An End-to-End System for Large Scale P2P MPC-as-a-Service and Low-Bandwidth MPC for Weak Participants*

Assi Barak† Martin Hirt‡ Lior Koskas† Yehuda Lindell†

August 20, 2018

Abstract

Protocols for secure multiparty computation enable a set of parties to compute a joint function of their inputs, while preserving *privacy*, *correctness* and more. In theory, secure computation has broad applicability and can be used to solve many of the modern concerns around utilization of data and privacy. Huge steps have been made towards this vision in the past few years, and we now have protocols that can carry out large computations extremely efficiently, especially in the setting of an honest majority. However, in practice, there are still major barriers to widely deploying secure computation, especially in a decentralized manner.

In this paper, we present the first end-to-end automated system for deploying large-scale MPC protocols between end users, called MPSaaS (for *MPC system-as-a-service*). Our system enables parties to pre-enroll in an upcoming MPC computation, and then participate by either running software on a VM instance (e.g., in Amazon), or by running the protocol on a mobile app, in Javascript in their browser, or even on an IoT device. Our system includes an automation system for deploying MPC protocols, an administration component for setting up an MPC computation and inviting participants, and an end-user component for running the MPC protocol in realistic end-user environments. We demonstrate our system for a specific application of running secure polls and surveys, where the secure computation is run end-to-end with each party actually running the protocol (i.e., without relying on a set of servers to run the protocol for them). This is the first such system constructed, and is a big step forward to the goal of commoditizing MPC.

---

*This work appeared at ACM CCS 2018.

†Dept. of Computer Science, Bar-Ilan University, Israel. assaf.barak@biu.ac.il, liork.cryptobiu@gmail.com, lindell@biu.ac.il. Supported by the European Research Council under the ERC consolidators grant agreement n. 615172 (HIPS), by the BIU Center for Research in Applied Cryptography and Cyber Security in conjunction with the Israel National Cyber Directorate in the Prime Minister's Office, and by the Alter Family Foundation.

‡ETH Zurich, Switzerland. hirt@inf.ethz.ch

One of the cryptographic difficulties that arise in this type of setting is due to the fact that end users may have low bandwidth connections, making it a challenge to run an MPC protocol with high bandwidth. We therefore present a protocol based on Beerliova-Trubiniova and Hirt (TCC 2008) with many optimizations, that has very low concrete communication, and the lowest published for small fields. Our protocol is secure as long as less than a third of the parties are *malicious*, and is well suited for computing both arithmetic and Boolean circuits. We call our protocol HyperMPC and show that it has impressive performance. In particular, 150 parties can compute statistics—mean, standard deviation and regression—on 4,000,000 inputs (with a circuit of size 16,000,000 gates of which 6,000,000 are multiplication) in just 45 seconds, and 150 parties can compute a circuit over $GF[2^8]$ (which can be used for a Boolean computation) with 1,000,000 multiplication gates and depth-20 in just 2 seconds. Although our end-to-end system can be used to run any MPC protocol (and we have incorporated numerous protocols already), we demonstrate it for our new protocol that is optimized for end-users without high bandwidth.

1 Introduction

1.1 Background

Secure multiparty computation (MPC) allows a set of parties to securely compute an agreed function on their private inputs. Informally speaking, security means that the secrecy of the inputs and the correctness of the output is maintained, even in the presence of some *malicious* parties who may deviate from the protocol.

Feasibility of secure multiparty computation (MPC) was proven in the late 1980s, demonstrating that *any* functionality can be securely computed in the presence of a malicious adversary who may arbitrarily deviate from the protocol specification in an attempt to breach security. Secure computation can be achieved with computational security [33, 15] for any number of corrupted parties, with perfect security as long as less than one third of the parties are corrupted [4], and with statistical security for an honest majority assuming broadcast [26]. These powerful results prove that MPC can solve the problem of data utilization while preserving privacy. However, they do not address the problem of concrete efficiency, and the ability to carry out MPC in practice.

Over the past decade there has been huge progress on the problem of constructing efficient MPC; see [6, 24, 14, 12, 22, 23, 18, 25, 1, 28, 21] for just a few examples. The current state of affairs is that it is possible for parties to carry out huge computations (with millions and even billions of Boolean or arithmetic gates) in reasonable time.

The above demonstrates that secure computation can be made practical, and indeed there are a number of startup companies offering MPC solutions, and other large companies using MPC for internal computations. This is of great importance, and we are already seeing the use of MPC in real-world business solutions. However, the promise of MPC goes far beyond the B2B context, and is a perfect fit for peer-to-peer decentralized settings.

In its basic theory, MPC enables an arbitrary set of end-user parties to carry out joint computations on their own personal private data, without revealing it to anyone else. This is extremely attractive, since parties participating in MPC protocols do not need to trust anyone beyond themselves.

However, despite the above, there are still significant obstacles to deploying peer-to-peer MPC in practice:

1. It is unclear how in practice a set of parties can synchronize what protocol to run and when.
2. Most MPC implementations to date require the installation of software, and cannot be run on end-user devices like Mobile phones or in a browser, which is the way that most end users prefer to work.
3. Almost all MPC protocols require all parties to be online at the same time. This is a serious obstacle to carrying out large-scale MPC between end users who may reside in different time zones.

These above issues, and more, must be solved before peer-to-peer MPC can be used in practice.

In order to illustrate the above, we refer to an extremely interesting example of a real MPC deployment that was carried out recently. The Boston Women's Workforce Council used MPC to carry out a study of the gender wage gap among employers within the Greater Boston Area [20]. This study required obtaining (anonymized) employee compensation from private companies, which was refused due to requirements to protect sensitive information. After considerable legal and technical work, secure multiparty computation was used, and enabled the study to be carried out. In the specific case of [20], two servers received *shares* of the inputs from all and were used to run the entire computation. This means that all parties involved had to trust that the two servers would not collude, or be simultaneously breached. Indeed, in many cases, such a model can be legally problematic due to questions of trust (who runs these servers) and the regulatory requirements for each party to protect its own data.

A far more compelling way of carrying out the above study would be to have each company run its own copy of the MPC. However, as described by [20], none of these companies would install software and would only use a browser interface. Furthermore, it took great effort on the part of the Boston Women's Workforce Council to get the companies to agree to this model. It would be much easier, and more effective, to ask end users to provide their own information and run the MPC themselves. This would enable the study to span the entire nation, and even world, and give a far bigger and better picture of the gender wage gap problem.

1.2 An End-to-End System for P2P MPC

In this paper, we describe a full-fledged end-to-end MPC system that can be used to run large-scale secure computations between end users on their own personal data. Our aim was to build an MPC system that works the way *modern software works*. As such, end users can participate in MPC executions on platforms that real end users use, a mobile app or web browser (with *any* browser), and without requiring software installation or allocating servers. In addition the MPC system can be offered in the software-as-as-service model, meaning that there is a *cloud service provider* who offers the MPC service, *subscribers* who can purchase/use the service and initiate MPC executions, and *end users* who can participate in these executions. Crucially, end users need not trust anyone but themselves, since they are active participants in the MPC. The system must provide elasticity, be low cost, and not require out-of-pocket expenditure in order to participate (i.e., in the OPEX model).

We call the system MPSaaS, or MPC system-as-a-service, and it can be used to setup and deploy end-to-end MPC computations in practice. The system works by a provider (or anyone running the system) publishing an upcoming MPC execution that anyone can register to join. The existing system uses Google or Facebook authentication to register, but this is just used as a demonstration of capability. After registering, users can view upcoming executions, and can join any that they wish (or are invited to). At this point, the user makes a decision as to whether or not they will be online during the actual MPC execution. If yes, then they can run the MPC from their mobile phone or browser (and will receive an automatic notification a little before the execution is supposed to begin). The secure computation is then run on their phone itself (or within their browser), and their *private input never leaves the device* (or browser boundary). If they cannot guarantee being online, then they can choose to connect a cloud VM instance to the MPC which will receive their input and run the MPC for them (this can be done in two ways, as will be described below); once again, their private input never leaves the VM instance. We stress that the system generates the circuit on-the-fly when the computation begins, so if a user who is supposed to join the execution does not show up, this will not prevent the execution from running.

In order to achieve the above, MPSaaS is comprised of the following components:

1. *The automation backend:* In order to deploy large scale secure computation, it is essential that instances of the MPC protocol be automatically delivered and executed. We implemented a backend that can be used to deploy general MPC executions, in a fully automated way. The capabilities of the backend include: a fully automated environment setup in AWS or Azure, multiple execution coordination (including setting up parties by bidding for instances, running the MPC protocol, and tearing the parties down upon completion), and monitoring, result collection and reporting (in order to obtain results, as well as reporting on running time and other resource usage). The backend works with arbitrary protocols and we have already incorporated 10 different protocols into the sys-

tem. It is possible to setup complex experiments with executions for different numbers of parties on different circuit types in different topologies in minutes, and automatically receive back a summary of the results.

We believe that this backend is of independent interest, and will be very useful for the MPC community at large. As with all of our software, we stress that the backend automation system is open source and will be made freely available to all.

2. *MPSaaS administrator component:* This component enables a provider (or anyone running the open source software) to invite participation in a secure computation. The administrator can publish proposed secure computations, track how many users have registered (and potentially which users), and obtain the results (we stress that participants also receive all results as part of the MPC protocol that they run). The system is not intended for providing anonymity regarding the identity of the participants, but rather focuses on guaranteeing privacy of their inputs. This has the advantage of making it easier to prevent sybil attacks and bogus registrations. The administrator component is linked to the backend, to actually deploy the MPC executions that are published.

3. *MPSaaS end-user execution:* The end-user component consists of a software layer for running MPC on end-user devices, including mobile phones, browsers and even on IOT devices. We demonstrated MPC on Raspberry Pi for the latter, and this can be used for secure computations over data gathered by IOT devices.

We demonstrated the above system by constructing an MPC private polling solution for carrying out surveys while keeping inputs private. This solution has very broad applicability, and could be used to run a gender wage gap study as discussed above, but with each individual party actually running the MPC protocol and therefore not needing to trust anyone with their data.

1.3 HyperMPC – A Low-Bandwidth Protocol

One of the challenges that arises when attempting to deploy end-user MPC protocols is due to the large amount of communication that can be required. This is especially a problem when users run on mobile devices with limited bandwidth (and costs encountered) and on IOT devices that need to save power. There has been great progress in this direction in recent years for the case of an honest majority. For example, [2] achieve secure computation over Boolean circuits with security in the presence of malicious adversaries at the cost of just 7 bits per AND gate. However, their protocol only works for 3 parties (with an honest majority), and only achieves this efficiency when running secure computations with massively parallelizable circuits or when amortizing over many secure computations. Thus, such protocols do not suit the type of environment needed for MPSaaS, where many parties participate in a single execution.

Highly efficient protocols were recently presented for the setting of an honest majority and computation over arithmetic circuits in [21] with additional improvements in [10]. These protocols have very impressive performance in practice, especially when working over large fields. However, on smaller fields, they require considerably more bandwidth. In particular, they are not efficient for Boolean-circuit computations, which are very common.

A new protocol with low bandwidth. In order to address this issue, we present a new protocol for information-theoretic multiparty computation that is secure in the presence of an adaptive, malicious adversary corrupting up to $t < n/3$ parties. Our protocol is based on the protocol of [5], with significant optimizations to achieve practical efficiency. In particular, since robustness cannot be achieved in the asynchronous setting that we consider without great cost, there is no need to take (expensive) preparations to recover from adversarial attacks. Rather, if cheating is observed, then the parties simply abort the computation. Hence, as opposed to the protocol of [5], we employ neither the player elimination framework [16] nor circuit randomization [3]. This significantly simplifies the implementation (an important feature in practice) and improves the performance.

We call our protocol HyperMPC, since one of the main techniques utilized to achieve such high efficiency is *hyper-invertible matrices* [5]. A matrix is hyper-invertible if every non-trivial square sub-matrix is invertible. Hyper-invertible matrices allow for extremely efficient *verifiable* secret-sharing (a small factor slower than just sending the shares to the parties), extremely efficient randomness generation ($n - t$ random values at the cost of n sharings), and also extremely efficient public reconstruction (n times faster than just sending the shares to each party). As a result, the communication complexity of each party is *constant per multiplication gate*. Thus, the overall communication is linear, meaning that the overall number of field elements sent per *multiplication gate* is $O(n)$ (when we measure the communication complexity, by default, we count the communication of *all parties together*). We describe the hyper-invertible matrix technique in detail in Section 3.1.3.

HyperMPC follows the usual offline/online pattern: In the offline phase, a number of so-called random double-sharings are generated [13, 5]. A random double-sharing is a pair of two (valid) sharings, one with degree t, and one with degree $2t$, of the same uniformly distributed (and unknown) value. The generation of these double-sharings can be performed in parallel. In the online phase, the actual circuit is computed. For input gates, multiplication gates, and random gates, one of the prepared double-sharings is consumed (where for input and random gates, only the t-sharing is used). Due to the linearity of the secret-sharing scheme, linear gates can be evaluated locally. Finally, output gates use standard secret reconstruction.

We prove the following theorem:

Theorem 1.1 *Let f be an n-party functionality, and let C be an arithmetic circuit over field $\mathbb{F}$ that computes f. Then, protocol* HyperMPC *computes f with perfect security and* with abort*, in the presence of an adaptive, malicious adversary corrupting $t < n/3$ parties.*

The exact communication complexity of HyperMPC *is*

$$13n \cdot c_M + 13.5n \cdot c_I + 10n \cdot c_R + n \cdot c_O + 2n^2 \cdot D_M + 10n^2$$

field elements (sent by all parties overall), where c_m *denotes the number of multiplication gates,* c_I *the number of input gates,* c_R *the number of random gates,* c_O *the number of output gates, and* D_M *the multiplicative depth of* C*.*

The basic version of the protocol that we present does not achieve fairness, but this can be achieved with almost no additional cost as we describe in Section 3.8.

Perfect security and concrete efficiency. Our protocol achieves *perfect security*, in contrast to previous honest majority protocols like [21, 10] that are only statistically secure. Although this seems to not be an issue to be concerned with in practice, it turns out that it has real benefits. In particular, all known efficient protocols with statistical security have the property that the adversary can cheat with probability $1/|\mathbb{F}|$ where $\mathbb{F}$ is the field over which the arithmetic circuit is defined. Thus, in order to obtain the required level of security over a small field, the verification parts of the protocol need to be repeated. For this reason, the protocol of [10] requires each party to send only 12 field elements per multiplication gate (*overall*, not to each other party) when working over a large field (for which $1/|\mathbb{F}|$ error is acceptable). In contrast, the analogous protocol in [10] for small fields requires each party to send $6+8\delta$ field elements per multiplication gate where $\delta > \frac{\sigma}{\log(|\mathbb{F}|/3)}$ for error $2^{-\sigma}$. Concretely, if 2^{-80} error is desired, and $\mathbb{F}$ is 16 bits long, then each party must send 46 field elements per multiplication gate. Furthermore, if the parties wish to run a Boolean computation, then as described in the beginning of Section 3.1.1, a Boolean circuit can be easily embedded into the field $GF[2^\kappa]$ and run within an arithmetic-circuit protocol (as long as 2^κ is larger than the number of parties). If [10] is used, then $GF[2^8]$ suffices for up to 250 parties. However, in order to achieve 2^{-80} error, this would require each party to send 86 field elements per multiplication gate. In contrast, for *all fields*, in our protocol each party sends only 13 field elements per multiplication gate. Thus, for this example of a Boolean circuit and $GF[2^8]$, our protocol has *one seventh* the bandwidth of the previous best protocol of [10]. In Section 4.2, we demonstrate that HyperMPC has excellent performance, especially for small fields.

Related low-bandwidth protocols. Our protocol is based on the linear communication protocol of [5], with significant simplifications made possible by the fact that we focus on security with abort. We chose [5] as basis because it is at the same time very simple and very efficient (and has low constants). As we have discussed, by the fact that the protocol is perfectly secure, one does not need to worry about field sizes and error probabilities. Other protocols, like e.g. [13], offer similar asymptotic efficiency, but require probabilistic consistency checks, which complicate the implementation and increase the constants in the

communication complexity. We are not the first to base a concretely efficient construction on [5]. In particular, the VIFF system presented in [11] is based on [5] and uses many of their ideas. However, the actual construction of [11] has *quadratic* communication complexity, in contrast to *linear* communication complexity in our protocol. For a large number of parties, which is the goal for our end-to-end system, this difference is paramount. Another related protocol discussed above is that of [10], which improves on [21]. The protocol of [10] has comparable communication to HyperMPC for large fields, but much higher communication for small fields. We stress, however, that [10] is secure for any $t < n/2$ whereas we require $t < n/3$.

1.4 Implementation and Experiments

We implemented both the MPSaaS system and the HyperMPC protocol. In Section 2, we describe the system design and the platforms upon which it was implemented. Then, in Section 4.1 we provide running times of executions with different end-user endpoints. We are currently extending these experiments and will include them in the full version of this paper.

In Section 4.2, we provide running times of executions of the HyperMPC protocol for a number of settings. In particular, we provide results for experiments with circuits of different sizes and depths for 150 parties, for experiments computing statistics with 10–150 parties at increments of 10, and a single experiment demonstrating that HyperMPC can even support an MPC with 500 parties in reasonable time. To the best of our knowledge, this is the largest number of parties ever reported for an MPC protocol, in any setting.

2 The MPSaaS System and PrivatePoll

2.1 Overview

In the interconnected world of browsing, social networks, mobile applications and cloud services, the focus is on usability and experience. While MPC is a promising technology with prominent privacy guarantees, use cases in research are mostly Business-To-Business (B2B) or Business-To-Customer (B2C). This limits the applicability of MPC, making it unsuitable for many real world use cases. The goal of our system is to enable true P2P MPC on a large scale for the first time. We demonstrate the use of MPC for peer-to-peer applications involving end users over the Internet. In order to demonstrate our system, we constructed an application called **PrivatePoll** that enables end users to participate in *private* polls and surveys. The incentive for the user to participate is to receive the results of the poll (or simply to support the study being carried out), and their privacy is provable guaranteed via secure computation. One example of where this could be used is a salary poll among peers in the same professional category and skill level.

MPC-as-a-service. We developed an application that can be deployed in a standard cloud service scenario, with necessary adaptions for MPC. An MPC Cloud Service Provider (MPC-CSP) can provide the service to register participants in an upcoming MPC computation. Then, upon registering for the computation, each user is provided with the means necessary to actively participate as one of the parties in the executed MPC protocol. Our application enables end users to participate using their Google or Facebook account (oAuth2 provider), and are not required to fill in additional information, to ensure usability and increase participation level. Of course, this can be easily modified to accept alternative forms of authentication (e.g., LinkedIn login in order to run MPC computations for a salary survey, or Active Directory for running MPC computations between university students with university accounts). Using such authentication mechanisms also helps to prevent double registration and bogus users.

MPC end-user interaction. Our application is novel in the way end-users interact with the service. Specifically, the MPC functionality can be run directly inside the user's browser or in a mobile app, as these are the methods used for virtually all modern user interactions. In addition, users can register a cloud instance to which they upload their input, and which runs their party in the MPC protocol. This latter mode is useful for users who do not wish to (or cannot) guarantee that they will be online (via their mobile or browser) at the time of the actual MPC execution. In more detail, the following execution modes are offered to end users:

1. *Mobile App:* in this mode, the user enters her private input in a Mobile app. The MPC protocol is executed inside the mobile (MPC code ported to mobile platform), ensuring that the private information does not leave the device boundary.

2. *Online In-Browser:* in this mode, the user enters her private input in a Web Form. Then, the MPC protocol is executed in Javascript inside *any* browser, ensuring that private information does not leave the Browser boundary.

3. *Cloud Instance:* In both the mobile app and browser modes, the user must be online at the time of actual execution, which is a major problem for real-world deployment. In order to overcome this obstacle, we enable users to define a cloud instance of their own to run the MPC protocol for them. In our implemented solution, users can participate by using a cloud instance that they own (like an AWS Docker instance), or they can choose to use an instance generated for them by the MPC service provider. The latter is of course less secure (since the user sends their input to a machine controlled by the provider), but is an option provided nevertheless; we discuss this in more detail in Section 2.5. This simple solution solves the acute problem that all MPC participants must be *simultaneously online* in order to run the protocol, since parties who cannot be online (e.g., they cannot guarantee that their mobile is connected, etc.) can define cloud instances that are always online.

4. *IoT Device:* This mode considers MPC involving IoT devices that take sensitive measurements which are input to the MPC computation. The MPC protocol is run inside the device, ensuring that private information never leaves the device. Our implementation currently works for Raspberry Pi3; if the remote device is inactive then it can be automatically woken for the MPC execution. (This can be used for drones, and can be extended to weaker IoT devices like wearables in the future.)

Deployment and delivery of MPC software. To increase user trust, secure artifacts can be used. The MPC Mobile App (apk) can be delivered via Google Play, or equivalently as an IOS App via the Apple App Store, the Docker Image which executes cloud instance for a user can also be signed to ensure trusted delivery, and Subresource Integrity via CDN can be used on a browser. In addition, the code of the MPC protocol being used should be open source to increase trust. We stress that our *entire* system, including all components, is completely *open source* and available for review (and use). An independent review by an organization like EFF.org can be carried out on specific MPC modules to further increase trust.

The overall system. The system components and their interactions are shown in Figure 1.

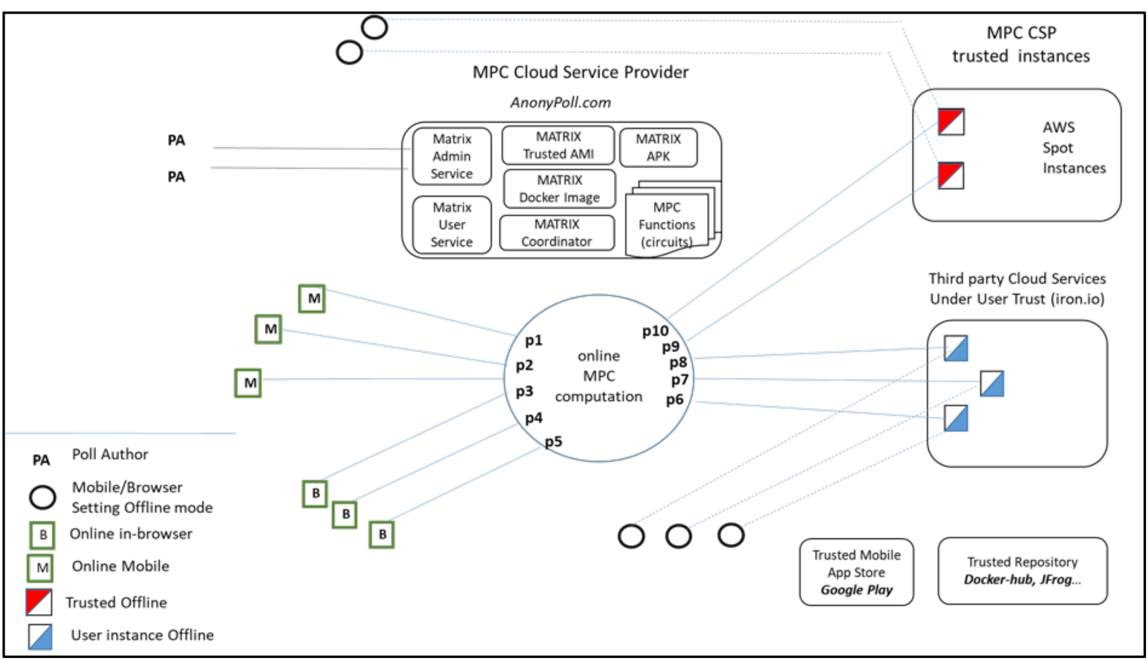

Figure 1: Components of the system, and their interaction.

2.2 MPSaaS Backend – the MATRIX Automated Execution System

The backend service, called MATRIX, provides the ability to automatically deploy large-scale MPC experiments in the cloud. As such, this component is of independent interest to the community. In particular, running multiparty experiments can be very time-costly if done manually, causing researchers to either spend a large amount of effort on this menial task or to cut corners. As an example of the capabilities of the system, we recently ran a protocol experiment for an MPC protocol for 10–100 parties with increments of 10, with 12 different configurations per number of parties (a configuration being a mix of a circuit and execution parameters), and 5 repetitions of each experiment for statistical accuracy, in just 17 minutes (observe that this consists of 600 different executions, with different numbers of parties and different configurations each time). We stress that this includes the *entire time* to setup the experiment and get the results back. Given the automation capabilities of MATRIX, this serves as the perfect backend for the MPSaaS system. We now proceed to describe the different components of MATRIX; throughout the explanation Figure 2 can serve as a guide for the different components and capabilities. We remark that MATRIX works in an identical manner for running experiments, and for real protocol executions (of course, for real executions, there is no need to collect logs containing running times, and so on).

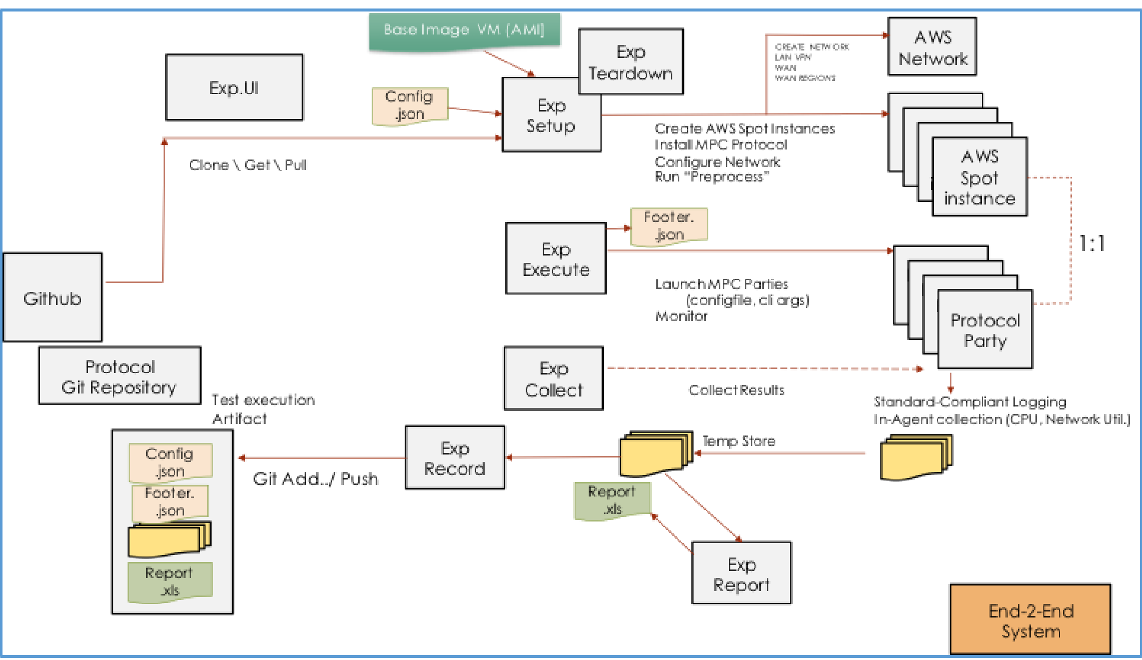

Figure 2: The MATRIX backend.

The key features of the MATRIX backend are as follows:

Protocol compatibility: The system includes a well-defined "contract" that must be fulfilled by an MPC executable that is run in the system. This contract ensures that it runs correctly in the Test Environment, and includes definitions for configuration, result reporting and fault handling. This contract can be easily implemented in wrapper classes and protocols, and is all that is needed to use MATRIX for a new protocol. We have already incorporated 10 different MPC protocols into the system, for both semi-honest and malicious adversaries, and for honest majority and dishonest majority. Specifically, the protocols incorporated so far are semi-honest GMW, semi-honest Yao, semi-honest BMR, malicious Yao [22], SPDZ [12], the protocols in [1, 2, 21, 10] and HyperMPC from Section 3. In order to incorporate new protocols in MATRIX it suffices to write a python script that accepts the command line arguments and does the following:

1. Converts formats (parties' file IP addresses and Ports to its internal parties' presentation),

2. Converts input circuit if required, and

3. Launches the actual protocol executable (using the party ID and other configuration flags).

In addition, in order to utilize the result reporting feature, it is necessary to add code to the protocol to log in our format, or write a script to convert its internal log files to our format. We incorporated the SPDZ-2 protocol [12] in one day.

Fully Automated Environment Setup across regions: MATRIX provides full automation to create instances for MPC execution, and immediately tear them down afterwards (reducing cost). In detail:

1. *Host setup:* We create cloud instances from custom AMI images, and deploy the protocol code, which is then compiled on the host for optimal native performance (e.g., utilizing Intel-AVX instructions where available). MATRIX is integrated with Github or any GIT provider to pull experimental or master versions as required. We select the EC2 instance type based on the capabilities and requirements of the protocol, and create standard or Spot Instances based on a cost strategy. For example, when the actual time of the execution is not significant, one can execute during off load hours (say weekends), using spot-instances issued in 6-hour quotas. This results in significant savings, assuming many executions are run (and so the 6-hour quota is utilized).

2. *Network setup:* MATRIX automatically sets the network for all parties, defining the security groups and rules as required. Once a network is set up, it collects information on allocated addresses and DNS names in order to generate a parties file that is used in the execution. MATRIX also supports setting up a VPC (virtual private cloud, ensuring that all pairwise connections between parties are encrypted and authenticated).

However, we stress that this comes at a high monetary cost, and so only makes sense for real-world applications and not experiments.

The configuration file to define the experiment can specify the regions that the parties are deployed, enabling experiments over fast and slow networks to be seamlessly carried out.

Execution coordination: MATRIX launches the MPC process on each hosts, automatically providing it with the details of all participating parties, and the relevant parameters for computation (where amongst other things parameters include a circuit description, or code to be compiled). MATRIX can receive a script for automatically generating circuits based on the number of parties, and then will provide the circuits of the correct size for each experiment. This is important in real-world systems like MPSaaS, since the circuit needs to be generated for the exact number of parties who are online when the experiment starts. MATRIX takes care of this issue automatically. We stress that MATRIX can run any MPC protocol, and it does not have to be circuit based (e.g., specific protocols like PSI can be run, as well as SPDZ-like systems that generate gates to be computed on the fly; in fact, as we have mentioned, SPDZ-2 together with its run-time [12] is already incorporated in MATRIX).

MATRIX receives an experiment configuration file that defines a series of experiments (executions). For each experiment, the number of repetitions is defined, how many parties participate, the location of the parties (cloud regions), the machine type for all parties, and the protocol parameters (for example, the circuit that they execute if the protocol is a circuit-based protocol). When spot (or low priority) instances are desired, then the configuration also includes the maximum price that you are willing to pay for the instances (in the cloud bidding scheme).[1]

The MATRIX binding to a specific cloud vendor is abstracted to a small layer of the code, and can be deployed to any vendor providing a complete automation API. MATRIX was first deployed in AWS, and the time to port it completely to Azure (using Azure CLI 2.0) was just 2 person days.

Monitoring, result collection, reporting and fault handling: MATRIX monitors the execution of the MPC program across all the hosts, and detects exit signals that indicate cheat detection, protocol breach or simply machine failure. In addition, the system collects MPC logs written during the protocol execution (according to the format in the contract). Such logs typically include micro-benchmarking of the different protocol phases. For example, it is possible to measure the times for peer-2-peer communication setup, the input sharing phase, online computation, and so on. The granularity of the logs and measurements taken is at the full discretion of the protocol, and typically includes running-time and actual bandwidth sent and received. MATRIX also automatically measures the memory usage of each machine.

[1] Although the system receives a maximum price, in AWS it begins by checking the latest winning price for the instances desired, and bids that price first, if it is lower. By experience, in almost all cases, the last winning price was sufficient, resulting in considerable cost savings.

All of the results are recorded by MATRIX in an ElasticSearch database, which enables users to review performance dashboards and compare the performance of different protocols, running on different machines in different regions and so on, using a tool like Kibana; see Figure 3 for an example. MATRIX also generates spreadsheet reports for post-execution analysis.

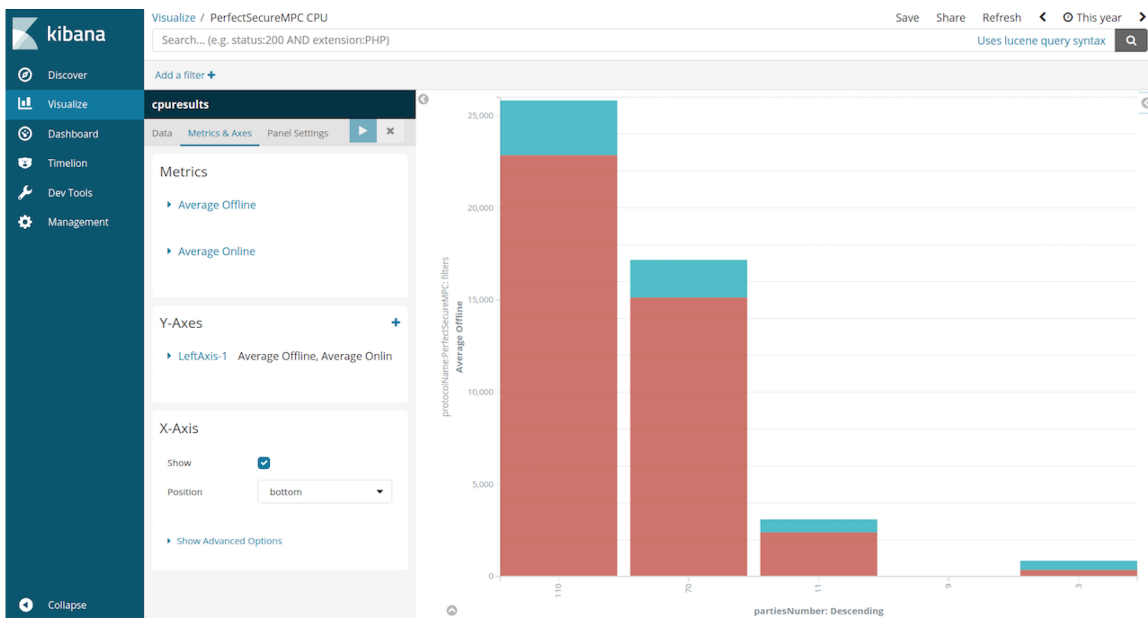

Figure 3: MPC analytics using Kibana.

2.3 MPSaas Administration Component

The MPSaaS Administrator component is used to create, manage and execute MPC protocols as desired. The MPSaaS administrator is the party who initiates the execution, and invites participants. In the PrivatePoll example, this could be an HR consulting company authoring Salary Polls, or it could be a researcher wishing to survey gender wage gap. The administrator can purchase these services from an MPC-CSP (service provider) or may act itself as a service provider using the open source platform. We stress that the administrator has no access to private inputs, since each participating party runs the MPC protocol locally, and thus need *not* be trusted at all. The administrator component contains all of the capabilities necessary to manage the MPC executions, and is connected to the administration capabilities of the MATRIX system. In Figure 4, you can see the administrator page for creating new executions (in our specific example of PrivatePoll these are all polls) and viewing the status of active ones. The administrator can also see the results of completed polls.

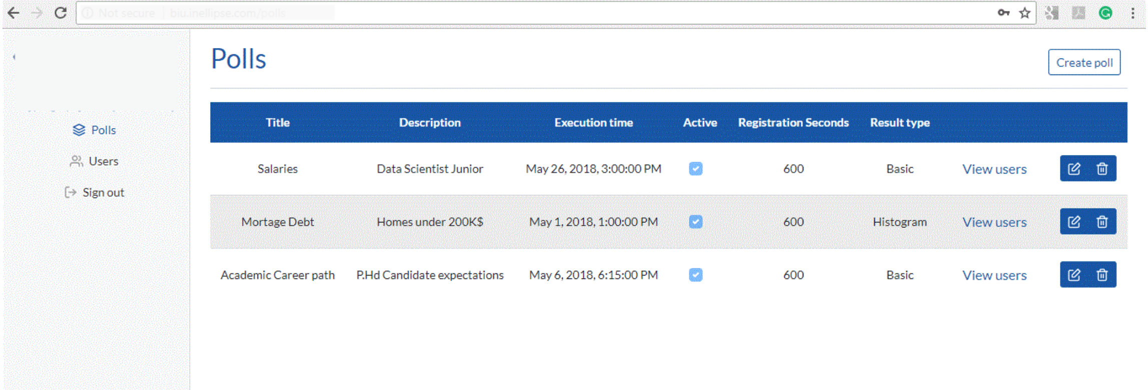

Figure 4: The main admin page for MPSaaS.

In Figure 5, you can see the administrator page for creating or editing a poll. In our Proof-Of-Concept (POC) implementation, each poll includes a single numeric private input (such as "salary").

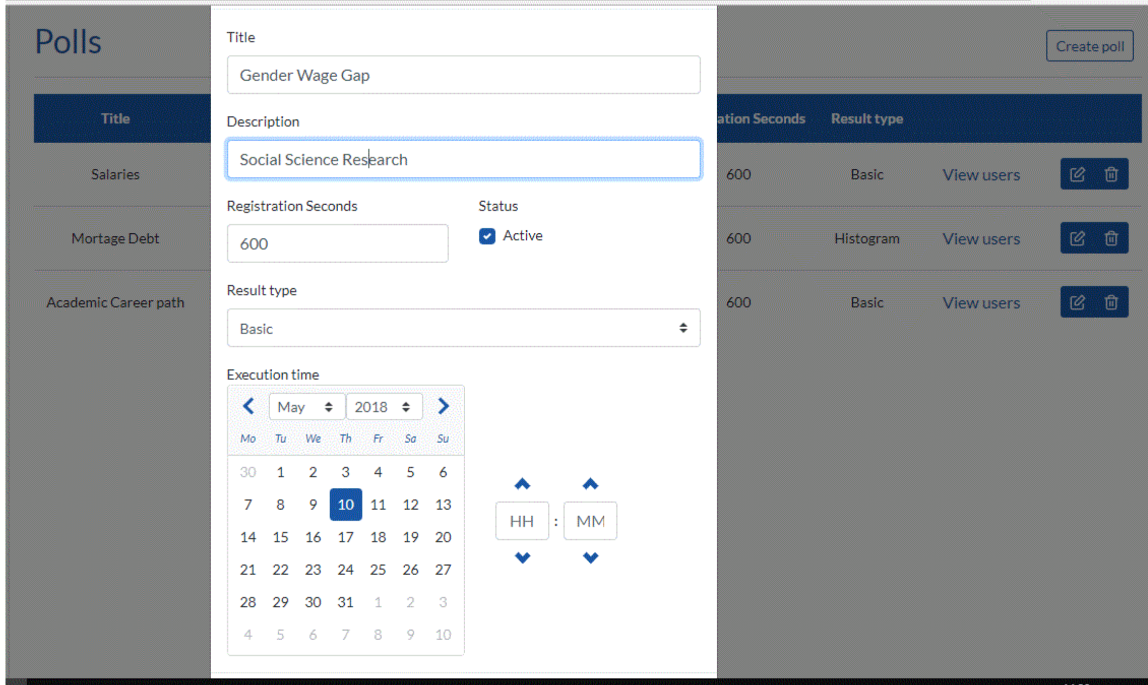

Figure 5: The poll create/edit page.

The result type governs the circuit chosen to evaluate the functionalilty. For our POC, the "Basic" option defines a circuit that evaluates the mean and variance, whereas the "Histogram" result would require a circuit that also sorts or buckets the input values.

We stress that any reasonable MPC solution would also include a minimal number of participants (enforced as part of the protocol) in order to provide privacy. (it is also possible to include differential privacy guarantees by adding the required noise to the *result* inside the MPC; by adding the noise to the result and not to the input, far higher accuracy can be achieved. The best way to do this is left to future work.) Although very basic, we remark that this already suffices for achieving quite general results. In particular, a general study of gender wage gap can be carried out by defining a different poll for each sector/years of experience/sex, and then obtaining the results for each set separately. Since only the salary needs to be protected, this suffices.

Finally, the administrator page also includes the option to view information about the uses registered to the poll; see Figure 6. As a matter of policy, this page can be open to all participants to view, depending on the desired result. Specifically, in many cases, the parties' input is private but the fact of their participation need not be. In these cases, it is useful to allow all parties to view who is registered, to ensure that their private input is really protected under a large aggregation. For simplicity and a seamless login flow, the basic login functionality in the POC is carried out simply using a Google of Facebook ID. Additional attributes may be added to users by creating an extended user profile, if this is desired.

Users

Name	Email	Gender
Admin	admin@test.com	Male
John Smith	john.smith@gmail.com	Male
Steven Smart	steven.smart @gmail.com	

Figure 6: The user participation page.

We stress that none of the above is a recommendation as to how to run a secure MPC system for polling. This depends exactly on the use case, and is an orthogonal issue. Here, we merely demonstrate the capabilities of the system.

2.4 MPSaas End-User Component

The end-user component is used for actually running the MPC protocol. Although some MPC executions have been demonstrated on a mobile platform (e.g., [17]), they are limited and do not include a rich variety of MPC protocols. Since realistic end-user execution is essential for achieving true peer-2-peer MPC, enabling Mobile, IOT and in-browser general MPC execution is a cornerstone of this research. We reviewed available open-source implementations of MPC and found that none work on our platforms. For example, for oblivious transfer, neither `LibOTE`, `SimpleOT` or any TinyOT implementations are available.

It is of course possible to simply reimplement any MPC protocol for mobile and browser environments. However, this is very time consuming and an obstacle to deployment. We therefore searched for general techniques that can be reused to help port existing MPC code to mobile devices and browser. We report on these techniques and the challenges encountered in Appendix A.

2.5 PrivatePoll End User Workflow

In this section, we describe the user experience in participating in an MPSaaS secure multiparty computation, via their mobile phone (after having installed the PrivatePoll mobile app). Users are notified of existing polls via any standard means. In order to join a poll, the user logs in using an existing Google or Facebook account, at which point she is presented the available and completed polls. The user can view completed polls she has participated in previously, and can see active polls. For each active polls, the user may view how many users are registered (see the middle screen in Figure 7 and the number next to the "active" button), and the poll open date and time. The user may then select to join a poll. See Figure 7 for this flow.

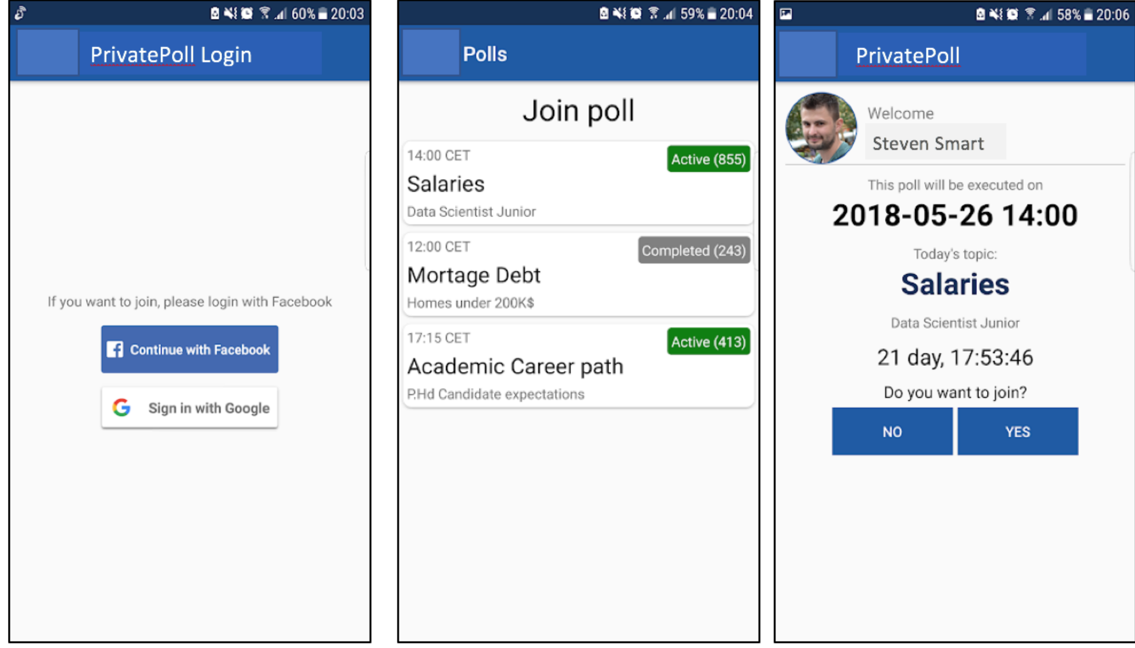

Figure 7: The user login, poll status, and poll join pages.

Once the user selects to join an active poll, she is prompted to select her participation mode. The online mode refers to running a mobile or browser instance (which requires her to be online when the poll begins), whereas the offline mode refers to using a cloud instance. We stress that the cloud instance is always online, thereby allowing the user

herself to be offline during the actual MPC execution. If the user selects online mode, then her request is registered and a notification will be sent to her to login and input her private value prior to the poll. We stress that the MPSaaS system generates the circuit at the time of the actual execution, so if a user does not actual come online, this will not prevent the poll from proceeding.

If the users selects to be offline, then she can choose to either use a hosted instance or her own instance; see Figure 8.

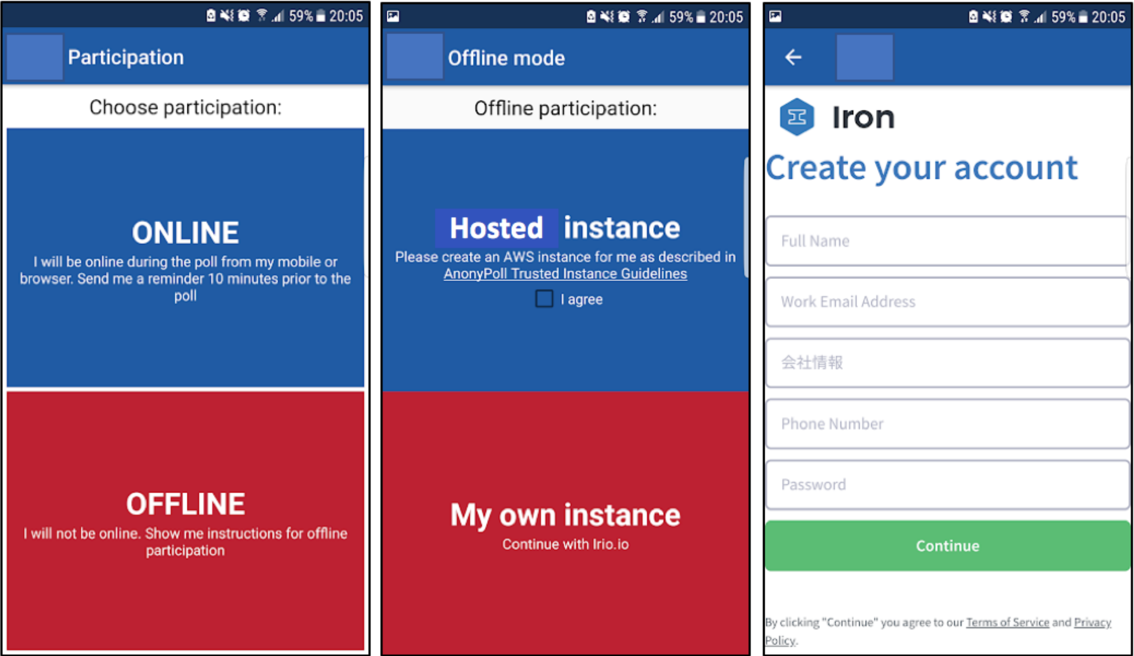

Figure 8: The user instance generation pages.

It is always preferable for a user to choose her own instance, and in this case the user is directed to connect a Docker instance. This Docker instance can be in her own AWS account, or the AWS organizations service can be used. This service enables the service provider (MPSaaS) to set a policy and have users create accounts via the service provider, with the policy transparent to the user. This policy can be set to only allow the provider to monitor usage, thereby preventing the MPSaaS administrator from viewing the data. (Future versions will allow cross-cloud instances so that users are not required to rely on AWS or Azure or any single cloud provider.) If the user insists on not setting up her own instance, then she can choose a hosted instance which is run by the poll administrator. This is clearly not recommended, but may be chosen by a participant who does not consider the poll information to be private (e.g., a faculty member of a public university in the USA, whose salary is anyway public record). In this latter case, the image of a separate instance is automatically deployed for the specific user, on the administrator's AWS account. This

image includes a user facing mini web-server, which will serve the user granted access to the instance (this can be extended so that the user receives an out-of-band one-time-password to login, or some other method).

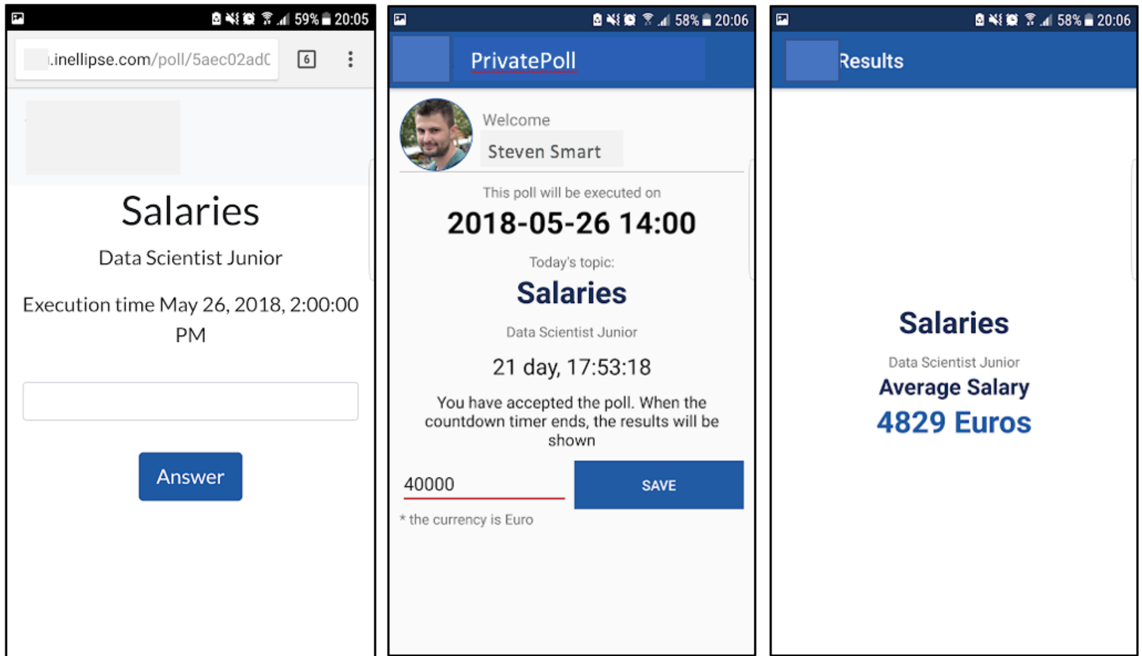

Figure 9: The input/output pages.

In offline mode executions, the user is immediately redirected to the registered instance to enter her input (left screen in Figure 9). In online mode executions, the user is sent a reminder just before the execution begins, at which point she can enter her input and get output (middle screen in Figure 9). In all cases, the user receives the computation result to her mobile app, providing additional motivation for participating (right screen in Figure 9).

3 The HyperMPC Protocol

3.1 Preliminaries

3.1.1 Model

We consider a set $\mathcal{P}$ of n parties, $\mathcal{P} = \{P_1, \ldots, P_n\}$, who are connected by a complete network of secure (private and authentic) channels.

We consider a computation over any finite field $\mathbb{F}$ with $|\mathbb{F}| > 2n$. To every party $P_i \in \mathcal{P}$ a unique, non-zero element $\alpha_i \in \mathbb{F} \setminus \{0\}$ is assigned (e.g. the i-th element in $\mathbb{F}$). We stress

that HyperMPC is perfectly secure, and thus there is no need to work in a large field (this is in contrast to statistically secure protocols that have a basic error of $1/|\mathbb{F}|$ and thus need to repeat checks in case $\mathbb{F}$ is small, making the complexity higher). We also remark that since we can work over any field, Boolean circuits can be embedded into $GF[2^k]$ for the smallest k such that $2^k > 2n$. In this case, as long as we verify that all input values are either 0 or 1, it follows that field addition is equivalent to Boolean XOR and field multiplication is equivalent to Boolean AND.

The function to be computed is specified as an arithmetic circuit over $\mathbb{F}$, with input, addition, multiplication, random, and output gates. A random gate is simply a gate that outputs a random value; it is always possible to generate randomness by adding random inputs provided by the parties, but dedicated random gates can be computed more efficiently. We denote the number of gates of each type by c_I, c_A, c_M, c_R, and c_O, respectively.

We consider a malicious adversary who can corrupt up to t parties for any fixed t with $t < n/3$, and make them deviate from the protocol in any desired manner. The adversary is computationally unbounded, active, adaptive, and rushing. We use the standard definition of security following the ideal/real paradigm [8]. Since our protocol achieves perfect security, it suffices to prove security in the stand-alone model of [8] and this implies universal composability [9], as shown in [19].

3.1.2 Non-Robust Computation

Our MPC protocol does not (and cannot) achieve robustness; the adversary can delay the progress of the protocol arbitrarily, which is equivalent to aborting the protocol. As anyway the adversary can abort the protocol, we allow (honest) parties to abort the computation once they have observed an inconsistency. Inconsistencies only occur in the presence of corruption, and instead of causing inconsistencies the adversary could also have made the protocol abort.

All sub-protocols in this paper achieve non-robust security only, which means that in the presence of malicious parties, honest parties might not complete the sub-protocol (either wait forever or abort after some timeout).

We stress that all other security properties are preserved. As a result, formally, our protocol achieves the notion of *perfect security with abort, in the presence of adaptive, malicious adversaries.*

3.1.3 Hyper-Invertible Matrices

Our protocol uses hyper-invertible matrices [5]. A hyper-invertible matrix is a matrix whose every non-trivial square sub-matrix is invertible. More formally, an r-by-c matrix M is *hyper-invertible* if for all index sets $R \subseteq \{1, \ldots, r\}$ and $C \subseteq \{1, \ldots, c\}$ with $|R| = |C| > 0$, the matrix M_R^C is invertible, where M_R denotes the matrix consisting of the rows $i \in R$ of

M, M^C denotes the matrix consisting of the columns $j \in C$ of M, and $M_R^C = (M_R)^C$.

A hyper-invertible r-by-c matrix M can be constructed as

$$M = \{\lambda_{i,j}\}_{\substack{i=1,\ldots,r \\ j=1,\ldots c}} \text{ with } \lambda_{i,j} = \prod_{\substack{k=1 \\ k \neq j}}^{c} \frac{\beta_i - \alpha_k}{\alpha_j - \alpha_k},$$

where $\alpha_1, \ldots, \alpha_c, \beta_1, \ldots, \beta_r$ are fixed distinct elements in $\mathbb{F}$ (and $|\mathbb{F}| \geq c+r$). The mappings defined by hyper-invertible matrices have a very nice symmetry property: Any subset of n input/output values can be expressed as a linear function of the remaining n input/output values:

Lemma 3.1 **([5])** *Let M be a hyper-invertible n-by-n matrix and $(y_1, \ldots, y_n)^T = M \cdot (x_1, \ldots, x_n)^T$.*[2] *Then for all index sets $A, B \subseteq \{1, \ldots, n\}$ with $|A| + |B| = n$, there exists an invertible linear function $f : \mathbb{F}^n \to \mathbb{F}^n$, mapping the values $\{x_i\}_{i \in A}, \{y_i\}_{i \in B}$ onto the values $\{x_i\}_{i \notin A}, \{y_i\}_{i \notin B}$.*

More details and proofs can be found in [5, Section 3].

An intuitive explanation. The power of a hyper-invertible matrix can best be demonstrated in the linear mapping induced by the matrix. We say that a matrix M with r rows and c columns induces a linear mapping f_M, which takes as input c values $x_1, \ldots, x_c$, and computes as output r values $y_1, \ldots, y_r$, such that $(y_1, \ldots, y_r)^T = M \cdot (x_1, \ldots, x_c)^T$. In other words, for any c input values $x_1, \ldots, x_c$, the mapping defines the corresponding output values $y_1, \ldots, y_r$. If the matrix M is hyper-invertible, then one can show that actually for *any* c values (some inputs, some outputs), the remaining r values are defined (cg. [5, Lemma 2]).

Such a mapping can, for example, be used for efficiently generating sharings of random values. The standard way for generating random sharings is to let every party share a random value (using a linear secret-sharing scheme) and then take the sum of theses sharings. However, this approach is rather inefficient, as it requires n initial sharings to produces a single good sharing. As t of the initial sharings are provided by corrupted parties and hence known to the adversary, the best one can hope for is to produce $n - t$ good sharings from n initial sharings; and this can actually be achieved. Using an $(n - t)$-by-n hyper-invertible matrix, the given n sharings can be mapped to $(n - t)$ sharings such that the resulting sharings are "as random as" the good $n - t$ initial sharings. The reason for this is that the $n-t$ output sharings are determined by a bijective function from the $n-t$ input sharings of the honest parties, where the function is determined by the t sharings of the corrupted parties. Formally, assume that $I \subseteq \{1, \ldots, n\}$ is the (unknown) set of indexes of the corrupted parties, where $|I| = t$. Then, one can write $\vec{y} = M^{\bar{I}} \cdot \vec{x}_{\bar{I}} + M^I \cdot \vec{x}_I$,

[2] Note that $(x_1, \ldots, x_n)^T$ and $(y_1, \ldots, y_n)^T$ are *column vectors.*

where $\bar{I} = \{1, \ldots, n\} \setminus I$ is the set complement of I.[3] As $M^{\bar{I}}$ is invertible, this mapping is bijective.

The above construction requires all initial sharings to be valid (e.g., to have the right degree). This can be ensured by either assuming that the adversary is passive only, or by using some expensive mechanisms to verify the validity of each single sharing. Further exploiting the structure of hyper-invertible matrices, this verification can be done at very little cost. Again, each party shares a random value, but this time, we allow corrupted parties to distribute invalid sharings (which is equivalent to using polynomials of too high degree). This time, the parties apply an n-by-n hyper-invertible matrix on these initial sharings, and $2t$ of the produced sharings are reconstructed, each towards a different single party. Each party checks whether its sharing is valid, and complains otherwise. If no party complains, then the remaining $n - 2t$ output sharings are used as random sharings.

We argue why the remaining $n-2t$ sharings are (i) valid sharings and are (ii) unknown to the adversary. (i) We know that at least $n - t$ of the initial sharings, and, if no party has complained, also at least t of the output sharings are valid. Due to the property of the hyper-invertible matrix, the other sharings can be expressed as linear combination of those n valid sharings, which implies that they are valid as well (due to the linearity of the sharing scheme). (ii) The adversary knows at most t initial and t output sharings. Fixing these sharings, there is a bijective (affine) function from any other $n-2t$ input sharings to the non-reconstructed $n - 2t$ output sharings. Hence, the produced random sharings are indeed random and unknown the the adversary.

3.2 Protocol Overview

Our protocol is based on [5], adjusted and simplified to better fit our needs. In particular, as in the considered model robustness is not possible, we employ neither the player elimination framework [16] nor circuit randomization [3]. This not only significantly reduces the intricacy of the implementation, but also improves the performance of the protocol by a constant factor of 4, which is significant in practice. See Section 3.9 for a detailed explanation of the difference between HyperMPC and the protocol of [5].

Our protocol follows the usual offline/online pattern: In the offline phase, a number of so-called random double-sharings is generated [13, 5]. A random double-sharing is a pair of two sharings, one with degree t, and one with degree $2t$, of the same random (and unknown) value. The generation of these double-sharings can be performed in parallel.

In the online phase, the actual circuit is computed. For input gates, multiplication gates, and random gates, one of the prepared double-sharings is consumed (where for input and random gates, only the t-sharing is used). Due to the linearity of the secret-sharing scheme, linear gates can be evaluated locally. Output gates use standard secret reconstruction.

[3] We write M_R for the matrix consisting of those rows of M with the row index in R, and M^C for the matrix consisting of those columns of M with the column index in C.

All given protocols are non-robust. This means that in the presence of malicious parties, honest parties might not complete the protocol. They may abort (e.g., when observing cheating) a run forever (e.g., when waiting for a message from a corrupted party). The latter case can easily be turned into abort by implementing some reasonable time-out mechanism.

3.3 Secret Sharing

The secret-sharing scheme used in our protocol is the standard Shamir sharing scheme [27]. We assume that every party P_i is assigned some fixed unique non-zero field element α_i. We say that a value s is d-*shared* if every honest party $P_i \in \mathcal{P}$ holds a share s_i, such that there exists a degree-d polynomial $p(\cdot)$ with $p(0) = s$ and $p(\alpha_i) = s_i$.[4] The vector $(s_1, \ldots, s_n)$ of shares is called a d-sharing of s, and is denoted by $[s]_d$.

We will mostly use sharings of degree t. However, in the multiplication, we also need sharings of degree $2t$.

Below, we provide protocols for sharing and reconstruction. The sharing protocol Share is completely insecure; it computes a correct sharing only if the dealer is honest. Hence, the consistency of the sharing must be checked before the sharing can be used. We provide two reconstruction protocols, one for private reconstruction ReconsPriv, and one for public reconstruction ReconsPubl. Both protocols non-robustly reconstruct sharings of degree d with $d < n - t$. For private reconstruction, the parties simply send their respective shares to the output party. For public reconstruction, the private reconstruction protocol could be invoked once for each output party; however, this would result in a complexity of $\mathcal{O}(n^2)$ field elements. We provide a protocol that allows to reconstruct $n - t$ sharings towards all n parties at an overall cost of $\mathcal{O}(n^2)$ field elements, which corresponds to $\mathcal{O}(n)$ elements per reconstructed sharing.

Protocol Share$(P_D \in \mathcal{P}, s, d)$**.**

1. P_D chooses a random degree-d polynomial $p(\cdot)$ with $s = p(0)$ and sends $s_i = p(\alpha_i)$ to every $P_i \in \mathcal{P}$.

Protocol ReconsPriv$(P_R \in \mathcal{P}, d, [s]_d)$**.**

1. Every party $P_i \in \mathcal{P}$ sends its share s_i of s to P_R.
2. P_R checks whether there exists a degree-d polynomial $p(\cdot)$ such that all received shares lie on it, and if so, computes the secret $s = p(0)$. Otherwise P_R aborts.

The following lemma is immediate, by inspection of the protocol.

[4]By convention, we say that a polynomial $p(x) = p_0 + p_1 x + \ldots, p_d x^d$ is a degree-d polynomial, even if the leading coefficient(s) are zero (and hence the mathematical degree is smaller than d).

Lemma 3.2 *For $[s]_d$ being a valid sharing of s with degree $d < n-t$, the protocol* ReconsPriv *non-robustly reconstructs s towards P_R.* ReconsPriv *communicates exactly n field elements.*

The public reconstruction protocol ReconsPubl takes $T = n - t$ correct d-sharings $[s_1]_d, \ldots, [s_T]_d$ for $d < n-t$, and publicly (to all parties in $\mathcal{P}$) outputs the secrets $s_1, \ldots, s_T$. Every party either learns the correct secrets or aborts. In ReconsPubl we use an idea from [13]: first the T sharings $[s_1]_d, \ldots, [s_T]_d$ are expanded (using a linear error-correction code) to n sharings $[u_1]_d, \ldots, [u_n]_d$, each of which is reconstructed towards *one* party in $\mathcal{P}$ (using ReconsPriv). Then every $P_i \in \mathcal{P}$ sends its reconstructed value u_i to every other party in $\mathcal{P}$, who decodes the received codeword $(u_1, \ldots, u_n)$ to the secrets $s_1, \ldots, s_T$. More concretely, we interpret the secrets $s_1, \ldots, s_T$ as coefficients of a degree-$(T-1)$ polynomial, and the code word $(u_1, \ldots, u_n)$ as evaluation of this polynomial at positions $\alpha_1, \ldots, \alpha_n$, respectively.

Protocol ReconsPubl$(d, [s_1]_d, \ldots, [s_T]_d)$**.**

1. For $i = 1, \ldots, n$ the parties in $\mathcal{P}$ (locally) compute $[u_i]_d$ as:

$$[u_i]_d = [s_1]_d + [s_2]_d\alpha_i + [s_3]_d\alpha_i^2 + \ldots + [s_T]_d\alpha_i^{T-1}$$

2. For every $P_i \in \mathcal{P}$, ReconsPriv is invoked to reconstruct $[u_i]_d$ towards P_i.
3. Every $P_i \in \mathcal{P}$ sends u_i to every $P_j \in \mathcal{P}$.
4. $\forall P_j \in \mathcal{P}$: If the values $(u_1, \ldots, u_n)$ lie on a degree-$(T-1)$ polynomial, then compute its coefficients $s_1, \ldots, s_T$ (using Lagrange interpolation). Otherwise abort.

Lemma 3.3 *For $[s_1]_d, \ldots, [s_T]_d$ being T degree-d sharings with $d < n-t$ and $T = n-t$, the protocol* ReconsPubl *non-robustly reconstructs $[s_1]_d, \ldots, [s_T]_d$ towards all parties in $\mathcal{P}$, i.e., each party who completes the protocol learns the correct secrets.* ReconsPubl *communicates exactly $2n^2$ field elements.*

Proof: As each $[s_i]_d$ is of degree $d < n-t$, the secrets s_i are uniquely defined by the honest parties' shares. Denote the polynomial of degree $T-1$ with the coefficients $s_1, \ldots, s_T$ by f. After Step 1, each $[u_i]_d$ is a degree-d sharing of $u_i = f(\alpha_i)$. By Lemma 3.2, in Step 2 each honest P_i learns the correct secret $u_i = f(\alpha_i)$, and sends it (in Step 3) to every P_j. Hence, P_j now holds n values u_i, where at least $n-t$ of them are from honest P_i's and hence correct. These $T = n-t$ correct u_i's lie on f, and if all values $(u_1, \ldots, u_n)$ lie on a degree-$T-1$-polynomial, then this must be f. Hence, the correct coefficients $(s_1, \ldots, s_T)$ are computed.
The claimed communication follows directly from inspection. ■

Observe that ReconsPubl provides public reconstruction for $n - t > \frac{2n}{3}$ elements, at a cost of $2n^2$. Thus, the amortized cost of public reconstruction per element is $2n^2/\frac{2n}{3} = 3n$.

3.4 Broadcast

In the sequel, we construct a non-robust broadcast protocol with linear communication complexity. This protocol is a variation of the basic broadcast protocol of [5]. Note that the trivial approach to non-robust broadcast (distribute value, one round of echoing) requires quadratic communication per element, which is too much.

We construct a broadcast protocol that allows one (or several) senders to broadcast $T = n - t$ field elements overall. First, these elements are sent to every party. Then the parties need to verify that they all hold the same values. Instead of echoing all elements, the parties use a hyper-invertible matrix and then each element is echoed to only one recipient, who checks consistency. The protocol has quadratic communication complexity for broadcasting $n - t$ elements, and so is linear per element.

Protocol $\mathsf{Broadcast}(P_s, \vec{x})$**.**

1. P_s sends $\vec{x}$ (of length $T = n - t$) to every $P_j \in \mathcal{P}$.[5] Denote the received vector as $\vec{x}^{(j)}$ (P_j-th view on the vector).
2. Every $P_j \in \mathcal{P}$ applies the hyper-invertible matrix M:[6]
$$\left(y_1^{(j)}, \dots, y_n^{(j)}\right)^T = M \cdot \left(x_1^{(j)}, \dots, x_T^{(j)}, \underbrace{0, \dots, 0}_{t \text{ zeros}}\right)^T$$
3. Every $P_j \in \mathcal{P}$ sends $y_k^{(j)}$ to every $P_k \in \mathcal{P}$.
4. Every $P_k \in \mathcal{P}$ checks whether all received values $\{(y_k^{(j)}\}_j$ are equal; if yes sends a 1-bit and if not a 0-bit, to all other parties.
5. Every $P_k \in \mathcal{P}$ checks that he received a 1-bit from every other party, and aborts otherwise.

Lemma 3.4 *The protocol* $\mathsf{Broadcast}$ *achieves non-robust broadcast, i.e., every (honest) party who does not abort outputs the vector* $\vec{x} = (x_1, \dots, x_{n-t})$*.* $\mathsf{Broadcast}$ *communicates exactly* $2n^2 - nt > 2n^2 - \frac{n^2}{3}$ *field elements and* n^2 *bits.*

Proof: An honest party does not abort only if in Step 5, it received a 1-bit from every other party. Hence, in Step 4, every honest party $P_k \in \mathcal{P}$ has received n identical versions of $\{(y_k^{(j)}\}_j$. This means that all honest parties holds the same value for each y_k where P_k is honest; there are at least $n - t$ such values y_k.

[5] Actually, not all values x_i must be sent by the same sender P_s. It is sufficient that for each x_i there is a sender P_i.

[6] Instead of appending t zeros to the x-vector, one can use an n-by-$(n-t)$ hyper-invertible matrix (which can be derived from a n-by-n hyper-invertible matrix by removing t rows).

Furthermore, for every honest party $P_j \in \mathcal{P}$, the equation

$$\left(y_1^{(j)}, \ldots, y_n^{(j)}\right) = M \cdot \left(x_1^{(j)}, \ldots, x_{n-t}^{(j)}, \underbrace{0, \ldots, 0}_{t \text{ zeros}}\right)^T$$

holds. As M is hyper-invertible, from any n input/output values, the other n input/output values can be computed. As seen above, all honest parties have the same value for at least $n - t$ of the y_k, and furthermore have the same values for the t zeros. Hence, all honest parties have all values identical, in particular the values in $\vec{x}$.

Regarding communication, the first step of Broadcast communicates $n \cdot T = n(n-t) = n^2 - nt > n^2 - \frac{n^2}{3}$ field elements. An additional n^2 elements are communicated in the 3rd step, and n^2 bits in the 4th step. From here on, we ignore the n^2 bits since they are insignificant next to the field elements. ■

Observe that Broadcast broadcasts $n - t > \frac{2n}{3}$ elements, at a cost of $2n^2 - \frac{n^2}{3} = \frac{5n^2}{3}$ field elements. Thus, the amortized cost of broadcast per element is $\frac{5n^2}{3} / \frac{2n}{3} = 2.5n$.

3.5 Preparation (Offline) Phase

The goal of the preparation phase is to generate a number of double-sharings of random values, i.e., each random value is shared once with degree t and once with degree $2t$. One double-sharing is consumed for each input gate, for each random gate, and for each multiplication gate. We generate L random double-sharings, where L is an upper bound on $c_I + c_R + c_M$. We stress that in actuality, a simple sharing suffices for input and random gates, and double sharing isn't needed. Nevertheless, we do it this way for simplicity (and since it makes little difference to the cost in practice in most circuits).

Formally, we say that a value x is (d, d')-*shared*, denoted as $[x]_{d,d'}$, if x is both d-shared and d'-shared via valid Shamir sharings. We observe that double-sharings are linear, i.e., the sum of correct (d, d')-sharings is a correct (d, d')-sharing of the sum.

The following protocol DoubleShareRandom(d, d') non-robustly generates $T = n - 2t$ independent secret random values $r_1, \ldots, r_T$, each independently (d, d')-shared among $\mathcal{P}$. Every (honest) party who completes the protocol holds the corresponding shares.

The protocol for non-robustly generating random double-sharings is taken from [5] as is. For completeness, we repeat the protocol and the lemma, but omit the proof.

The generation of the random double-sharings employs hyper-invertible matrices: First, every party $P_i \in \mathcal{P}$ selects and double-shares a random value s_i. Then, the parties compute double-sharings of the values r_i, defined as $(r_1, \ldots, r_n) = M(s_1, \ldots, s_n)$, where M is a hyper-invertible n-by-n matrix. Then, $2t$ of the resulting double-sharings are reconstructed, each towards a different party, who verify the correctness of the double-sharings. The remaining $n - 2t = T$ double-sharings are outputted. This procedure guarantees that if no honest party aborts, then at least n double-sharings are correct (the $n - t$ double-sharings

inputted by honest parties, as well as the t double-sharings verified by honest parties), and due to the hyper-invertibility of M, *all* $2n$ double-sharings must be correct (the remaining double-sharings can be computed linearly from the good double-sharings). Furthermore, the outputted double-sharings are random and unknown to the adversary, as there is a bijective mapping from any T double-sharings inputted by honest parties to the outputted double-sharings.

Protocol $\mathsf{DoubleShareRandom}(d, d')$**.**

1. Secret Share: Every $P_i \in \mathcal{P}$ chooses a random s_i and acts (twice in parallel) as a dealer in Share to distribute the shares among the parties in $\mathcal{P}$, resulting in $[s_i]_{d,d'}$.
2. Apply M: The parties in $\mathcal{P}$ (locally) compute $\left([r_1]_{d,d'}, \ldots, [r_n]_{d,d'}\right) = M \cdot \left([s_1]_{d,d'}, \ldots, [s_n]_{d,d'}\right)^T$. In order to do so, every P_i computes its double-share of each r_j as linear combination of its double-shares of the s_k-values.
3. Check: For $i = T+1, \ldots, n$ where $T = n - 2t$, every $P_j \in \mathcal{P}$ sends its double-share of $[r_i]_{d,d'}$ to P_i, who checks that *all* n double-shares define a correct double-sharing of some value r_i. More precisely, P_i checks that all d-shares indeed lie on a polynomial $g(\cdot)$ of degree d, and that all d'-shares indeed lie on a polynomial $g'(\cdot)$ of degree d', and that $g(0) = g'(0)$. P_i sends a bit to all P_j signaling whether inconsistencies were observed or not, and every P_j aborts if at least one party reports inconsistencies.
4. Output: The remaining T double-sharings $[r_1]_{d,d'}, \ldots, [r_T]_{d,d'}$ are outputted.

Lemma 3.5 **([5])** *All (honest) parties who complete the protocol* $\mathsf{DoubleShareRandom}(d, d')$ *output* $T = n - 2t$ *correct and random* (d, d')*-sharings, unknown to the adversary.* $\mathsf{DoubleShareRandom}$ *communicates exactly* $2n^2 + 4nt > 2n^2 + \frac{4n^2}{3}$ *field elements to generate* $n - 2t$ *random double-sharings.*

The proof can be found in [5]. Regarding the communication cost, observe that the secret sharing step costs exactly $2n^2$ since double sharings are sent. Then, in the check phase, double sharings are sent for $n - T = n - (n - 2t) = 2$ values, resulting in an additional $2n \cdot 2t > \frac{4n^2}{3}$ field elements. This results in $n - 2t = \frac{n}{3}$ double sharings, and thus the amortized cost is $\left(2n^2 + \frac{4n^2}{3}\right) / \frac{n}{3} = 10n$ elements per double sharing generated.

In the preparation phase, the parties need to generate L random $(t, 2t)$-sharings. This is achieved by invoking the protocol $\mathsf{DoubleShareRandom}$ $\lceil \frac{L}{n-2t} \rceil$ times (in parallel).

Regular sharings. We stress that for input and random gates, it suffices to generate regular sharings of random values of degree t only. The cost of generating (non-double) random sharings is exactly half, and works in the same way as $\mathsf{DoubleShareRandom}$ except that only single sharings are sent. Thus, the cost of generating these single sharings is $5n$ per element. Nevertheless, for simplicity (and since it only makes a difference if there is

a large number of input or random gates), we describe the protocol with random double-sharings only, and in this case only the degree-t part is consumed for input and random gates.

3.6 Summary of Costs

Before proceeding to the full protocol, we summarize the communication costs of all subprotocols. Table 1 includes the overall cost for each subprotocol, how many elements that cost counts for, and the amortized cost per element. The amortized cost refers to the cost per element reconstructed or broadcast, or to double sharing generated, depending on the specific protocol. All counts are in field elements.

Subprotocol	**Cost**	**# Generated**	**Amortized**
ReconsPubl	n	1	n
ReconsPriv	$2n^2$	$\frac{2n}{3}$	$3n$
Broadcast	$2n^2 - \frac{n^2}{3}$	$\frac{2n}{3}$	$2.5n$
DoubleShareRandom	$2n^2 + \frac{4n^2}{3}$	$\frac{n}{3}$	$10n$

Table 1: Costs for all subprotocols.

3.7 Computation Phase

In the computation phase, the circuit is evaluated, whereby all intermediate values are t-shared among the parties in $\mathcal{P}$.

Input gates are realized by reconstructing a pre-shared random value r towards the input-providing party, who then broadcasts the difference of this r and its input. We always process $n-t$ input gates at once, as the protocol Broadcast allows to broadcast that many values at once.

Due to the linearity of Shamir sharing, addition gates (and general linear gates) can be computed locally simply by applying the linear function to the shares. That is, for any linear function $\mathcal{L}$, a sharing $[c] = [\mathcal{L}(a, b, \ldots)]$ is computed by letting every party P_i apply $\mathcal{L}$ and its respective shares, i.e., $c_i = \mathcal{L}(a_i, b_i, \ldots)$.

With every random gate, one random sharing $[r]$ (from the preparation phase) is associated and $[r]_t$ is directly used as outcome of the random gate.

With every multiplication gate, one random double-sharing $[r]_{t,2t}$ (from the preparation phase) is associated. The idea is that each party multiplies its share of $[a]_t$ and its share of $[b]_t$, resulting in a degree-$2t$ sharing $[ab]_{2t}$ of the product ab. Then the difference of $[ab]_{2t}$ and $[r]_{2t}$ is reconstructed (using ReconsPubl) and added on top of $[r]_t$, resulting in a degree-t sharing $[ab]_t$ of the product ab. As the protocol ReconsPubl reconstructs $n-t$ sharings at once, we process $n-t$ multiplication gates in parallel.

Output gates involve private reconstruction ReconsPriv.

Protocol ComputationPhase**.**
Evaluate the gates of the circuit as follows:

- Input Gate (party P_I inputs s):
  1. The parties invoke $\mathsf{ReconsPriv}(P_I, t, [r])$ to reconstruct the associated sharing $[r]_t$ towards P_I.
  2. Party P_I computes $d = s - r$ and invokes $\mathsf{Broadcast}(P_I, d)$.
  3. Each $P_i \in \mathcal{P}$ computes its share s_i of s locally as $s_i = r_i + d$.

  As the protocol Broadcast allows to broadcast $n - t$ values at once, we process $n - t$ input gates in parallel. Note that not necessarily all these input gates need to be for the same party P_I, as Broadcast can take values from several parties.
- Addition/Linear Gate: Every $P_i \in \mathcal{P}$ applies the linear function on its respective shares.
- Random Gate: Pick the sharing $[r]_t$ associated with the gate.
- Multiplication Gate: We denote the two sharings to be multiplied as $[a]_t$ and $[b]_t$, respectively, and the associated double-sharing as $[r]_{t,2t}$. The shares of $[r]_t$ are denoted by $(r_1, \ldots, r_n)$, those of $[r]_{2t}$ by $(r'_1, \ldots, r'_n)$.
  1. Each $P_i \in \mathcal{P}$ computes $e_i = a_i \cdot b_i - r'_i$. Observe that $(e_1, \ldots, e_n)$ form a random degree-$2t$ sharing of a random value, independent of a and b (as blinded by r).
  2. The parties invoke ReconsPubl to reconstruct e to all.
  3. Each $P_i \in \mathcal{P}$ computes $c_i = r_i + e$. Observe that $(c_1, \ldots, c_n)$ form a degree-t sharing of $c = ab$.

  As the protocol ReconsPubl reconstructs $n - t$ sharings at once, we process $n - t$ multiplication gates in parallel.
- Output Gate (output $[s]$ to party P_O): Invoke $\mathsf{ReconsPriv}(P_O, t, [s]_t)$.

Theorem 3.6 (Theorem 1.1–restated) *Let f be an n-party functionality, and let C be an arithmetic circuit over field $\mathbb{F}$ that computes f. Then, protocol* HyperMPC *computes f with perfect security and* with abort*, in the presence of an adaptive, malicious adversary corrupting $t < n/3$ parties. The exact communication complexity of* HyperMPC *is*

$$13n \cdot c_M + 13.5n \cdot c_I + 10n \cdot c_R + n \cdot c_O + 2n^2 \cdot D_M + 10n^2$$

field elements (sent by all parties overall), where c_m denotes the number of multiplication gates, c_I the number of input gates, c_R the number of random gates, c_O the number of output gates, and D_M the multiplicative depth of C.

Proof: [Proof sketch] The security of the protocol for input, random, and output gates follows directly from inspection. The security of the protocol for addition / linear gates

follows directly from the linearity of Shamir sharing. The security of the multiplication protocol can be seen as followed: Given that the shares $(a_1, \ldots, a_n)$ and $b_1, \ldots, b_n)$ lie on a degree-t polynomial, then the pairwise products of the shares $(a_1 b_1, \ldots, a_n b_n)$ lie on a degree-$2t$ polynomial. Note that this polynomial is not necessarily random. However, from the preparation phase we have a random degree-$2t$ sharing of a random value r, given by the shares $(r'_1, \ldots, r'_n)$. Hence, the differences $\{a_i b_i - r'_i\}_i$ lie on a uniform random degree-$2t$ polynomial; reconstructing its secret does not reveal any information. The secret of this polynomial can easily be seen to be $ab - r$, hence $[r]_t + (ab - r)$ is a degree-t sharing of the product ab.

The claimed communication complexity follows from the communication complexities of the involved sub-protocols. Recall that each double-sharing costs $10n$, and one is used for each input, random and multiplication gate. Now, input requires a single double sharing, one ReconsPriv invocation (n), and one Broadcast invocation $(2.5n)$, resulting in $13.5n \cdot c_I$. Random gates have no additional cost beyond a single double sharing, and we therefore have $10n \cdot c_R$. Multiplication uses a single double sharing $(10n)$ and a single ReconsPubl invocation $(3n)$, at the cost of $13n \cdot c_M$. Finally, output is just a single ReconsPriv, adding $n \cdot c_O$.

We now explain the $2n^2 \cdot D_M + 10n^2$ terms. Recall that ReconsPubl in the multiplication gate computation processes $n - t$ values at once, but we can only process gates in parallel when the gates are independent from each other (at the same level in the circuit). Thus, on each (multiplicative) level of the circuit, one instance of ReconsPubl may process less than $\frac{2n}{3}$ gates. In the worst case, only one gate is processed, at the cost of $2n^2$. This explains the additional $2n^2 \cdot D_M$ term in the communication. Likewise, double-random sharings are generated at a cost of $10n^2$ for $n/3$ elements; if there are very few needed, then $10n^2$ field elements still need to be computed. We stress, however, that both these terms will be insignificant, except for extremely small or narrow circuits. This completes the proof sketch. ■

3.8 A Note on Fairness

HyperMPC as presented does not guarantee fairness. In order to see why, consider the case that only one honest party participates in the output phase (since the other honest parties have aborted previously). Then, in the output phase, the corrupted parties can simply not send their shares, but they will obtain the honest party's share. In this case, the corrupted parties will have $t + 1$ shares and can reconstruct the output, while the honest parties cannot.

One may make the protocol fair quite easily. The idea is that before any output gate is processed, the parties check that all (honest) parties are still participating and have not aborted. The output gates are processed only if all parties declare that they are alive and ready to generate output.

In order to check that all parties are alive, one needs asynchronous broadcast (we must

ensure that if some honest party concludes that everybody is alive, then all honest parties will eventually conclude the same way). This can be achieved with Bracha broadcast [7], as described below.

Protocol $\mathsf{CheckAlive}$**.**

1. $\forall P_i \in \mathcal{P}$: send every $P_j \in \mathcal{P}$ the message (alive).
2. $\forall P_j \in \mathcal{P}$: Upon receiving ($\mathsf{alive}$) from P_i, send (echo, i) to every $P_k \in \mathcal{P}$.
3. $\forall P_k \in \mathcal{P}$: Upon receiving (echo, i) from $n - t$ *different* parties *or* receiving (ready, i) from $t + 1$ *different* parties, send (ready, i) to every $P_\ell \in \mathcal{P}$.
4. $\forall P_l \in \mathcal{P}$: Upon receiving (ready, i) from $n-t$ *different* parties, conclude that P_i is alive.

Each party proceeds to the output phase only once he has concluded that all parties in $\mathcal{P}$ are alive. In the output phase, outputs need to be reconstructed *asynchronously*. In particular, the output party P_O must not wait until it has received all n shares of the output, but only until it has received enough shares such that the output value can be safely reconstructed (i.e., such that at least $2t + 1$ of the received shares lie on a degree-t polynomial). This suffices to ensure that the reconstructed output is valid, since at least $t + 1$ of those shares must be correct.

Protocol $\mathsf{ReconsAsync}$**.**

1. $\forall P_i \in \mathcal{P}$: send share s_i of output to the output party P_O.
2. P_O: Upon receiving a new share s_i from P_i, check that among all received shares there are at least $2t + 1$ shares that lie on a degree-t polynomial $p(\cdot)$. If so, output $s = p(0)$. Otherwise, keep on waiting for more shares.

Step 2 can be implemented efficiently with an appropriate error-decoding algorithm (e.g. the algorithm of Berlekamp and Welch). This is needed since at this point honest parties cannot abort (the corrupted parties have already received output). Thus, honest parties must be able to correctly reconstruct, even if they receive incorrect shares from the corrupted parties.

3.9 Comparison to [5]

As we have stated, HyperMPC is a simplification of the protocol of [5], with significant optimizations made possible by the fact that we compromise on security with abort. The main difference between HyperMPC and [5] is that we can skip the generation of Beaver triples. In particular, [5] generates random double-sharings in a preparation phase, and uses those to generate Beaver triples. Then, in the evaluation phase, the Beaver triples are used for multiplying shared values. In contrast, HyperMPC uses the random double-sharings generated in the preparation phase *directly* to multiply shared values in the evaluation

phase. As a result, the dominating costs *per multiplication gate* in the protocols are as follows:

Protocol [5]:

1. *Preparation phase:* 3 random double-sharings and 1 public reconstruction
2. *Evaluation phase:* 2 public reconstructions

HyperMPC:

1. *Preparation phase:* 1 random double-sharing
2. *Evaluation phase:* 1 public reconstruction

This already yields a factor of 4 in the preparation phase and 2 in the evaluation phase (and so a factor 3 overall). However, since we do not require guaranteed output delivery, HyperMPC is even more efficient. In particular, [5] can carry out $n - 2t$ public reconstructions at once, whereas HyperMPC achieves $n - t$ public reconstructions at exactly the same cost. This means that our reconstructions are actually doubly as efficient per reconstructed value (since for $t = n/3$, it holds that $n - t$ is double $n - 2t$). Thus, in actuality, the improvement is is also a factor of 4 in the evaluation phase, making HyperMPC four times faster than [5] overall.

4 Implementation and Experimental Results

We ran experiments to verify the feasibility of running MPC on end-user devices via MPSaaS, and to test the efficiency of the HyperMPC protocol. Our experiments demonstrate that MPC is feasible in these environments, albeit not providing real-time results. We stress that most end-user secure computations that MPSaaS would be used for do not require execution times that are measure in milliseconds. However, very long computations are also problematic, mainly due to the possibility of parties failing midway.

4.1 Endpoint HyperMPC Experiments

We ran experiments with mobile phones, MPC-in-the-browser, and Raspberry Pi3, running the HyperMPC protocol. We ran two sets of experiments; in the first the size of the circuit was varied, and in the second the depth of the circuit was varied. In all of these experiments, we ran with 3 parties (due to the technical difficulties of deploying many endpoints). These experiments will be extended to a large number of endpoints in the full version of this paper. In Figure 10 you can see the running times (in milliseconds) of four different platforms for a circuit of size 10,000 multiplication gates with depths 10, 20 and 30. The Raspberry Pi3 was connected via a LAN and so depth was less significant than for the other platforms.

In our experiments we used 31-bit and 61-bit Mersenne primes to define the prime fields (modular multiplication is extremely efficient modulo Mersenne primes, and these suffice for integer computations).

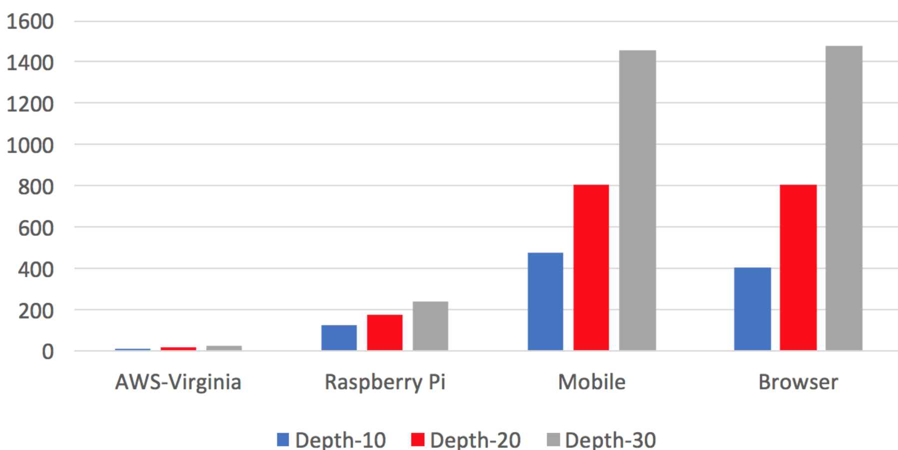

Figure 10: 3 parties running a circuit of size 10,000 mult. gates (over a 61-bit prime), with varying depths and platforms (time in ms).

Figure 11 presents the results of the analogous experiment with varying circuit size (i.e., varying number of multiplication gates), where the parties run three different circuits of sizes 10,000, 50,000 and 100,000. In each case, the depth of the circuit was 20.

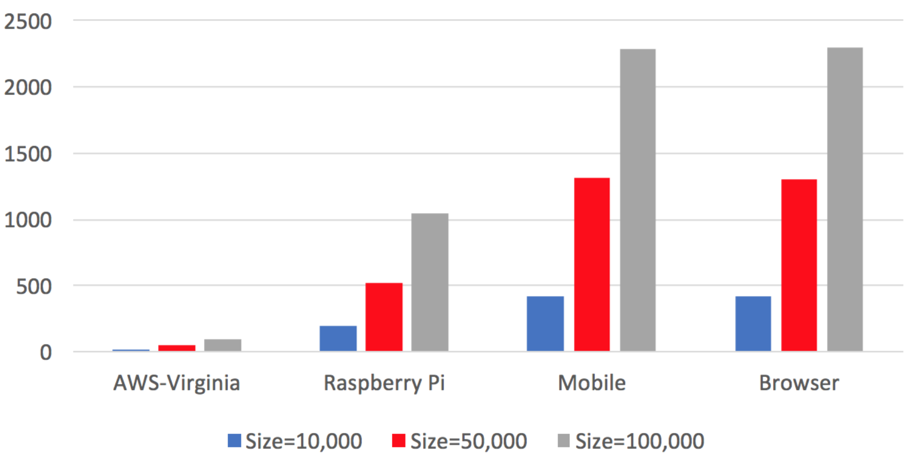

Figure 11: 3 parties running a circuit of depth 20 (over a 61-bit prime), with varying sizes and platforms (time in ms).

4.2 Server HyperMPC Experiments

In addition to the endpoint experiments, we ran larger scale experiments on HyperMPC in order to verify its scalability for a large number of parties and large circuits. In this section, we describe these experiments. In our experiments, all parties are in a single AWS region, running on `m5.12xlarge` instances.

Statistics computation – varying parties. First, we ran an experiment aimed at demonstrating a real-world computation of interest in the multiparty setting. We consider the case of parties holding private data, who wish to compute statistics of their joint data. Specifically, each party holds a series of two variables (e.g., patient weight and blood sugar), and the parties wish to compute the mean and standard deviation of each variable, and to compute a regression test between them. The circuit has 4,000,000 inputs and 16,000,000 gates, of which approximately 10,000,000 are addition and 6,000,000 are multiplication. This specific circuit is only of depth 1, but it models a very real computation of interest that one may wish to compute privately (experiments on circuits of different depths appear above). We ran this experiment with a differing number of parties, from 10 parties up to 150 parties, with increments of 10. The number of inputs is 4,000,000 always, and thus with 10 parties it models a case where each of the 10 parties has an input database of size 400,000, and with 150 parties each of the parties has an input database of size ≈26,000. We stress that the specific circuit we used was not optimized since the specific computation is not of relevance; our aim was to take a circuit of a large size that represents a real computation. Our results unequivocally demonstrate that it is possible to carry out large scale computations using HyperMPC. The results are presented in Table 2 and Figure 12, and clearly show the viability of the computation, and the linear scaling of the protocol.

Parties	Results	Parties	Results	Parties	Results
10	5.01	60	18.42	110	33.84
20	7.42	70	22.57	120	36.23
30	9.73	80	25.13	130	40.20
40	14.08	90	27.35	140	41.97
50	15.99	100	31.34	150	44.18

Table 2: Results of the statistics experiment with a 31-bit prime (sec)

Different field sizes and comparison to [10]. We also ran a circuit of 1,000,000 multiplication gates and depth 20 for different field sizes with HyperMPC; see Table 3 and Figure 13 for the results. As is expected, the running-time of the protocol is *lower* for smaller fields since the communication is lower. This is unlike previous protocols like [21, 10] that require repetition to reduce the statistical error when the field is small. In addition, we compared the running times of HyperMPC to [10] for the same large field Mersenne61.

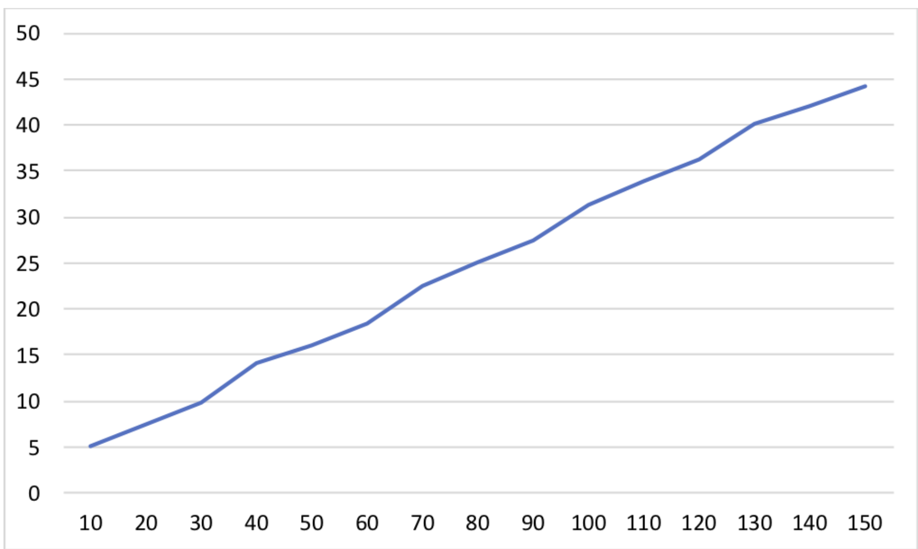

Figure 12: Statistics experiment with 6,000,000 multiplication gates; 10–150 parties

Observe that the running times are almost the same (this can be clearly seen in Figure 13), which is as expected by the theoretical analysis since [10] sends 12 field elements per gate and HyperMPC sends 13 field elements per gate. We stress, however, that HyperMPC would be *much faster* than [10] for smaller fields since [10] would require multiple repetitions of the protocol in order to obtain the required security. For example, [10] would require 5 repetitions of the protocol in the case of $GF[2^8]$ (the small-field version of [10] was never implemented and so we cannot compare to it for actual running times). We stress that HyperMPC requires $t < n/3$ versus [10] that only requires $t < n/2$. Thus, this does not mean that HyperMPC is a "better" protocol, but rather that when one can assume $t < n/3$ then it provides better performance. This makes it an excellent choice for MPSaaS.

Parties	[10] (Mers61)	HyperMPC (Mers61)	HyperMPC (Mers31)	HyperMPC ($GF[2^8]$)
10	906	941	569	283
20	1643	1609	822	606
30	2361	2185	1189	419
40	3049	3165	1853	625
50	3721	3866	2090	677
60	4503	4203	2481	745
70	4965	5341	3057	865
80	6069	6022	3228	954
90	7315	6603	3447	1018
100	8431	7399	4169	1337
110		8259	4915	1289
120		8698	4942	1511
130		10010	5663	2095
140		10371	5959	1986
150		10665	6419	1915

Table 3: Executions in AWS (same region) on `c5.xlarge` machines on a circuit with 1,000,000 multiplication gates and depth 20; time in ms

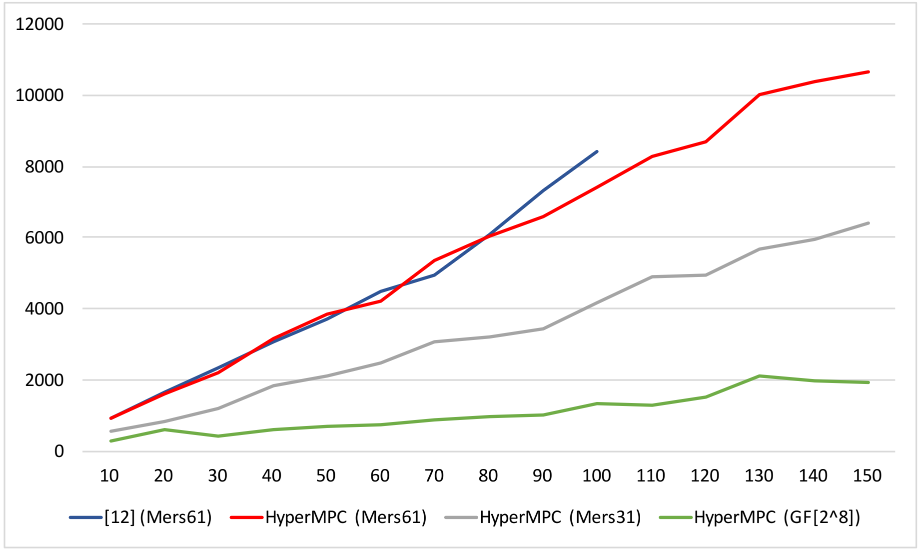

Figure 13: Graph of results in Table 3. Top two lines are [10] and HyperMPC on the same 61-bit Mersenne prime field; next line is HyperMPC with Mersenne-31, and bottom line is HyperMPC with $GF[2^8]$.

Deeper circuits. We ran the same experiment as described in Table 3 for HyperMPC on a circuit with 1,000,000 multiplication gates and depth 100 (instead of depth 20). The results appear in Table 4. Observe that the running times are not much slower than for the depth-20 circuit. This is due to the fact that the majority of the work in HyperMPC is the preparation of double-random sharings and this is computed in parallel for all gates at the beginning of the execution. Then, each multiplication is very cheap, and so when running on a fast network, the difference between depth-20 and depth-100 is only about 10-20% for most cases (indeed, the penalty is higher for $GF[2^8]$; we do not know why there is a difference between this field and the others on this issue). This experiment demonstrates that even quite deep circuits can be effectively run on fast networks.

Parties	HyperMPC (Mers61)	HyperMPC (Mers31)	HyperMPC ($GF[2^8]$)
10	1151	691	355
20	1737	1003	435
30	2647	1391	579
40	3739	2269	948
50	4422	2600	1113
60	5106	3165	1225
70	6002	3826	1782
80	6238	3714	1984
90	6929	4169	1913
100	7874	5554	2666
110	8982	5536	1289
120	9702	7020	2510
130	11259	7379	3020
140	11910	8018	3691
150	12909	8918	5822

Table 4: Executions in AWS (same region) on `c5.xlarge` machines on a circuit with 1,000,000 multiplication gates and **depth 100**; time in ms

HyperMPC with many parties. Finally, we demonstrated that HyperMPC can support MPC with a very large number of parties. We considered a scenario (like that of PrivatePoll) where many parties compute mean and variance over their inputs. We tested this for 250 parties (with a circuit size of 2253 multiplication gates) and 500 parties (with a circuit size of 4503 multiplication gates). We ran this experiment on `c4.large` machines in AWS, in a single region. The execution for 250 parties took 2:30 minutes and the execution of 500 parties took 6:15 minutes. Although not extremely fast, these results do demonstrate that it is possible to run MPC with a very large number of parties. To the best of our knowledge, this is the largest number of parties run in any MPC experiment to date.

Acknowledgements

We thank Zuzana Trubini (Beerliová) and Daniel Tschudi for helpful discussion, and Hila Dahari and Meital Levy for their contribution to the implementation of HyperMPC.

References

[1] T. Araki, J. Furukawa, Y. Lindell, A. Nof and K. Ohara. High-Throughput Semi-Honest Secure Three-Party Computation with an Honest Majority. In the 23*rd ACM CCS*, pages 805–817, 2016.

[2] T. Araki, A. Barak, J. Furukawa, T. Lichter, Y. Lindell, A. Nof, K. Ohara, A. Watzman and O. Weinstein. Optimized Honest-Majority MPC for Malicious Adversaries – Breaking the 1 Billion-Gate Per Second Barrier. In the *38th IEEE Symposium on Security and Privacy*, pages 843–862, 2017.

[3] D. Beaver. Efficient Multiparty Protocols Using Circuit Randomization. In *CRYPTO'91*, Springer (LNCS 576), pages 420–432, 1991.

[4] M. Ben-Or, S. Goldwasser and A. Wigderson. Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation. In the 20*th STOC*, pages 1–10, 1988.

[5] Z. Beerliova-Trubiniova and M. Hirt. Perfectly-Secure MPC With Linear Communication Complexity. In *TCC 2008*, Springer (LNCS 4948), pages 213–230, 2008.

[6] D. Bogdanov, S. Laur and J. Willemson. Sharemind: A Framework for Fast Privacy-Preserving Computations. In the 13*th ESORICS*, Springer (LNCS 5283), pages 192–206, 2008.

[7] G. Bracha. An Asynchronous $\lfloor (n-1)/3 \rfloor$-Resilient Consensus Protocol. In the 3*rd PODC*, pages 154–162, 1984.

[8] R. Canetti. Security and Composition of Multiparty Cryptographic Protocols. In the *Journal of Cryptology*, 13(1):143–202, 2000.

[9] R. Canetti. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In the 42*nd FOCS*, pages 136–145, 2001.

[10] K. Chida, D. Genkin, K. Hamada, D. Ikarashi, R. Kikuchi, Y. Lindell and A. Nof. Fast Large-Scale Honest-Majority MPC for Malicious Adversaries. To appear at *CRYPTO 2018.*

[11] I. Damgård, M. Geisler, M. Krøigaard and J.B. Nielsen. Asynchronous Multiparty Computation: Theory and implementation. In *PKC 2009*, Springer (LNCS 5443), pages 160–179, 2009.

[12] I. Damgård, M. Keller, E. Larraia, V. Pastro, P. Scholl and N.P. Smart. Practical Covertly Secure MPC for Dishonest Majority – Or: Breaking the SPDZ Limits. In the 18*th ESORICS*, Springer (LNCS 8134), pages 1–18, 2013.

[13] I. Damgård and J.B. Nielsen. Scalable and Unconditionally Secure Multiparty Computation. In *CRYPTO 2007*, Springer (LNCS 4622), pages 572–590, 2007.

[14] I. Damgård, V. Pastro, N.P. Smart and S. Zakarias. Multiparty Computation From Somewhat Homomorphic Encryption. In *CRYPTO 2012*, Springer (LNCS 7417), pages 643–662, 2012.

[15] O. Goldreich, S. Micali and A. Wigderson. How to Play Any Mental Game — a Completeness Theorem for Protocols With Honest Majority. In the 19*th STOC*, pages 218–229, 1987.

[16] M. Hirt, U. Maurer and B. Przydatek. Efficient Secure Multi-Party Computation. In *ASIACRYPT 2000*, Springer (LNCS 1976), pages 143–161, 2000.

[17] Y. Huang, P. Chapman and D. Evans. Secure Computation on Mobile Devices. In *IEEE S&P Poster Session*, 2011.

[18] M. Keller, E. Orsini and P. Scholl. MASCOT: Faster Malicious Arithmetic Secure Computation With Oblivious Transfer. In the 23*rd ACM CCS*, pages 830–842, 2016.

[19] E. Kushilevitz, Y. Lindell and T. Rabin. Information-Theoretically Secure Protocols and Security Under Composition. In the 38*th STOC*, pages 109–118, 2006.

[20] A. Lapets, E. Dunton, K. Holzinger, F. Jansen and A. Bestavros. Web-Based Multi-Party Computation With Application to Anonymous Aggregate Compensation Analytics. http://www.bu.edu/today/2016/gender-pay-equity/, 2017. See also http://www.cs.bu.edu/techreports/pdf/2015-009-mpc-compensation.pdf.

[21] Y. Lindell and A. Nof. A Framework for Constructing Fast MPC over Arithmetic Circuits with Malicious Adversaries and an Honest-Majority. In the 24*th ACM CCS*, pages 259–276, 2017.

[22] Y. Lindell and B. Riva. Blazing Fast 2PC in the Offline/online Setting With Security for Malicious Adversaries. In 22*nd ACM CCS*, pages 579–590, 2015.

[23] P. Mohassel, M. Rosulek and Y. Zhang. Fast and Secure Three-Party Computation: The Garbled Circuit Approach. In the 22*nd ACM CCS*, pages 591–602, 2015.

[24] J.B. Nielsen, P.S. Nordholt, C. Orlandi and S.S. Burra. A New Approach to Practical Active-Secure Two-Party Computation. In *CRYPTO 2012*, Springer (LNCS 7417), pages 681–700, 2012.

[25] J.B. Nielsen, T. Schneider and R. Trifiletti. Constant Round Maliciously Secure 2PC With Function-Independent Preprocessing Using LEGO. *IACR Cryptology ePrint Archive*, 2016.

[26] T. Rabin and M. Ben-Or. Verifiable Secret Sharing and Multiparty Protocols With Honest Majority. In the 21*st STOC*, pages 73–85, 1989.

[27] A. Shamir. How to Share a Secret. *Communications of the ACM*, 22:612–613, 1979.

[28] X. Wang, A.J. Malozemoff and J. Katz. Faster Secure Two-Party Computation in the Single-Execution Setting. In *EUROCRYPT 2017*, Springer (LNCS 10210), pages 399–424, 2017.

[29] WebAssembly, `https://developer.mozilla.org/en-US/docs/WebAssembly`.

[30] Web Cryptography API, `https://www.w3.org/TR/WebCryptoAPI/`.

[31] Web Sockets, `https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API`.

[32] Web Workers, `https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers`.

[33] A. Yao. How to Generate and Exchange Secrets. In the 27*th FOCS*, pages 162–167, 1986.

A Challenges for End-User Implementation

We list the challenges we found and solved for running in each platform:

ARM for Raspberry Pi IOT and Mobile: We used a Raspberry Pi3 with a quad-core 1.2Ghz ARM 7a processor, and Android mobiles. These are common and cheap devices used widely. Deploying MPC on this platform involved solving the following issues:

1. *No Intel SIMD:* Existing MPC code heavily uses SIMD instructions from SSE2, SSE4, and AVX2 extensions for fast operations; e.g., fast OT implementations use `_mm_movemask_epi8` and `_mm_slli_epi64` for Bit Matrix Transposition. We used an open source library SSE2NEON to bridge the gap, and extended SSE2NEON to support missing instructions with good performance.

2. *Limited compiler support:* ARM compilers are limited in some of the advanced features of C++ 14, for example with template instantiations. We backported code from C++14 to C+11 to make these libraries work correctly. We also built a cross-compile environment to compile our code simultaneously to all platforms in a Linux build environment.

3. *Limited library support:* Typical MPC code uses gmp, NTL, libmiracl math libraries and in some cased boost for communication. We created valid builds for these libraries. However, not all NTL functionality (used by our code) is available on all platforms (we ported up to NTL 9.7).

We are now working on ports to energy-constraint IOT environments with even less powerful boards.

MPC In-The-Browser: As the browser is becoming the new OS for end users, additional features and capabilities are constantly added, such as access to device (orientation, location), new Web Cryptography API [30], and WebAssembly [29]. These open the possibility of achieving high performance MPC directly in the browser. While writing plain Javascript code is an option, it would require a major rewrite to MPC protcols. Our solution is to cross-compile MPC code to WebAssembly. We have achieved this for the HyperMPC protocol described in Section 3.1.3 and all required libraries. WebAssembly (abbreviated Wasm) is a binary instruction format for a stack-based virtual machine. Wasm is designed as a portable target for compilation of high-level languages like C/C++/Rust, enabling deployment on the web for client and server applications. Wasm is currently supported by all major browsers. The design goal of Wasm is a 2x Slowdown compared to native code. Wasm can interact with additional Javascript libraries, and is executed under the Same-Origin-Policy of the page from which it loaded. We dealt with the following issues:

1. *Code porting:* We ported our C++ code to Wasm using Emscripten 1.37 without introducing design changes to the protocol. Our method is generic and can be applied to any additional MPC protocol. Emscripten is a compiler that runs as a backend to the LLVM compiler and produces a asm.js / WebAssembly. Emscripten has been used to port Python, Lua, libsodium, SQLite, Unreal Engine 3 and several other code bases. To run our MPC code in the browser, we ported the relevant math libraries (NTL, gmp, mpir) and our protocol code. The compiled JavaScript code executes either in the browser, or in a Node.js server. Integration into Node.js is also an important outcome, as MPC can now be delivered through the NPM package manager.

2. *Asynchronous communication:* Emscripten automatically ports C/C++ sockets to the WebSockets protocol [31], and we tweaked this to work for our scenario. However, most open source implementations of MPC use synchronous socket communication, with a thread-per-peer model, and with "send"−"receive" pairs representing round exchanges, with send-receive replaced with receive-send based on a role or id in order to avoid deadlock. This simplistic model is not applicable in a Web Browser, as all socket and I/O

in general is asynchronous. Javascript is asynchronous by-design, and we were unable to use the standard MPC send-receive pattern, even using Web Workers [32]. We therefore had to re-design our protocol code to use a state-machine and asynchronous-IO, with a state-machine per-peer. We use the Emscripten capability to create a Virtual File System in memory to load circuit, input and configuration files in the same manner as used today by our protocol.

3. *SIMD instructions:* WebAssembly does not currently support SIMD instructions. Thus, these instructions need to be replaced.

See Figure 14 for a screen-shot of the protocol of Section 3.1.3 running in a browser.

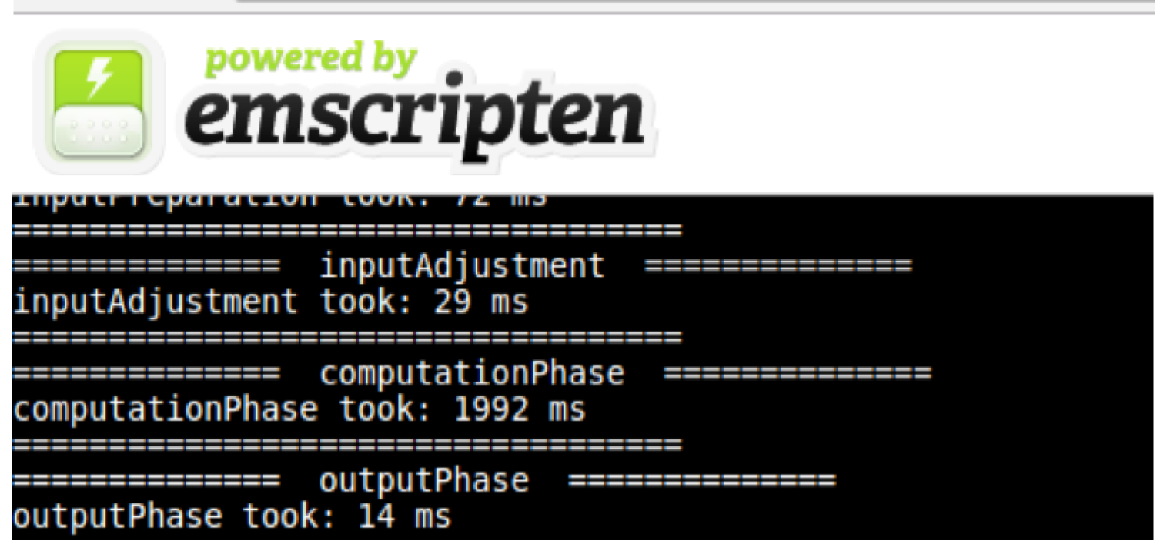

Figure 14: The MPC in the browser view.