be computed and the size of inputs to the circuit, `Transputation` calculates the required buffer sizes on the sender and the receiver. We explain the implementation of the transport layer in Section 4.

We integrated, in both layers, popular and representative protocols to demonstrate the usage of framework as well as the evaluation of 2PC implementations. In the secure computation layer, we have integrated a garbled circuit protocol under standard assumptions [27], a garbled circuit protocol under circularity assumption [59] and the scheme that assumes AES as a ideal cipher [7]. In the transport layer, we support TCP [29], UDP and SABUL [24,25,26].

`Transputation` brings together the *secure computation* layer and the *transport* layer by choosing a suitable transport protocol based on the combination of parameters obtained from them. `Transputation` is built to support easy extensions at both layers. New implementations of secure computation as well as new transport layer protocols can be integrated into `Transputation`˙The flexibility for easy integration of transport layer protocols is critical – new protocols are continually devised, e.g., to improve performance of applications in diverse network conditions or to support new Internet architectures. Future protocols may be able to offer an even better performance for secure computation than the existing options. Transputation provides agility and flexibility for 2PC developers enabling them to constantly extend the implementations with new transport protocols.

2.2 Usage of Transputation

Transputation makes it convenient for secure computation developers to focus on the 2PC protocol. First, the developers can implement the secure computation protocol without worrying about the underlying transport layer protocol. Then, the developers can invoke the *transport layer* part of Transputation which is implemented as a wrapper. Our implementation provides functions for common operations such as setting up a connection between the two parties, sending and receiving data.

We implement abstract classes into Transputation that makes selecting transport layer protocols simple and automated. Adding a new transport layer protocol to the framework only requires specifying a string to identify the protocol without requiring any changes to the secure computation implementation. Without the framework, substantial parts of the secure computation implementation will need to be re-written to accommodate new transport layer protocols. In many existing secure computation implementations, the code is not modular. The network code is interwoven with the application code such that not only is the transport layer protocol hard-coded into the application code, but also the IP-address and the port numbers are hard-coded. This has also been recently observed by [30]. Unwrapping such an implementation and adding transport layer protocols requires extensive rewriting of the code base. To begin with, an entangled code base prevents retrofitting of transport layer protocols. The code base will need to be made modular before considering the integration of different transport protocols.

Secure computation developers can use Transputation in three ways. First, they can utilize the *transport layer* module to automatically choose the most appropriate transport protocol in a given network setting for their secure computation implementation and 2PC application. The secure computation implementation is then integrated with the transport layer protocol which provides the most efficient run time. Second, if the developer wants to test secure computation protocol manually, it is possible to run the implementation and 2PC application using the various transport layer protocols available in Transputation under various network conditions to choose the most suitable transport layer protocol for the 2PC application. Finally, secure computation protocols consist of interactive primitives such as oblivious transfer (OT) and OT extensions. Transputation can be used not only by developers who use these primitives to construct a 2PC protocol, but also by the designers of the primitives to test the practicality of their design.

2.3 Front End

Since the first implementation of secure computation, running and testing the implementations requires the tiresome process of downloading the source files and installing them along with the dependencies. Though this process is necessary if one wants to make changes to the code and experiment, it should not be necessary if one wants to run the code as it is without any modifications. Hence, to make it easy for users of secure computation to use Transputation, we provide a GUI.

The GUI offers two possibilities: First, secure computation protocols can be run over WAN or LAN by choosing the application and the transport protocol. Once the choices are made and the user clicks the Run icon, the experiment runs in the background and the result of the execution is displayed after the evaluation is completed. Second, GUI also supports running simulations. For the secure computation of a particular application, various network parameters—delay, loss, bandwidth, transport protocol—can be chosen. Running simulations gives the user an insight into the efficiency of secure computation protocol and allows them to compare and choose the appropriate transport protocol under specific network constraints.

3 Two-Party Computation Layer

In this section we present the secure computation layer of Transputation. We explain our choices of the selected 2PC protocols and the applications. We describe the parameters which define the amount of data to be transmitted, the properties which impact the communication performance and list recent computation optimizations relevant for our improvements on the transport layer, which we integrate into Transputation. We identify computations and operations which have most impact on communication efficiency and deploy corresponding optimizations.

Currently most efficient technique for secure two-party computation is based on Yao's protocol with garbled circuits (see preliminaries in Appendix A). The main features of Yao's protocol (as opposed to other protocols) are: it has constant rounds (in fact, it is round optimal when using a two-message oblivious transfer protocol), it is computationally cheap (since it mostly uses lightweight symmetric-key operations, and very few expensive public-key operations such as exponentiations). Moreover, due to the nowadays ubiquitous presence of the AES native instruction (AES-NI) in modern CPUs, the computational overhead of generating and evaluating garbling circuits has further decreased.

The main alternative to Yao's protocol is the GMW protocol [21] (which is purely based on Oblivious Transfer). Unlike Yao's protocol, the GMW protocol has non-constant round complexity (proportional to the depth of the circuit to be evaluated). This requirement hinders its efficiency, especially in high latency setting. In [52], the authors concluded that GMW protocol with multiple rounds was more efficient than Yao's garbled circuits in low latency LAN settings. However, the evaluations were compared to that of Huang et. al [36] from 2011. Since then, the garbled circuits protocol has been optimized further. As an additional contribution, in Appendix A.3, we show evidence that Yao's protocol is currently more efficient than GMW in all network settings. Thus, we do not consider GMW further in this work.

We focus on the recent optimized garbling schemes. The three main characteristics of such garbling schemes are: the size of the produced garbled circuits (which impacts the communication efficiency of the protocol), the number of encryptions/decryptions which have to be performed for generating/evaluating a garbled gate (which impacts the computational overhead of the protocol), and the computational assumptions under which the garbling scheme can be proven secure.

We included the most representative, with respect to efficiency-security trade-offs, garbling schemes into Transputation. All other schemes in the literature are strictly worse than the ones we consider when it comes to either efficiency or security.

Concretely, we implemented 2PC protocols based on the following three garbling schemes in Transputation:

(1) The most efficient known garbling scheme which only makes very conservative and standard assumptions (namely, that AES is a PRF), proposed by Gueron, Lindell, Nof and Pinkas [27], henceforth GLNP15;

(2) A more communication efficient garbling scheme which supports free-XOR [39] and half-gates [59] (at the price of having to assume circular security properties of AES);

(3) A further improvement in the computational overhead at the price of having to assume that AES with a fixed-key behaves like an ideal permutation [7];

In our evaluations with `Transputation`, we extend upon the work of GLNP15 [27] to show that with a better selection of transport layer protocols, secure computation with standard assumptions can be made more efficient. More precisely, we show that the use of TCP limits the performance of secure computation protocols in a high latency setting when the communication is large.

3.1 Garbling Schemes

In this section we give an overview of the different garbling schemes that have been implemented in `Transputation`. We identify the computations and operations which have impact on the communication efficiency and explain the corresponding optimizations that we deployed (which, as we explain, do not sacrifice security). We use the implementations which are part of the libscapi library [20] for garbled circuits, since all the known garbled circuit optimizations have been incorporated into it. All implementations (Table 1) make use of AES-NI instruction set.

Protocol	**Scheme**
GLNP15 [27]	Garbled Circuits (Pipelined-garbling + Key Scheduling; 4-2 GRR + XOR-1)
	Garbled Circuits (Pipelined-garbling + Key Scheduling; Free-XOR; Half-gates)
ZRE15 [59]	Garbled Circuits (Fixed key AES; Free-XOR; Half-gates)

Table 1. List of implementations.

GLNP15. The first implementation is based on the garbling scheme from [27]. The main advantage of this garbling scheme is that it only makes very conservative computational assumptions, i.e., it can be proven secure under the assumption that AES behaves like a pseudorandom function (PRF). Like all garbling schemes that we consider in `Transputation`, the garbling of linear gates (e.g., XOR) and non-linear gates (e.g., AND) is performed differently. In GLNP15, garbling an AND gate produces two ciphertexts (using the *4-2* Garbled Row Reduction technique, or *GRR* for short), while garbling an XOR gate produces one ciphertext using the *XOR-1* technique. From a computational point of view, garbling with AES key scheduling is pipelined. Four key schedules are required for every gate (AND and XOR) during garbled circuit construction while two key schedules are required per gate during circuit evaluation.

Half-Gate. The second implementation is based on the work of [59], and uses the so called "half-gate" optimization, which in turn is compatible with the "free-XOR" optimization of [39]. This optimization is achieved at the price of having to make a stronger computational assumption on AES, namely assuming some form of circular-security (a kind of related-key assumption). The half-gate optimization reduces the number of ciphertexts necessary to garble an AND gate from 4 to 2 (using a different approach than GLNP15). We refer to the original paper for further details.

JustGarble. The final optimization we consider was proposed in the JustGarble framework of [7]. Recall that key-scheduling is the most expensive phase when using the AES-NI instruction, i.e., the instruction is optimized to garble large amount of data under the same key, but loses some of its efficiency when different keys have to be used all the time. As described above, in garbling schemes each gate consists of ciphertexts where different keys are used, thus the full power of the AES-NI set is not exploited. In JustGarble, AES is used as an ideal permutation “in stream cipher mode” by setting the key as a fixed constant, e.g., to encrypt message m under key k one computes $C = AES_c(k) \oplus m$ for some constant c. This usage of AES is quite non-standard, and amongst the three presented, provides the most extreme efficiency/security trade-off.

3.2 Applications and Circuit Size

Once we have fixed the protocol and the garbling scheme, we are left with one dimension, namely which function should be evaluated using the 2PC protocol.

For garbled circuit protocols, the circuit size plays a significant role in defining the amount of data that is to be transferred over the network. The amount of data transferred from the circuit Garbler to the Evaluator is a linear function of the circuit size. In particular, there is a difference in the price to pay (in terms of communication complexity) for linear gates (e.g., XOR gates) vs. non-linear gates (e.g., AND gates), and different garbling schemes have different coefficients for these two types of gates. In this work, we consider three applications with circuits of three different sizes. These circuits are becoming the de-facto standards for benchmarking of MPC protocols, mostly since they represent three different orders of magnitude in circuit sizes. In particular, we benchmark `Transputation` on the circuits for AES ($\approx 10^5$ gates), SHA256 ($\approx 10^6$ gates) and MinCut ($\approx 10^7$ gates). The exact number of gates and the distribution between AND and XOR gates for these circuits is shown in Table 2.

Function	AND gates	XOR gates
AES	6,800	25,124
SHA256	90,825	42,029
MinCut	999,960	2,524,920

Table 2. Number of gates.

Depending on the security assumptions, the amount of data for a particular circuit varies. When garbling using GLNP15, each AND gate produces two ciphertexts while each XOR gate produces one ciphertext. For AES, SHA256 and MinCut, amount of data communicated is 0.59MB, 3.41MB, 69.04MB respectively. When garbling using the Half-Gate construction or JustGarble, no ciphertexts are needed for XOR gates while two ciphertexts are produced per AND gate. For AES, SHA256 and MinCut, the amount of data communicated is 0.21MB, 2.77MB and 30.52MB. As can be observed here, for MinCut circuit with many XOR gates, the number of ciphertexts sent over the network has a high dependency on the garbling scheme used. In Table 3, a summary of the circuit sizes in megabytes is provided.

4 Transport Layer

In this section we describe the network characteristics which `Transputation` measures in order to optimise performance, parameters for selection of optimal transport layer protocol and the challenges of integrating transport sockets.

Assumption	**AES**	**SHA256**	**MinCut**
PRF	0.59	3.41	69.04
Circularity	0.21	2.77	30.52
Ideal cipher	0.21	2.77	30.52

Table 3. Garbled circuit size in Megabytes.

4.1 Transport Protocol Selection

Given the sizes of the inputs and the 2PC implementation, the transport layer of `Transputation` measures the packet loss and the latency and consequently determines which transport protocol is optimal. The decision whether reliability and congestion control mechanisms are needed is made also considering the network characteristics, e.g., LAN or WAN.

`Transputation` runs continuous experiments with PING, probes network for losses, tries sending at different rates and selects a protocol which empirically produces optimal performance.

Protocol selection based on latency. Latency plays an important role in the choice of transport layer protocol. When the time to transmit one TCP window is higher than the round trip time (RTT), the transmission proceeds in full pipe, and is essentially similar to UDP since the congestion window does not limit the transmission.

Specifically, let W be the bytes in the TCP window and let t_{trans} be the transmission delay of one byte. Let RTT be the time it takes to transmit one TCP segment and receive an ACK. If $W \cdot t_{trans} > RTT$, then there is no impact of TCP congestion window on the latency since transmission proceeds in pipeline.

`Transputation` measures the RTT, the window size W and the ratio $W \cdot t_{trans}$ vs. RTT and based on this determines which transport protocol to use (i.e., window-based or to transmit in full pipe). In low latency networks, e.g., evaluations on the same LAN, congestion control is generally insignificant (typically LANs do not suffer from packet losses and have low latency). In those settings `Transputation` resorts to UDP-like protocols.

Protocol selection based on communication rounds. The relevant parameters here are number of interaction and data volume. Window based protocols, such as TCP, are not optimal for 2PC implementations with small number of interactions and large data volumes. Window-less protocols, such as SABUL, provide better performance. This is due to the fact that the transmission window increases with the number of RTTs. In TCP the window starts with one segment and increases exponentially with every received ACK. As a result, although on the application layer only a few interactions are required, they will be performed in multiple interactions round on the transport layer. This factor is most evident in networks with high latency, such as WANs.

Protocol selection based on packet loss. TCP performs poorly during packet loss events, even when very few packets are lost, say 1%. During packet loss, reliability should be taken care on the user space with UDP protocol or with SABUL.

4.2 Avoiding Packet Loss

Packet loss is the phenomenon, where one or more packets that are sent, do not reach the destination. Lost packets need to be retransmitted, which increases the overall time spent for communication. The main cause for packet loss is the buffers. When buffers are full, subsequently arriving packets are discarded. This can be either at the sender side or the receiver side. Transputation avoids packet loss at the sender and receiver by adjusting the size of buffers as a function of latency, transmission rates and the data volume that needs to be transmitted.

Specifically, given (1) the differences between the transmission rate and the rate at which the data is passed on to the IP buffer (respectively transport layer) at the sender, (2) the transmission rate and the rate at which the IP buffers (and respectively transport layer) are depleted at the receiver and, (3) the input sizes and the secure computation implementation (both of which define the data volume and the rate at which it will be exchanged), Transputation performs a computation of the maximal amount of data that can be sent in one window.

The computation is performed by Transputation is as follows. Given a receiver buffer of size B, with data arrival rate $R_{arrival}$, and the buffer depletion rate R_{read}. Transputation computes the maximum window size as geometric series that converges to:

$$L = \frac{B}{1 - R_{arrival}/R_{read}}$$

During the computation, the data transmitted will be limited by L bytes during each transmission window. This accounts for the data that is being read, while new data arrives, and allows to optimize the communication.

4.3 Transport Protocols in Transputation

Implementing network functionality requires handling raw network sockets provided by the operating system. As a result, the same code, e.g., to open a TCP socket, is written numerous times and mixed with application code. However, if the transport protocol is to be changed, then it is hard to unwrap the network code from the application code. To ease development, our framework is modular. We provide abstract methods independent of the actual transport protocol implementation to make switching of transport protocols convenient. The use of abstract methods enables any secure computation protocol to use any transport protocol. Developers of secure computation protocols as well as developers implementing transport layer protocols do not require in-depth knowledge of the other layer; therefore, reducing development overhead. Furthermore, developers of secure computation protocols save time as the need to implement network functionality is reduced to calling the abstract function set provided by the transport layer of Transputation.

We integrated into Transputation the following transport protocols: TCP, UDP and SABUL. We implemented TCP for comparison to other transport protocols. We used UDP as a benchmark for connection-less protocols on networks without losses. Examples for such networks is evaluations on LAN, or implementations which are meant to run on 5G networks, which guarantees reliability and no packet loss [48,53]. We integrated SABUL, as it is currently used by an increasing number of applications, and provides reasonable performance for Internet communication. See preliminaries in Appendix B. New and even more efficient protocols may be developed in the future. Transputation enables easy integration of new and additional transport protocols. We describe the steps needed for integrating new protocols into Transputation in Section 5.

5 Transputation Implementation

The main design goal of Transputation is to provide a modular design which can be extended with other secure computation protocols as well as transport layer protocols. Transputation allows secure computation researchers to focus on the protocol details without worrying about the networking aspects of the implementation. Modularity is not restricted to secure computation protocols. Transport layer protocols, ancillary to the those included in the framework, can be added to the framework if required.

Abstraction. The transport layer part of `Transputation` is implemented as a wrapper written in C++ which can be easily plugged in with secure computation protocols. The current version uses synchronous sockets which is sufficient for our purposes. It abstracts the network functionality and removes the requirement to deal with the transport layer protocols themselves. The transport layer protocol can be set at runtime, making it easier to compare different protocols without the need for recompilation. Since most of the secure computation implementations developed in the past few years are written in C++, polymorphism in C++ are used to achieve modularity. We implemented an abstract class with methods needed to establish and close connections, and to send and receive data. Every transport layer protocol that is or will be implemented in our wrapper extends this class and implements these methods. This provides secure computation developers with two benefits: First, they do not need to know how to use the transport layer protocol and second, they can use new protocols that are added to the wrapper without making any change to the executable or library of the secure computation implementation. To implement a new protocol only the four methods of the abstract class are required: `SetupClient`, `SetupServer`, `RecvRaw`, `SendRaw`. Adding a transport layer protocol to the framework is as easy as adding it to a C or C++ program.

The wrapper currently supports UDP, TCP and SABUL. Since UDP does not provide reliability through features such as retransmission or reordering of packets, it cannot be used in real-world scenarios with packet losses and where the correct order of packets cannot be guaranteed. If a dedicated network without packet loss is available, then UDP can be used. In all other cases, we recommend to use TCP or SABUL in `Transputation`. To choose between TCP and SABUL, we provide evaluations in the following sections.

Simplification. We have used predefined class methods to simplify common tasks. When an instance of the `Transport` class is created, a socket is already allocated and set up on creation (by using `socket()` e.g. for TCP or UDP). Then the user decides if a client which connects to a server should be created, or if a server which listens on a port and waits for incoming connections should be created. For example, using our wrapper, a TCP server can be setup as shown in Figure 2. This reduces the number of lines needed as well as improves the readability and encourages users to separate program logic from network code. This is important to make the code more reusable. It is also easier to test the functionality of different parts when they are separated in a modular design.

```
auto *t = GetTransport(”tcp”);
t->SetupServer(”0.0.0.0”, 1234);
```

Fig. 2. Setting up a server with our wrapper.

The wrapper also includes two static methods, `GetLatencyClient()` and `GetLatencyServer()` to measure the latency. These methods use UDP packets to measure the RTT in milliseconds. This can be used to decide which protocol should be used. Finally, the wrapper takes care of packet sizes' byte order, which makes it simple to port applications to different platforms.

Packet handling. Transport layer protocols have contrasting methods to send data. For instance, UDP sends single packets, and hence sending ten 100 byte packets is not an issue. However, sending packets larger than the maximal allowed packet size, limited by the underlying maximum transmission unit (MTU) of the network stack, is not possible without splitting the data into smaller chunks. This has to be done by the program that incorporates UDP, which does not provide this by itself. In contrast to UDP, TCP sends data as a stream. If the data to be sent is too large, it will be split into multiple packets by the protocol. To simplify usage of different transport layer protocols, the wrapper allows users to send and receive packets in arbitrary sizes.

To solve the issue where multiple packets are received as a big chunk, the wrapper includes a `Packet` class which can be used to send a given amount of data. The receiver can then, without knowing the packet size prior to receiving, receive the packet. For all protocols, that can be included in the wrapper, the data will be split into multiple packets if needed and reassembled at the receiver side. When 10 packets are sent with the wrapper it will receive 10 packets. In order to tell the packets apart, the length information is added to the data so it can be interpreted by the receiving side. This length information is converted to network byte order to make it cross compatible among systems with a different endianness. So sending data from a big-endian machine to a little-endian machine (or vice versa) will still work since the bytes of the 16 bit length value are not accidentally flipped due to different byte order.

Integration. The wrapper can be integrated into any secure computation protocols, by simply calling methods of the wrapper such as `Send()` or `Recv()` in the `Transport` instance. The protocols can be supplied as a string such as `udp`, `tcp`. If the wrapper is then extended with a new transport layer protocol, then the framework will be able to use this protocol without any changes. For example, an echo client that connects to a server, receives a packet and echos it can implemented with the following code:

```
auto *t = GetTransport(”tcp”);
t->SetupClient(”0.0.0.0”, 1234);
t->Connect();
auto p = t->Recv();
t->Send(p);
```

Fig. 3. Example of an echo client using our wrapper.

We have integrated the wrapper into libscapi. Libscapi uses external OT-extension libraries which has its own network code. By incorporating our wrapper in libscapi, both - libscapi code and the OT extension code - can use suitable transport layer protocols without making any change to the rest of the code. This is an advantage that can be achieved by using a common network wrapper. Transport layer protocols can be added to the wrapper and used by both. With our approach, we want to separate the network code from the application code.

6 2PC TestBed

For easy testing of secure computation protocols using different transport layer, we propose a testing environment. Here users can upload their own implementations and evaluate different transport layers on real networks.

The testing environment consists of several computers located all over the globe. Only one computer serves as a gateway and is accessible from the Internet. Users can upload the code to the gateway machine.

Users are required to register for an account so that their programs are locked inside a virtualized environment (using QEMU[6]) which also limits the amount of CPU resources, memory, and harddisk space which can be used. This virtualized environment is set up at every location where the tests will be run.

The non-gateway servers mount the filesystem that contains the uploads from users via SSH[7] for easy distribution of the files while maintaining security. The users have to upload two script files ('customname-client.sh' and 'customname-server.sh') along with their programs, where 'customname' can be any name only containing alphanumeric letters. An example for a 'client.sh' is shown in Figure 4. This way it is possible to upload multiple files to test. The frontend will then display the chosen name as an option if both, the 'server.sh' and 'client.sh' are present for it. When running the evaluation, the server will invoke these files and provide the IP address of the opposite server and of the current server running the script as the first parameter and the second parameter. This way, the experiments can be run on different servers without any changes to the code.

As a third parameter the transport layer is passed as a lowercase string. In the Bourne Shell ('/bin/sh') these parameters are accessed by the variables '$1', '$2' and '$3', which would contain the values of the opposite server's IP address, the current server's IP and the transport layer protocol being used. As in Figure 4 '$2' may not be needed, since clients usually don't bind to IP addresses. However, to avoid confusion when writing server and client shell scripts, and to enable bidirectional communications, this parameter will still be passed. Optionally, the user can install libraries from Ubuntu 16.04 repository to the virtualized environment. The `Transputation` libraries for abstracting the transport layer are already preinstalled.

```
#!/bin/sh
# $2 is not needed by a client
./a.out --connect $1 --transport $3
```

Fig. 4. Example client.sh script.

When the user uploads the programs and scripts, it is possible to run simulations from the front end. Here the user is presented with all the uploaded scripts. It is possible to select different locations (in LAN or WAN setups) by choosing the program to run, the location for the server and the client to connect with. Then, the user can check the transport layer protocols to be used for the evaluation. The environment will run the evaluation and present the user with the result for all the selected transport layers.

7 Simulation and Experimental Evaluation

In this Section, we provide the simulation results which are used to understand the effect of latency and packet loss on the performance of secure computation protocols (Section 7.1).

[6] https://www.qemu.org/

[7] https://github.com/libfuse/sshfs

Then we describe realistic deployment scenarios (Section 7.2) followed by evaluation results obtained in LAN and WAN settings (Section 7.3).

7.1 Simulations

In this section, we provide the results obtained by simulating two parties performing secure computation on a single machine. The simulation results provide the benchmark for the executions in real network setups describe in Section 7.3. We simulate latency and packet loss using `tc qdisc` network emulator on a single Vultr instance with a 64-bit single core CPU with 2.6 GHz and 2 GB RAM. The communication takes place over the loopback interface.

Through these simulations we aim to understand the impact of latency and packet loss on secure computation protocols. We simulate latencies between 1ms to 300ms, packet losses between 0.01% and 0.05% and bandwidth of 200Mbps, 500Mbps, 1Gbps, 10Gbps, 15Gbps and 25Gbps. The latencies were chosen to represent communication between machines on the same network as well as those in different parts of the world. For instance, the round-trip time (RTT) within North America is 50ms on average, RTT between machines on either side of the Atlantic Ocean is about 100ms and machines placed in North-America and Asia or EU and Australia is about 300ms. Packet losses were chosen such that they are representative of realistic packet losses observed in networks[8]. Bandwidths were chosen based on measurements performed using `iperf` on different networks—local and cross-Atlantic. Here we have present the results on a 10Gbps network.

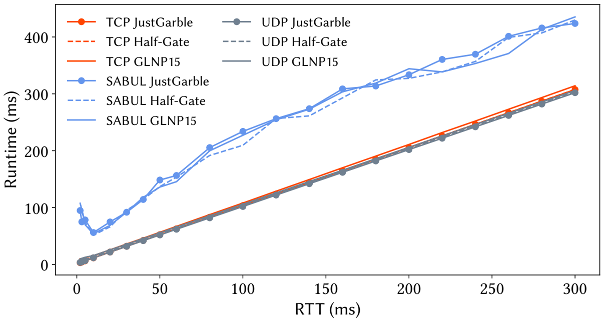

Fig. 5. Effect of latency on AES on a 10Gbps link.

To understand the effect of latency on secure computation protocols, we run them using TCP, UDP and SABUL. We use UDP to provide the best possible runtime of the protocol if a reliable transport layer protocol was not needed. As reliable communication is required to satisfy the correctness property of secure computation in real networks, we use UDP to benchmark the runtimes achievable when the bandwidth is optimally utilized. We use two reliable transport layer protocols, TCP and SABUL, to show the bandwidth utilisation for different circuit sizes. All experiments are run using single thread using AES-NI and the results here are the average of 100 runs.

For a small circuit such as AES, it can be observed in Figure 5 that TCP with Nagle's algorithm [43] disabled performs better than SABUL. This is because for small circuits, few

[8] `http://www.verizonenterprise.com/about/network/latency/`

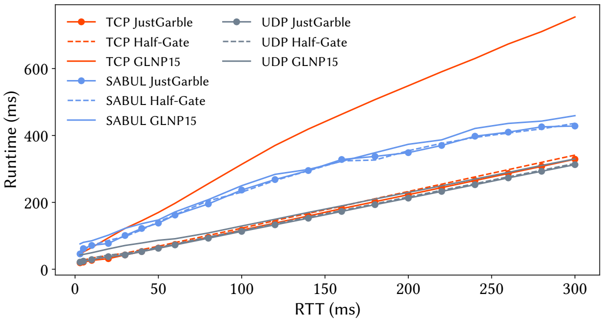

Fig. 6. Effect of latency on SHA256 on a 10Gbps link.

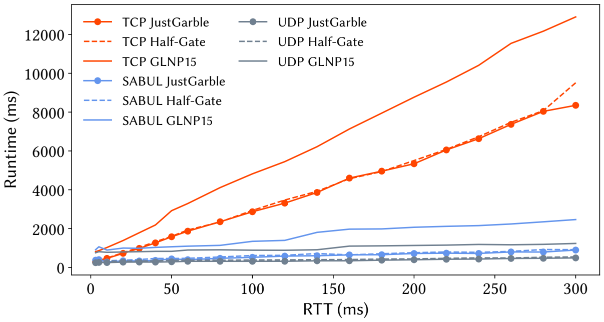

Fig. 7. Effect of latency on MinCut on a 10Gbps link

kilobytes of data are sent over the network while SABUL is optimized for transfer of large data transfer. For a medium-sized circuit such as SHA256, it can be observed in Figure 6 that TCP performs better than SABUL under ideal cipher assumption and circularity assumption, while under PRF assumption, the performance of SABUL is better than TCP as RTT increases. Our observation is due to a combination of reasons: the performance of SABUL is better as Bandwidth Delay Product (BDP) increases as well as when the amount of data transferred increases.

For a large circuit such as MinCut, the performance of SABUL and TCP is quite different from that observed for AES and SHA256. It can be observed in Figure 7 that SABUL utilizes the available bandwidth much better than TCP. As many packets are sent from the circuit Garbler to the Evaluator, the congestion control mechanism plays an important role in controlling the rate of packet transmission. The increase in latency affects the performance of TCP more than SABUL as SABUL uses a timer-based selective ACK instead of reacting to packet level events.

When considering packet loss, for small circuits such as AES, loss rate that we consider impacts the performance of TCP more than SABUL, as can be seen in Figure 8. For medium-size circuits such as SHA256 and large circuits such as MinCut, increase in loss rate deteriorates the performance significantly when TCP is used. On the other hand, SABUL handles packet loss better and the performance deterioration is very low. As can be observed in Figure 9 and 10, for large and medium-sized circuits on a network with at least 100ms delay, SABUL is more suitable than TCP. Here we have presented the results for GLNP15 on a 10Gbps network. We have also plotted the standard deviation of the runs as the variance is non-negligible in many of the results.

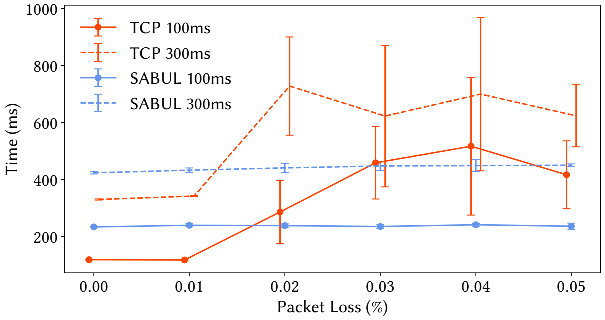

Fig. 8. AES on 10Gbps at various loss rates.

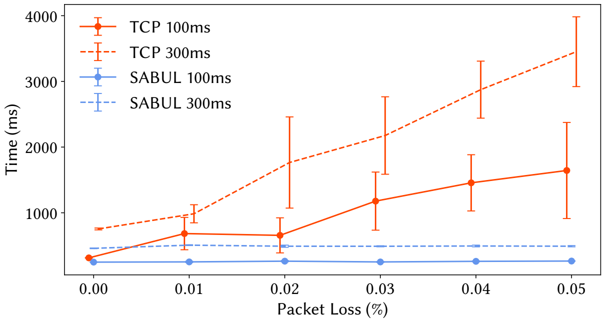

Fig. 9. SHA256 on 10Gbps at various loss rates.

Network bandwidth has been improving over the years. Considering this trend, we consider a wide range of bandwidth to show the performance of secure computation protocols for AES, SHA256 and MINCUT. Figure 11, 12 and 13 are the results for GLNP15 protocol for a simulated network bandwidth of 15Gbps and 25Gbps. Though the performance of TCP for secure computation is comparable to UDP at low latencies, such as 1ms, we observe that bandwidth is underutilized by TCP at higher latencies for both SHA256 and MINCUT.

7.2 Deployment Setups

For performance evaluation, two deployments were set up on our testbed environment: LAN setting and WAN setting. These settings provide two common setups of evaluation of secure computation protocols.

LAN setting. In the LAN setting, we run the experiments on two Azure instances located in the same data centre in the EU using high-bandwidth network and low latency. Each instance has a 64-bit Intel Xeon quadcore CPU with 2.4 GHz and 28 GB RAM. The latency was 0.5 ms on a 10Gbps link. The variance observed was within 10%.

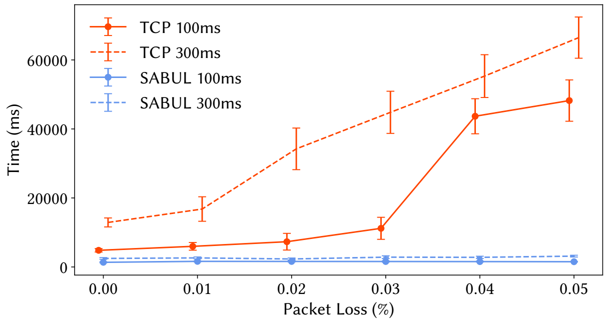

Fig. 10. MINCUT on 10Gbps at various loss rates.

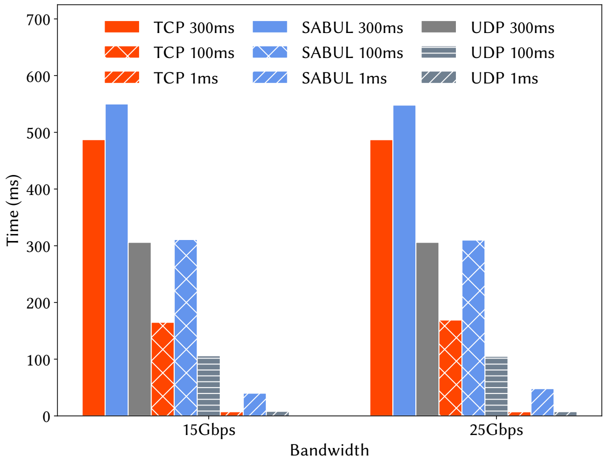

Fig. 11. Performance of AES-GLNP15 at high bandwidths

WAN setting In the WAN setting, we ran experiments on two pairs of locations with different latencies. These locations were chosen to show the behaviour of secure computation protocols with different transport layer protocols as latency increases.

EU-US: In this setting, two Azure instances with a 64-bit Intel Xeon quadcore CPU with 2.4 GHz and 28 GB RAM are used. One is located in the EU while the other is located in central US. The latency was 110ms and the network speed was estimated to be 1Gbps. The measured speed for a single TCP connection was 200Mbps on average. Both machines run Ubuntu 16.04. Variance of 15% was observed in this setting.

EU-AUS: In this setting, two Azure instances with a 64-bit Intel Xeon quadcore CPU with 2.4 GHz and 28 GB RAM are used. One is located in the EU while the other is located in south-east Australia. The latency was 300ms and the network was estimated to be 1Gbps. The measured speed for a single TCP connection was 100Mbps on average. Variance of 20% was observed in this setting for secure computation of AES and SHA256 while the variance for MinCut was 30% for TCP and 25% for SABUL.

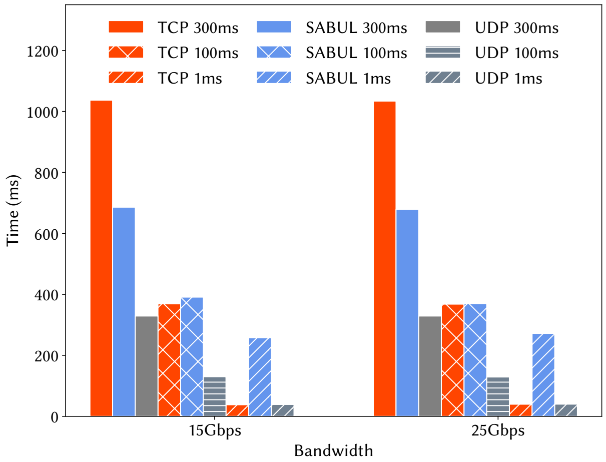

Fig. 12. Performance of SHA256-GLNP15 at high bandwidths

All experiments are run using single thread and the Azure instances use AES-NI. Garbled circuit protocols were run 100 times. The results in the following sections are the average of these executions.

7.3 Experimental Evaluations

In this section, we present the results obtained by using `Transputation`. The results were obtained using the network wrapper and running the three secure computation protocols using TCP and SABUL for communication in LAN and WAN settings.

Circuit	Setting	JustGarble [7]		Half-Gate [59]		GLNP15 [27]	
		TCP	SABUL	TCP	SABUL	TCP	SABUL
AES	LAN	**2.4**	187.1	**2.9**	191.3	**7**	202.7
	EU-US	**127.4**	403.9	**126.3**	408.3	**130.4**	444.2
	EU-AUS	**312.44**	566.84	**310.88**	580.2	**377.92**	592.5
SHA256	LAN	**13.5**	191.9	**19.9**	226.7	**30.5**	233.45
	EU-US	**146.23**	332.24	**151.99**	318.26	**266.46**	411.96
	EU-AUS	**362.53**	568.22	**394.13**	587.03	650.44	**612.43**
MinCut	LAN	**255.19**	598.2	**267.2**	740.2	**700.8**	1255.9
	EU-US	2616.59	**896.74**	2783.6	**957.6**	4911.89	**1802.25**
	EU-AUS	8204.61	**1068.07**	8693.57	**1163.14**	13805.2	**2001.27**

Table 4. Experimental results for garbled circuit protocols (Runtime in ms)

We summarize the experimental results in Table 4. The timing measurements in the table include the garbling time, transfer of data from the Garbler to the Evaluator and the computation of output. In LAN setting, TCP performs best for all circuit sizes. This is because, when Nagle's algorithm [43] is disabled, the packets are sent as soon as they

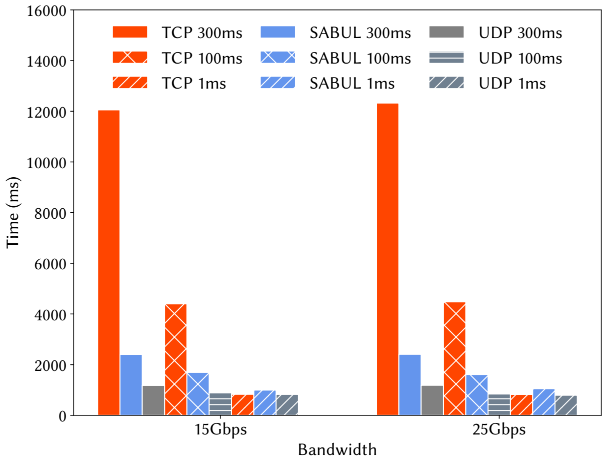

Fig. 13. Performance of MinCut-GLNP15 at high bandwidths

arrive at the buffer. Disabling Nagle's algorithm is advantageous in LAN setting as the communication is fast and computation consumes bulk of the time taken by the protocol.

In the WAN setting, when the secure computation protocol is run between Azure instances in EU and US, the circuit size begins to influence the performance. For AES and SHA256, TCP is still more efficient than SABUL but the tide tilts for MinCut. MinCut using SABUL is 2.7-3x faster than TCP. In fact, secure computation of MinCut under PRF assumption using TCP is more efficient than under ideal cipher assumption or circularity assumption using TCP.

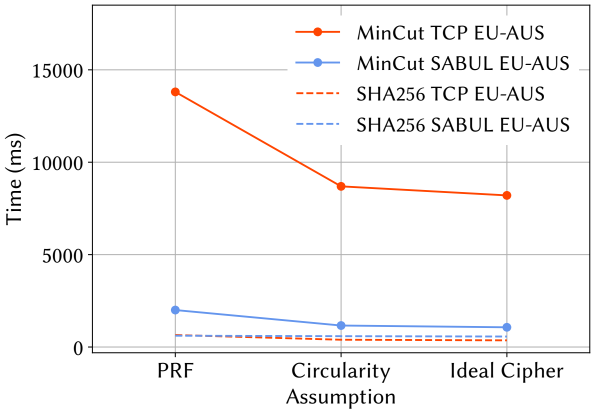

Fig. 14. Comparison of assumptions.

In the WAN setting, when the secure computation protocol is run between Azure instances in EU and Australia, the influence of latency on secure computation of large circuits becomes much more evident. Secure computation of SHA256 under PRF assumption (with 223,679 ciphertexts) using SABUL is only a little faster than using TCP. Secure computation of MinCut using SABUL is $7 - 8$x faster than using TCP. This is significantly more than the improvements that can be expected from secure computation protocol improvements [59]. In Figure 14, we provide a comparison of the performance of TCP and SABUL under the three security assumptions which further emphasizes the improvement in performance by using appropriate transport layer protocols.

The comparison of TCP and SABUL with respect to increasing ciphertext size can be seen in Figure 1. The impact of latency on the two transport layer protocols becomes apparent from this figure. When using SABUL, time taken by secure computation protocols increases by 200ms when latency between the two parties increases by 200ms. On the other hand, performance of TCP deteriorates much more significantly as latency increases.

8 Related Work

Secure Computation. Secure two-party computation (2PC) allows two parties to jointly compute any function of their inputs and obtain the result without leaking any other information. Secure 2PC protocols have been extensively studied since mid-1980s when the first feasibility results were provided by Yao [58] and Goldreich, Micali and Wigderson (GMW) [21]. These seminal results show that it is possible to evaluate any function in a secure way, that is, two parties P_1, P_2 with inputs x, y can jointly evaluate some function f (expressed as a Boolean circuit) on their inputs in such a way that both parties learn the desired output $z = f(x, y)$ *and nothing else* about the input of the other party.

A long line of work has tried to increase the efficiency of garbled circuits. The original protocol by Yao [58] requires a transfer of 4 ciphertexts per Boolean gate in the circuit, and requires 4 decryptions for the evaluation of a garbled gate. The number of decryptions per gate in the evaluation phase was reduced to 1 thanks to the *point-and-permute* strategy of [6], which also reduces the size of the ciphertexts by a factor 2 (approximately). The main measure of *communication complexity* of garbling schemes is the number of ciphertexts which are transmitted per Boolean gate. This was first reduced in [45] using the *garbled row-reduction technique (GRR)* which only requires 3 ciphertexts per gate (the so called 4-3 GRR technique). This was further optimized to 2 ciphertexts per Boolean gate in [49] (e.g., 4-2 GRR). Thanks to the *free-XOR* technique [39], it is not necessary to transfer any ciphertexts for XOR (or other linear) gates. Unfortunately the *free-XOR* technique was incompatible with the more advanced 4-2 GRR technique. The two techniques were finally combined thanks to the *half-gate* optimization, which combines the benefit of *free-XOR* (no ciphertexts for XOR gates) with the advanced 4-2 GRR technique (only 2 ciphertexts per AND gate). Unfortunately, the free-XOR technique is only secure under non-standard cryptographic assumption [14] and, in particular, it requires some form of "circular security" assumption. Using an even stronger assumption, i.e., *fixed-key* AES behaves like a *random permutation*, faster garbling schemes were proposed in [7].

While efficiency is a crucial aspect in 2PC, some have questioned how safe it is to keep making stronger assumptions about the encryption schemes to gain in efficiency, especially since such encryptions (e.g., AES in practice) have not been designed with these properties in mind. In particular, we note that a conservative approach is usually adopted by the industry as it is difficult to change protocols if vulnerabilities are discovered after deployment. Therefore [27] provided novel constructions of garbled circuits that require 2 ciphertexts for AND gates (4-2 GRR) and 1 ciphertext for XOR gates (XOR-1) and that can be proven secure using standard assumptions only. The main conclusion in [27] is that the price to pay for the stronger security guarantees in practice is much less than it is in theory. Our evaluations in some sense confirm and strengthen the conclusions of [27]: our experiments show that the choice of the right network protocol has a much higher impact on overall efficiency than gambling on security by using non-standard assumption. In particular (when evaluating large circuits over WAN), using SABUL plus standard assumptions is 4 times faster than using TCP and non-standard assumptions.

Secure Computation Frameworks. Since the implementation of Fairplay framework [42] in 2004, various frameworks have been developed for secure 2PC [36,20,17]. Currently, a garbled circuit framework with security against passive and active adversaries is provided in libscapi [20] library with the latest optimizations. Though our work only considers Boolean circuits, we note that a framework for mixed protocol was implemented in ABY [17]: this framework combines arithmetic sharing, boolean sharing and garbled circuits and allows for conversions between the sharing methods. All the previous frameworks focus on secure computation and use standard TCP socket provided by the operating system.

The work of [32] surveys general purpose compilers for secure computation. These compilers provide high-level abstractions to describe functions in an intermediate representation (such as a circuit). The compiled circuit is executed securely during runtime when the input is provided. While [32] focuses on compilers, we focus on efficiency of protocol execution. Hence, the two works address orthogonal problems. The sample programs chosen in [32] do not represent practical MPC use case and instead attempt to test the usability of the compiler code and how easy it is to write example code for an application for the compiler. Furthermore, the tests are performed on standalone environment and do not account for the context, such as the network condition, in which the protocol is run. The goal of their paper is not to provide a practical testing framework, neither is it to test the efficiency of the protocols. Instead, their goal is to make explicit which compilers are written for experts and which for non-experts.

Another recent related work is [4], which considers the possibility of providing MPC as a service where users use the platform to run protocols. They provide an environment where users can participate in a low-bandwidth MPC protocol using web-browser or an app on the phone. If the users are online during the execution of the protocol, then they can use their own device otherwise they can store their inputs on their own cloud instance or use one from the service provider (with lower privacy guarantees). While they focus on low-bandwidth MPC protocols (which require multiple rounds and present evaluations only in LAN setting), we focus on high-bandwidth (garbled circuits) constant round 2PC protocols in LAN and WAN settings.

Recent related work shows that significant progress has been made in computation complexity of 2PC implementations. When evaluated on a single host the implementations produce good performance. However, on real networks with other traffic and concurrent processes, latency and packet loss, the efficiency collapses and implementations incur prohibitive latency. In particular we note that no previous work has considered how to improve secure computation by addressing the issue of transport layer performances, as addressed in this work.

9 Conclusions and Future Research

For the first time, our work demonstrates that, contrary to folklore belief, bandwidth is *not* the bottleneck in performance of secure computation. The performance of 2PC implementations can be significantly improved if other transport protocols are used instead of the standard

TCP sockets. The under-utilisation will be further exacerbated with the adoption of new technologies such as 5G, where networks guarantee no packet loss and high transmission rates. Our evaluations demonstrate performance improvements for 2PC even with general purpose transport layer protocols, e.g., SABUL is 8 times more efficient than TCP for the same task and 4 times more efficient than TCP when comparing 2PC with standard assumptions over SABUL vs 2PC with non-standard assumptions over TCP.

Support of multiple protocols. Specifically tailored transport layer protocols, e.g., for a specific computation task or network setup, could further improve the performance.

To enable 2PC developers to benefit from the wide range of existing protocols, we have developed `Transputation`. Users can setup `Transputation` locally or can use our installation (with preconfigured 2PC implementations and transport protocols) which can be accessed online. Our installation of `Transputation` is running on the testbed with representative network setups for evaluation of secure computation implementations.

Evaluation of 2PC in realistic setups. Although evaluations on one host provide practical results, they "break" when run in real Internet networks. We setup a testbed for evaluation of 2PC implementations. Our testbed automates setup and configurations, and allows the developers to run evaluations without having to dive into complicated setups, renting remote machines and running network measurements.

Future Research. Our work opens opportunities for followup improvements towards wide-scale deployment of 2PC implementations for practical systems. Here are some examples of next steps for future work on the transport layer and on the computation layer:

- Different 2PC implementations exhibit distinct communication patterns (e.g., communication rounds and volume of data transmitted during each round). Future work is needed to devise specially engineered transport protocols for specific computation tasks and applications. This is an important yet nontrivial research direction, and requires investigation of the specific properties of the target application (say computations on medical data or computations for online trade market) as well as the network conditions in which they are run.
- So far we have considered two-party protocols based on garbled circuits, with focus on security against passive adversaries only. Two natural future directions for research are active security and protocols for multi-party computation.

References

1. Alizadeh, M., Greenberg, A., Maltz, D. A., Padhye, J., Patel, P., Prabhakar, B., Sengupta, S., and Sridharan, M. Data center tcp (dctcp). *ACM SIGCOMM computer communication review 41*, 4 (2011), 63–74.
2. Archer, D. W., Bogdanov, D., Lindell, Y., Kamm, L., Nielsen, K., Pagter, J. I., Smart, N. P., and Wright, R. N. From keys to databases - real-world applications of secure multi-party computation. *Comput. J. 61*, 12 (2018), 1749–1771.
3. Asharov, G., Lindell, Y., Schneider, T., and Zohner, M. More efficient oblivious transfer and extensions for faster secure computation. In *ACM Conference on Computer and Communications Security* (2013), ACM, pp. 535–548.
4. Barak, A., Hirt, M., Koskas, L., and Lindell, Y. An end-to-end system for large scale P2P mpc-as-a-service and low-bandwidth MPC for weak participants. In *CCS* (2018), ACM, pp. 695–712.
5. Beaver, D. Correlated pseudorandomness and the complexity of private computations. In *STOC* (1996), ACM, pp. 479–488.

6. Beaver, D., Micali, S., and Rogaway, P. The round complexity of secure protocols (extended abstract). In *STOC* (1990), ACM, pp. 503–513.
7. Bellare, M., Hoang, V. T., Keelveedhi, S., and Rogaway, P. Efficient garbling from a fixed-key blockcipher. In *IEEE Symposium on Security and Privacy* (2013), IEEE Computer Society, pp. 478–492.
8. Bellare, M., Hoang, V. T., and Rogaway, P. Foundations of garbled circuits. In *ACM Conference on Computer and Communications Security* (2012), ACM, pp. 784–796.
9. Bogdanov, D., Jõemets, M., Siim, S., and Vaht, M. How the estonian tax and customs board evaluated a tax fraud detection system based on secure multi-party computation. In *Financial Cryptography* (2015), vol. 8975 of *Lecture Notes in Computer Science*, Springer, pp. 227–234.
10. Bogetoft, P., Christensen, D. L., Damgård, I., Geisler, M., Jakobsen, T. P., Krøigaard, M., Nielsen, J. D., Nielsen, J. B., Nielsen, K., Pagter, J., Schwartzbach, M. I., and Toft, T. Secure multiparty computation goes live. In *Financial Cryptography* (2009), vol. 5628 of *Lecture Notes in Computer Science*, Springer, pp. 325–343.
11. Caini, C., and Firrincieli, R. Tcp hybla: a tcp enhancement for heterogeneous networks. *International journal of satellite communications and networking 22*, 5 (2004), 547–566.
12. Caini, C., and Firrincieli, R. TCP hybla: a TCP enhancement for heterogeneous networks. *Int. J. Satellite Communications Networking 22*, 5 (2004), 547–566.
13. Cardwell, N., Cheng, Y., Gunn, C. S., Yeganeh, S. H., and Jacobson, V. BBR: congestion-based congestion control. *ACM Queue 14*, 5 (2016), 20–53.
14. Choi, S. G., Katz, J., Kumaresan, R., and Zhou, H. On the security of the "free-xor" technique. In *TCC* (2012), vol. 7194 of *Lecture Notes in Computer Science*, Springer, pp. 39–53.
15. Chun, B., Culler, D., Roscoe, T., Bavier, A., Peterson, L., Wawrzoniak, M., and Bowman, M. Planetlab: an overlay testbed for broad-coverage services. *ACM SIGCOMM Computer Communication Review 33*, 3 (2003), 3–12.
16. Cristofaro, E. D., Manulis, M., and Poettering, B. Private discovery of common social contacts. In *ACNS* (2011), vol. 6715 of *Lecture Notes in Computer Science*, pp. 147–165.
17. Demmler, D., Schneider, T., and Zohner, M. ABY - A framework for efficient mixed-protocol secure two-party computation. In *NDSS* (2015), The Internet Society. Code: `https://github.com/encryptogroup/ABY`.
18. Dong, C., Chen, L., Camenisch, J., and Russello, G. Fair private set intersection with a semi-trusted arbiter. In *DBSec* (2013), vol. 7964 of *Lecture Notes in Computer Science*, Springer, pp. 128–144.
19. Dong, M., Li, Q., Zarchy, D., Godfrey, P. B., and Schapira, M. PCC: re-architecting congestion control for consistent high performance. In *NSDI* (2015), USENIX Association, pp. 395–408.
20. Ejgenberg, Y., Farbstein, M., Levy, M., and Lindell, Y. SCAPI: the secure computation application programming interface. *IACR Cryptology ePrint Archive 2012* (2012), 629. Code: `https://github.com/cryptobiu/libscapi`.
21. Goldreich, O., Micali, S., and Wigderson, A. How to play any mental game or A completeness theorem for protocols with honest majority. In *STOC* (1987), ACM, pp. 218–229.
22. Google. http://goo.gl/lGvnis.
23. Google. Spdy: An experimental protocol for a faster web, Accessed: Aug. 2017. `https://dev.chromium.org/spdy/spdy-whitepaper`.
24. Gu, Y., and Grossman, R. L. SABUL: A transport protocol for grid computing. *J. Grid Comput. 1*, 4 (2003), 377–386.
25. Gu, Y., and Grossman, R. L. UDT: udp-based data transfer for high-speed wide area networks. *Computer Networks 51*, 7 (2007), 1777–1799.
26. Gu, Y., and Grossman, R. L. Udtv4: Improvements in performance and usability. In *GridNets* (2008), vol. 2 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, Springer, pp. 9–23.

27. Gueron, S., Lindell, Y., Nof, A., and Pinkas, B. Fast garbling of circuits under standard assumptions. In *ACM Conference on Computer and Communications Security* (2015), ACM, pp. 567–578.
28. Ha, S., Rhee, I., and Xu, L. Cubic: a new tcp-friendly high-speed tcp variant. *ACM SIGOPS operating systems review 42*, 5 (2008), 64–74.
29. Ha, S., Rhee, I., and Xu, L. CUBIC: a new tcp-friendly high-speed TCP variant. *Operating Systems Review 42*, 5 (2008), 64–74.
30. Halevi, S. Advanced cryptography: Promise and challenges. In *ACM Conference on Computer and Communications Security* (2018), ACM, p. 647.
31. Handley, M. Why the internet only just works. *BT Technology Journal 24*, 3 (2006), 119–129.
32. Hastings, M., Hemenway, B., Noble, D., and Zdancewic, S. Sok: General purpose compilers for secure multi-party computation. In *2019 2019 IEEE Symposium on Security and Privacy (SP)* (Los Alamitos, CA, USA, may 2019), IEEE Computer Society.
33. Hemenway, B., Lu, S., Ostrovsky, R., and IV, W. W. High-precision secure computation of satellite collision probabilities. In *SCN* (2016), vol. 9841 of *Lecture Notes in Computer Science*, Springer, pp. 169–187.
34. Hock, M., Bless, R., and Zitterbart, M. Experimental evaluation of BBR congestion control. In *ICNP* (2017), IEEE Computer Society, pp. 1–10.
35. Huang, Y., Evans, D., and Katz, J. Private set intersection: Are garbled circuits better than custom protocols? In *NDSS* (2012), The Internet Society.
36. Huang, Y., Evans, D., Katz, J., and Malka, L. Faster secure two-party computation using garbled circuits. In *USENIX Security Symposium* (2011), USENIX Association.
37. Ishai, Y., Kilian, J., Nissim, K., and Petrank, E. Extending oblivious transfers efficiently. In *CRYPTO* (2003), vol. 2729 of *Lecture Notes in Computer Science*, Springer, pp. 145–161.
38. Kamm, L., and Willemson, J. Secure floating point arithmetic and private satellite collision analysis. *Int. J. Inf. Sec. 14*, 6 (2015), 531–548.
39. Kolesnikov, V., and Schneider, T. Improved garbled circuit: Free XOR gates and applications. In *ICALP (2)* (2008), vol. 5126 of *Lecture Notes in Computer Science*, Springer, pp. 486–498.
40. Kreuter, B. Secure multiparty computation at google. In *Real World Crypto Conference (RWC)* (2017).
41. Liu, S., Basar, T., and Srikant, R. Tcp-illinois: A loss- and delay-based congestion control algorithm for high-speed networks. *Perform. Eval. 65*, 6-7 (2008), 417–440.
42. Malkhi, D., Nisan, N., Pinkas, B., and Sella, Y. Fairplay - secure two-party computation system. In *USENIX Security Symposium* (2004), USENIX, pp. 287–302.
43. Nagle, J. Congestion control in ip/tcp internetworks. *RFC 896* (1984).
44. Naor, M., and Pinkas, B. Efficient oblivious transfer protocols. In *SODA* (2001), ACM/SIAM, pp. 448–457.
45. Naor, M., Pinkas, B., and Sumner, R. Privacy preserving auctions and mechanism design. In *EC* (1999), pp. 129–139.
46. Nielsen, J. B., Schneider, T., and Trifiletti, R. Constant round maliciously secure 2pc with function-independent preprocessing using LEGO. In *NDSS* (2017), The Internet Society.
47. Obata, H., Tamehiro, K., and Ishida, K. Experimental evaluation of tcp-star for satellite internet over winds. In *2011 Tenth International Symposium on Autonomous Decentralized Systems (ISADS)* (2011), IEEE, pp. 605–610.
48. Parvez, I., Rahmati, A., Guvenc, I., Sarwat, A. I., and Dai, H. A survey on low latency towards 5g: Ran, core network and caching solutions. *IEEE Communications Surveys & Tutorials 20*, 4 (2018), 3098–3130.
49. Pinkas, B., Schneider, T., Smart, N. P., and Williams, S. C. Secure two-party computation is practical. In *ASIACRYPT* (2009), vol. 5912 of *Lecture Notes in Computer Science*, Springer, pp. 250–267.
50. Pinkas, B., Schneider, T., and Zohner, M. Scalable private set intersection based on OT extension. *ACM Trans. Priv. Secur. 21*, 2 (2018), 7:1–7:35.

51. Roskind, J. Quic: Multiplexed stream transport over udp, Accessed: Aug. 2017. `https://docs.google.com/document/d/1RNHkx_VvKWyWg6Lr8SZ-saqsQx7rFV-ev2jRFUoVD34`.
52. Schneider, T., and Zohner, M. GMW vs. yao? efficient secure two-party computation with low depth circuits. In *Financial Cryptography* (2013), vol. 7859 of *Lecture Notes in Computer Science*, Springer, pp. 275–292.
53. Services, T. S. G., Management, S. A., and orchestration; 5G performance measurements, 2018. 3GPP TS 28.552 V16.0.0.
54. Wang, X., Ranellucci, S., and Katz, J. Authenticated garbling and efficient maliciously secure two-party computation. In *CCS* (2017), ACM, pp. 21–37.
55. Wei, D. X., Jin, C., Low, S. H., and Hegde, S. FAST TCP: motivation, architecture, algorithms, performance. *IEEE/ACM Trans. Netw. 16*, 6 (2006), 1246–1259.
56. Wu, H., Feng, Z., Guo, C., and Zhang, Y. ICTCP: incast congestion control for TCP in data center networks. In *CoNEXT* (2010), ACM, p. 13.
57. Wu, H., Feng, Z., Guo, C., and Zhang, Y. Ictcp: Incast congestion control for tcp in data-center networks. *IEEE/ACM Transactions on Networking (ToN) 21*, 2 (2013), 345–358.
58. Yao, A. C. How to generate and exchange secrets (extended abstract). In *FOCS* (1986), IEEE Computer Society, pp. 162–167.
59. Zahur, S., Rosulek, M., and Evans, D. Two halves make a whole - reducing data transfer in garbled circuits using half gates. In *EUROCRYPT (2)* (2015), vol. 9057 of *Lecture Notes in Computer Science*, Springer, pp. 220–250.
60. Zhao, Y., and Chow, S. S. M. Are you the one to share? secret transfer with access structure. *PoPETs 2017*, 1 (2017), 149–169.
61. Zhao, Y., and Chow, S. S. M. Can you find the one for me? privacy-preserving matchmaking via threshold PSI. *IACR Cryptology ePrint Archive 2018* (2018), 184.

A 2PC Background

In secure computation, three adversarial notions are most commonly used. First, *semi-honest* adversaries who follow the protocol but can try to extract more information from the transcript of the protocol. Second, *malicious* adversaries who can deviate from the protocol specifications and can try to collect as much information about honest parties. Third, *covert* adversary who may deviate but with the possibility of being caught with a certain probability. This work focuses on protocol that achieve *passive security* i.e., they are only secure against semi-honest adversaries.

A.1 Oblivious Transfer

Oblivious transfer is a cryptographic primitive in which one party, a sender, transmits multiple messages to another party, a chooser, such that the sender does not learn which message has been received by the chooser while the chooser does not learn anything about the other messages. In a 1-out-of-2 OT, the sender inputs two messages, say m_0 and m_1, and the chooser inputs a choice bit $b \in \{0, 1\}$. At the end of the protocol, the chooser obtains m_b, being ignorant of m_{1-b}, while the sender gains no information. OT protocols can be instantiated based on public-key type assumptions such as RSA or the Decisional-Diffie Hellman assumption. In practice, due to the high computational cost of the exponentiations required by OT protocols all modern implementation of 2PC use the OT-extension technique [5,37] which allows to generate an unbounded number of OT from a few base OT and using only symmetric crypto operations.

A.2 2PC Based on Garbled Circuits

Garbled Circuits. Following the framework of [8], we define a garbling scheme as a tuple of algorithm $(\mathsf{Gb}, \mathsf{En}, \mathsf{Ev}, \mathsf{De})$. Give a plaintext function f (expressed as a Boolean circuit) the randomized *garbling* algorithm Gb outputs a *garbled circuit* F, together with encoding/decoding information e and d. The *encoding* algorithm En on input a plaintext input x and the encoding information e outputs a garbled input X (similarly, the plaintext input y is mapped in a garbled input Y. The *evaluation* algorithm Ev can be used to evaluate the garbled function F on the garbled inputs X and Y thus producing a garbled output Z. The garbled output Z together with the decoding information d can be used to recover the plaintext output z. As a basic requirement we need garbled circuits to satisfy *correctness* i.e., $z = f(x, y)$. *Privacy* for garbled circuits informally states that the values (F, X, Y, d) can be simulated having access to the output z and the function f i.e., they reveal (computationally) no information about the inputs x, y.

Yao's Protocol. Standard garbled circuits that are used in 2PC protocols are called *projective* i.e., they have the property that the encoding information e is a set of pairs of random keys (k_i^0, k_i^1), and the number of pairs is the length of the input of the function. Now, the encoded version X of x is simply constructed by choosing the "right" key from each pair i.e., X is constructed by concatenating $k_1^{x_1}, k_2^{x_2}$ and so on. This allows to easily combine GCs with OT (since OT allows to select one element from a pair in an oblivious way). Armed with this tool, Yao's protocol for 2PC [58] can be described as follows: P_1 garbles a circuit f into F, then runs $|y|$ OT protocols where P_1 offers the key pairs from e and where P_2 uses the bits of y as choice bits. As a result, P_2 learns the garbled input Y and nothing else, and P_1 learns nothing at all. After this, P_1 sends the garbled function F, the garbled input X and the decoding information d to P_2. With this information, P_2 can evaluate the garbled function on the garbled inputs and decode the final output in such a way that P_2 learns nothing about P_1's input x. Yao's protocol requires only constant number of rounds of communication.

Yao Garbling. The original garbling scheme proposed by Yao follows a *gate-by-gate* approach which is still followed by every practically efficient instantiation of garbled circuits. In particular in Yao's garbling P_1 chooses two random keys for every wire in the circuit and then uses them to construct garbled gates. In particular for each wire i two random keys are sampled k_i^0 and k_i^1, one that logically represents the "plaintext" 0 bit and one that represents the "plaintext" 1 bit.

Then, P_1 uses these keys to construct so called "garbled gates", meaning that given a binary Boolean gate g with input wires u, v and output wire w and the keys representing its inputs and output P_1 constructs a table of ciphertexts consisting of encryptions of the output keys under the corresponding input keys i.e., a garbled gate is constructed by concatenating ciphertexts of the form

$$C_{a,b} = E(k_u^a, k_v^b; k_w^{g(a,b)})$$

for all four combinations of $a, b \in \{0, 1\}$, which are then randomly permuted. [9]

Garbled gates can then be evaluated obliviously by P_2 i.e., in Yao's protocol P_2 will know one key for each input k_u^a, k_v^b but not the values a, b. Yet P_2 will be able to learn the corresponding output key $k_w^{g(a,b)}$. To allow P_2 to begin the evaluation, the keys corresponding

[9] Many details are omitted in this simple overview e.g., E should have the property that it is easy to detect a successful decryption.

to the input wires must be sent from P_1 to P_2. This can be done directly for the input wires belonging to P_1, while it is done using *oblivious transfer (OT)* for the wires beloning to P_2. OT is a cryptographic primitive that allows P_2 to receive one out of two messages from P_1, in such a way that P_2 only learns one message and P_1 does not learn which one is being transfered. Finally, if P_2 is supposed to learn the output value, P_1 will also send some decoding information which allow to decode the keys corresponding to the output wires to their plaintext values.

Henceforth, we refer to P_1 as the *Garbler* and P_2 as the *Evaluator.*

Garbling Optimizations. The ciphertexts in the garbled gate are permuted at random in Yao's construction (to ensure that P_2 learns nothing about the output value $g(a, b)$) therefore P_2 must decrypt all 4 ciphertexts and have a mechanism to identify the right key. This can be achieved by adding redundancy to the plaintexts which in turns increases the communication complexity of the protocol. This was optimized in [6] with the *point-and-permute* technique where a "permutation" bit is added to the keys. By looking at these bits the evaluator can identify which ciphertext to decrypt thus reducing the computational overhead of evaluating by a factor 4 and the communication overhead by a factor approximately 2.

In the *garbled row reduction* (GRR) [45,49] technique one (or more ciphertexts) are artificially set to 0 and therefore do not need to be transmitted. Thus in the garbling process the key corresponding to the output wire is no longer picked at random but it is computed, pseudo-randomly, by decrypting the all 0 ciphertext with a combination of the input keys. When using the *free-XOR* technique [39] the choice of keys for the wires is even more restricted: now P_1 only chooses the 0 key for each input wire while the 1 key is defined by XOR-ing the 0 key with a global offset R. Then, by setting the output key corresponding to the 0 bit to be the XOR of the two input 0 key i.e.,

$$k_w^0 = k_u^0 \oplus k_v^0$$

it is easy to see that for all 4 combinations of $a, b \in \{0, 1\}$ it holds that

$$k_w^{a \oplus b} = k_u^a \oplus k_v^b$$

thus the evaluation of XOR gates only consists of XORing two keys which can be considered "for free". Unfortunately setting keys in this way makes it harder to prove the security of the garbling of non-XOR gates in the circuit e.g., the AND gates. In particular now a "circular security" assumption is required [14]: assume P_2 knows the 0 key for two input wires k_u^0, k_v^0 of an AND gate (and thus P_2 is allowed to retrieve k_w^0. Now, there are ciphertexts in the garbled gate that P_2 is *not* supposed to decrypt e.g., the one corresponding to inputs $1, 1$. But these ciphertex is now of the form

$$E(k_u^0 \oplus R, k_v^0 \oplus R; k_w^0 \oplus R)$$

that is, the only unknown for P_2 is R, which is encrypted under R itself.

The *half-gate* technique of [59] allows to combine *free-XOR* with garbled row reduction requiring only 2 ciphertexts to be sent for each AND gate. In a nutshell, the technique replaces every AND gate with two "half gates" (hence the name) which are easier to garble (and only require a single ciphertext) since either P_1 or P_2 already knows one of the two input bits of each half-gate. Due to the presence of free-XOR the security of the garbling still relies on a circular security assumption.

In [7] an efficient method to implement the encryption function E is given by defining:

$$E(k_u^a, k_v^b; k_w^{g(a,b)}) = \pi(k_u^a, k_v^b) \oplus k_w^{g(a,b)}$$

where π is assumed to be an ideal permutation and is implemented as $\pi(\cdot) = AES_\alpha(\cdot)$ i.e., by using AES with a fixed-key.

Finally [27] proposed a garbling method that only uses 2 ciphertexts for each AND gate and a single ciphertext for XOR gates using only the (minimal) assumption that AES is a *pseudorandom function.* In a nutshell, they garble AND gates by performing aggressive 4-2 row reduction i.e., both output keys of an AND gate are generated pseudorandomly from the input keys. Note that this is done in a more efficient way than [49] which uses (more expensive) polynomial interpolation techniques while [27] only uses (cheaper) XORs of strings. At the same time they garble XOR gates by using a variant of the Free-XOR technique in which the offset R is not global for the entire circuit but is computed pseudorandomly for each XOR gate by looking at the difference of a PRF evaluated on the two keys for an input wire, and by constructing a "half-gate" which allows to apply the same difference to the other wire.

A.3 GMW

Circuit	Setting	Yao (GLNP15)	GMW
AES	LAN	**7**	27.88
	EU-US	**130.4**	2917.3
	EU-AUS	**377.92**	7325
SHA1	LAN	**14.5**	1651.27
	EU-US	**144.41**	187080.6
	EU-AUS	**396.52**	493143.1

Table 5. Yao vs. GMW (Runtime in ms)

B Transport Layer Protocols

In this section we review TCP and other proposals for protocols for transport layer communication.

TCP exhibits unsatisfactory performance on intercontinental links with high bandwidth delay product (BDP) and links with high packet loss rates. This performance degradation may not be significant if the amount of data transferred is low but in applications with megabytes of data being transferred, this performance degradation can be critical and yet TCP is still being used even though it "only just works" [31]. Though approaches to improve TCP's congestion control has been proposed [29,12,55,55,41,56,13], they fail to perform consistently when the network assumptions on which they are based is violated.

Most congestion control algorithms rely on indicators such as packet loss and delay to determine the congestion in the network. TCP-CUBIC [29], used by default in Linux kernels since version 2.6.19, is a loss-based congestion control algorithm in which the congestion window is a cubic function of time since the last packet loss. It fills the available buffer capacity leading to large queueing delay. BBR [13], a recently proposed congestion control algorithm, does not rely on loss or delay indicators but estimates the available bandwidth to determine the sending rate and tries to avoid queueing delay. BBR is still a work in progress and in its current state, issues such as increased queuing delay and massive packet losses have been observed [34].

There are alternatives to TCP such as SABUL [24,25,26], SPDY [23], QUIC [51] and Performance-oriented Congestion Control (PCC) [19]. SABUL [25,26] is a UDP-based protocol with the goal of utilizing the bandwidth efficiently. Congestion control and reliability control mechanisms of SABUL are built on top of UDP in the application layer, making it easy for use by application developers. It was designed for transferring large data streams over high-speed wide area networks but has since been updated to support the commodity Internet.

B.1 TCP-CUBIC.

TCP-CUBIC uses a cubic function to manage the congestion window size. It uses a convex component as well as a concave component for increasing the window size. After window reduction on packet loss, the concave component contributes to the rapid increase in the window size. As it nears the window size W_{max} before the packet loss, the growth slows down to zero. Then TCP-CUBIC begins looking for more bandwidth and the window grows slowly away from the previous W_{max} due to the convex component. Considerable time is spent between the concave and convex growth regions as the network stabilizes to probe for more bandwidth. The congestion window of TCP-CUBIC is given by the following equation:

$$W = C(t - K)^3 + W_{max} \tag{1}$$

where C is a scaling factor, t is the elapsed time from the last window reduction, W_{max} is the window size before the last packet loss, and $K = \sqrt[3]{W_{max}\beta/C}$, where β is a constant multiplication decrease factor applied for window reduction at the time of loss event. For the rest of the paper, we refer to TCP-CUBIC as TCP.

B.2 SABUL.

SABUL adds reliability at the application layer on top of UDP. It makes use of timer-based selective ACK and packet-based sequencing. Packet loss is indicated through the use of an negative acknowledgement (NAK). It makes use of a hybrid rate-based congestion control and window-based flow control. Rate control, which manages packet sending rate, is triggered at every constant interval (SYN) while window control, which limits the number of unacknowledged packets, is triggered when an acknowledgement packet is received. Packet sending rate is controlled through an additive increase and multiplicative decrease algorithm. Multiplicative decrease is by a factor of 1/9, while additive increase is independent of the RTT. Additive increase factor is given by the following equation:

$$\alpha(x) = 10^{\lceil \log(L - C(x)) \rceil - \tau} \cdot \frac{1500}{S} \cdot \frac{1}{SYN} \tag{2}$$

where x has the unit of packets/second. L is the link capacity measured by bits/second. S is the SABUL packet size (in terms of IP payload) in bytes. $C(x)$ is a function that converts the unit of the current sending rate x from packets/second to bits/second ($C(x) = x * S * 8$). $\tau = 9$ is a protocol parameter.