Payman Mohassel, Ostap Orobets, and Ben Riva

Efficient Server-Aided 2PC for Mobile Phones

Abstract: Secure Two-Party Computation (2PC) protocols allow two parties to compute a function of their private inputs without revealing any information besides the output of the computation. There exist low cost general-purpose protocols for semi-honest parties that can be efficiently executed even on smartphones. However, for the case of *malicious* parties, current 2PC protocols are significantly less efficient, limiting their use to more resourceful devices. In this work we present an efficient 2PC protocol that is secure against malicious parties and is light enough to be used on mobile phones. The protocol is an adaptation of the protocol of Nielsen et al. (Crypto, 2012) to the *Server-Aided* setting, a natural relaxation of the plain model for secure computation that allows the parties to interact with a server (e.g., a cloud) who is assumed not to collude with any of the parties. Our protocol has two stages: In an offline stage – where no party knows which function is to be computed, nor who else is participating – each party interacts with the server and downloads a file. Later, in the online stage, when two parties decide to execute a 2PC together, they can use the files they have downloaded earlier to execute the computation with cost that is lower than the currently best semi-honest 2PC protocols. We show an implementation of our protocol for Android mobile phones, discuss several optimizations and report on its evaluation for various circuits. For example, the online stage for evaluating a single AES circuit requires only 2.5 seconds and can be further reduced to 1 second (amortized time) with multiple executions.

Keywords: Secure Two-party Computation, Privacy on Mobiles, Server-Aided Secure Computation

DOI 10.1515/popets-2016-0006
Received 2015-08-31; revised 2015-12-02; accepted 2015-12-02.

1 Introduction

Consider a mobile application that can help two users find common contacts, common friends on a social network, the favourite movies they both like, or an available time slot in their calendars for an upcoming meeting, all without revealing more information than needed for the output; or a different application that allows users to determine which services or even which friends are nearby without revealing their location to others. These are just a few simple examples of privacy problems that can be solved using secure multiparty computation (MPC) wherein multiple parties, each with their own private input, want to compute a function of their inputs without revealing extra information. In fact, in the last few years, a surge of new protocols, techniques and implementations for MPC have been presented, with major progress towards practical efficiency.

Increasingly, however, individuals use their smartphones for the bulk of their daily activities and store on them their personal or otherwise sensitive information (e.g. passwords, financial info, emails, etc.). It is crucial for MPC protocols to adapt themselves to this new computing environment, and for new designs to deal with the computation, memory and bandwidth limitations associated with the use of small devices. Protocols with *semi-honest* security (where players are assumed to follow the steps of the protocol) are efficient enough to run on smartphones. For example, the two seminal MPC constructions of Yao's garbled circuit [Yao86] and GMW [GMW87] have been the subject of numerous implementations and optimizations in the last few years [MNP+04, HSS+10, HEKM11, BHKR13, CHK+12, DSZ14].

But a *graceful transition to security against the more realistic malicious adversaries, and major reduction in bandwidth and memory usage* are two of the main challenges facing privacy-preserving computation on smartphones. In particular, secure computation of a boolean circuit using Yao's protocol [Yao86] requires the exchange of tens of bytes per each gate even in the semi-honest case, and to enhance security to the malicious case, the best solutions increase the computation and bandwidth by a multiplicative factor of 40 or more [Lin13] making secure two-party computation impractical for use on smartphones for most functions of interest. We also note that recent work shows that when executing many instances of 2PC it is possible to further reduce the multiplicative factor in the amortized cost [LR14, HKK+14]. We discuss other alternatives in Section 2.

Secure Computation on Mobiles. The research on secure multiparty computation for smartphones is in a very early stage with only a handful of work exploring the topic. Huang et al. [HCE11] studied the feasibility of MPC for mobile devices by implementing a semi-honest private set intersection protocol as an Android App. De Cristofaro et al. [DCFGT12] implemented semi-honest privacy-preserving protocols for a number of genetic testing problems on a smartphone, while Henry

Payman Mohassel: Yahoo Labs, pmohassel@yahoo-inc.com
Ostap Orobets: University of Calgary, oorobets@ucalgary.ca
Ben Riva: Google, benr.mail@gmail.com. Work was done while at Bar-Ilan University.

et al. [CADT14] advocate the use of homomorphic encryption over garbled circuits for secure computation of some functions on a smartphone, and [MLB12] studied efficiency of garbled circuit generation on smartphones. To the best of our knowledge, however, no prior work studies or implements MPC with malicious security for smartphones and existing techniques for transforming the semi-honest variants into maliciously secure ones, are too inefficient to scale.

The Server-Aided Setting. To overcome this problem, we consider a natural relaxation of the standard models for secure computation called the *Server-Aided model*. In particular, we add a third-party, i.e. a semi-honest server who does not collude with the players. This server wishes to learn as much as possible about the players' inputs but has incentive to follow the protocol and not cooperate in cheating. The server-aided model is quite natural given the widespread use of cloud services in today's computing environment, and particularly well-justified for small devices that often benefit from outsourcing to remote servers due to limited resources. Note that there is often little incentive for cloud providers to collude with cheating players whose business is not directly related to theirs, while existence of audits and the fear of legal/finacial repercussion or loss of reputation are additional motivation.

Server-aided MPC with two or more servers has been considered in the work of [DI05, DIK$^+$08] and even deployed in practice [BCD$^+$09]. In case of single-server aided MPC, currently two different approaches are considered: the first is to combine fully-homomorphic encryption (FHE) [G$^+$09] with a proof system [BG02, AJLA$^+$12] but this is only of theoretical interest (though achieving stronger security). The second is using Yao's garbled circuit technique [Yao86]. The latter was proposed by Feige et al. [FKN94] in the semi-honest model, formalized and extended to stronger models by Kamara et al. in [KMR11], and optimized and implemented in [KMR12, CMTB13, CLT14]. But even these protocols are not yet ready for prime time on mobile devices. Huang [Hua12] also explores server-aided 2PC, but based on the GMW construction[GMW87]. We discuss this work in more detail in the related work section.

Our Contributions. We focus on constructing an efficient 2PC protocol in the server-aided setting, one which is secure against malicious players and can still be executed on today's smartphones. Since smartphones usually have a very low bandwidth Internet connection most of the time, we split the protocol into two phases: In the first, each player interacts on its own free time with the server, independently of future

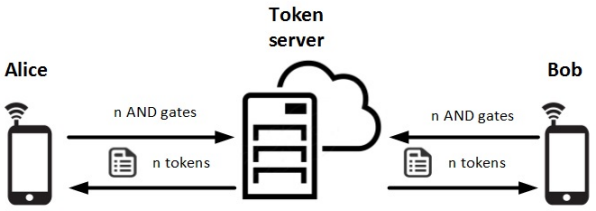

Fig. 1. Offline Stage

invocations of 2PC[1] and receives a file that it stores. (The file size is $\mathcal{O}(\lambda|C|)$ bits where λ is security parameter and $|C|$ is circuit size. Contrast this with cut-and-choose 2PC which requires $\mathcal{O}(\lambda|C|)$ ciphertexts.) We assume that in this phase the players have a high bandwidth connection to the server. This step can be done in bulk for multiple executions and hence only needs to be done occasionally, for example when the user has a high bandwidth Internet connection, or when the phone is connected to a PC.

Later on, any two players who wish to run a secure computation with their inputs need to send/receive a very short message to/from the server, and then interact directly between themselves, exchanging only a few bits per AND gate in the circuit. (In terms of computation, a small number of pseudo-random function calls are made per AND gate.) We stress that the players (nor the server) do not know in advance who would be the other player they would like to execute the secure computation with. The nature of our setting is dynamic, allowing players to choose their counterparts and the function to compute only before starting the actual secure computation in the online stage. Furthermore, the sever does not learn any information about the function being computed except for an upperbound on the number of AND gates. See Figures 1 and 2 for a visual description of the setting.

Our protocol is builds on the protocol of [NNOB12], which also presents a protocol with two stages. However, our protocol simplifies and transfers the heaviest parts of the protocol of [NNOB12] to the server, by utilizing the fact that the server is semi-honest. In addition, we modify the protocol so that the parties can execute the offline stage without knowing in advance which pairs of players would later wish to execute the online stage together.

1 Not entirely independent as players do need to decide on the maximal number of AND gates they will need in the actual computation. However, players can pick several such bounds and choose to download several files in the offline stage, leaving them the option to work later with variety of computation types.

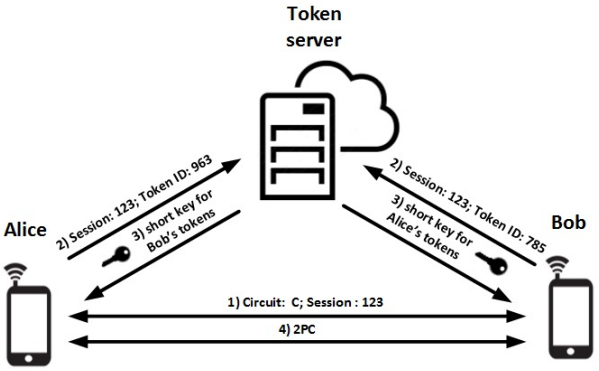

Fig. 2. Online Stage

In case a stronger security guarantee is required, in Section 4.6 we show how to enhance security to handle a *covert server* without any additional cost in the online stage. (Still, requiring that either the server is corrupted or the players, but not both.) In addition, we discuss how to achieve *fairness* in Section 4.7.

We describe a prototype implementation of our protocol and evaluate its efficiency for several circuits. The prototype is implemented in JAVA and works for Android mobile phones. It uses Android's Wi-Fi Peer-to-Peer feature that allows a pair of mobile phones to connect directly to each other via Wi-Fi without additional access point. The prototype itself is optimized based on extensive experiments, both communication-wise and computation-wise. For example, since a mobile phone has only small memory (compared to common desktops), we designed the prototype to be very memory-aware, reusing allocated memory as possible to reduce the overhead of JAVA's garbage collector. Similarly, the prototype works on a layered representation of the circuit being evaluated. As a result, only intermediate wire values are stored in memory, and the communication rounds are fewer.

Organization. Section 2 covers the related work. Section 3 outlines our setting and security definitions. Section 4 presents the setting, our protocol description and a proof sketch. Section 5 describes our implementation, the architecture used, various experimental results, and snapshots of the apps interface.

2 Related Work

Previous General 2PC Protocols. [Yao86] and [GMW87] presented the first 2PC protocols for semi-honest players. The case of malicious players is more complicated and less efficient, and has been the subject of extensive research in the recent few years. Based on ideas from [Yao86], the work of [JS07, NO09] show malicious 2PC protocols that require $\mathcal{O}(1)$ exponentiations per gate of the circuit. However, since exponentiation is rather expensive, these protocol are not efficient enough to be used in practice. [MF06, LP07, LP11, SS11, Lin13, MR13, SS13] use the *cut-and-choose* method which requires $\mathcal{O}(\lambda)$ symmetric encryptions per gate of the circuit (and additional overhead for handling the inputs), where λ is a statistical security parameter such that if a malicious player tries to cheat, it will get caught with probability at least $1 - 2^{-\lambda}$. Despite this linear overhead, several work demonstrate that this approach is relatively practical (e.g., [PSSW09, SS11, KSS12]). [IPS08, IKO$^+$11] suggest a different approach for constructing malicious 2PC protocols that are asymptotically more efficient than the above mentioned protocols, however, their concrete efficiency is unknown since no implementation exits, and it seems that for concrete security parameters the complexity would be worse than that of the above protocols. Last, [NNOB12] presents a very efficient protocol that is based on [GMW87]. We discuss this protocol further below.

2PC protocols with external assistance. Many protocols modify the execution setting by adding some external party who assists with the execution of the protocol. Here we only mention a few recent works which are the most related to ours.

[KMR12, CMTB13, MGBF14] show how to outsource some of the expensive steps of cut-and-choose based 2PC protocols (e.g., oblivious transfers). However, in those works, some of the parties are still required to work hard (e.g., generate $\mathcal{O}(\lambda)$ garbled circuits). [DSZ14] presents a protocol in which the parties have access to small trusted hardware tokens, and assumes that the parties are semi-honest. An outline of how to modify their protocol to work with malicious parties is presented in the full version of [DSZ14] but it is not fully described or implemented. The setting of [DSZ14] and ours are very close, and indeed, the protocols are quite similar. We note that our work was done concurrently to the work of [DSZ14], and for the specific setting we described. [Hua12] shows a protocol in which the parties have access to a trusted third-party who generates correlated randomness for them. This is essentially the same service we need from the server. However, the protocol of [Hua12] requires the server to know in the offline stage who are the pairs of parties that wish to execute the online stage later. In our setting, the offline stage is independent of those pairs, and resembles a more natural "registration" setting. In Section 5 and after describing our experimental results, we provide more concrete comparisons with the above-mentioned work.

Possible Alternatives to Our Protocol. We note that in our protocol, players do not need to know each other or the function they wish to evaluate in the offline stage. Those properties

cannot be directly achieved using protocols that use garbled circuits without sending at least one garbled circuit in the online stage. E.g., one alternative is to ask the server to send a garbled circuit to one of the parties in the online stage and have the second party send the labels for its input. If the communication between the parties and the server in the online stage is slow, this alternative would be impractical. A second alternative would be for the server to send the garbled circuit in the offline stage when the parties have high-speed connection to the server. However, this requires fixing the function to be evaluated in the offline stage. Both solutions, however, let the server learn the function being computed, which we can avoid in our solution.

Last, the alternative of having one of the parties generate the garbled circuit requires a mechanism (such as cut-and-choose) for verifying correctness of the circuits, which leads to generating/checking/ evaluating $\mathcal{O}(\lambda)$ garbled circuit by both parties. In contrast to these options, our protocol requires sending only several bits per AND gate of the circuit in the online stage, and the function to be evaluated is decided only in the online stage.[2]

We base our protocol on the protocol of [NNOB12] that has a highly efficient online stage. However, the offline stage of [NNOB12] cannot be delegated to the server as is without knowing who would be the parties that will execute the online stage later. This is the main modification we make to the protocol of [NNOB12]. In addition, we change the steps needed for evaluating an AND gate so the number of rounds are half of that needed in the protocol of [NNOB12]. Note that our protocol does not require running any oblivious-transfers, or use of *combiners* as needed in [NNOB12], mainly because we work in the server-aided setting.

3 Preliminaries and Security Model

Notations. Throughout this work we use λ to denote a statistical security parameter and κ to denote a computational one.

3.1 MAC with Homomorphism

Similar to [NNOB12], we use a stateful message authentication code (MAC) with a homomorphic property to authenticate message bits.

- $\mathsf{MacGen}(1^\lambda)$: Generates a uniformly random λ-bit string Δ and stores it.
- $\mathsf{MAC}(id, b)$: Picks a uniformly random λ-bit *MAC-base* BASE, stores the record (id, BASE) and returns the tag $\text{TAG} = \text{BASE} \oplus b\Delta$ and id, where $b\Delta = \Delta$ if $b = 1$ and 0 otherwise. (Note that without knowing BASE or Δ, TAG is λ-bit random. id is just an identifier known to all players.)
- $\mathsf{MacAdd}(id_1, id_2, id_3)$: If (id_1, BASE_1) and (id_2, BASE_2) are stored, stores $(id_3, \text{BASE}_1 \oplus \text{BASE}_2)$; Otherwise outputs error.
- $\mathsf{MacVerify}(id, b, \text{TAG})$: If (id, BASE) is stored, verifies that $\text{TAG} = \text{BASE} \oplus b\Delta$ and returns 1; Otherwise outputs error.

In our usage of the MAC, all the algorithms above are performed by the same party who holds the secret key Δ and keeps the "secret" state.

Homomorphic Property. Other participants can only perform addition on tags. In particular, given a tag TAG_1 for bit b_1 with id_1 and a tag TAG_2 for bit b_2 with id_2, one can compute a valid tag for the bit $b_1 \oplus b_2$ by computing $\text{TAG}_1 \oplus \text{TAG}_2$. The *homomorphic property* of the MAC guarantees that after running $\mathsf{MacAdd}(id_1, id_2, id_3)$, $\mathsf{MacVerify}(id_3, b_1 \oplus b_2, \text{TAG}_1 \oplus \text{TAG}_2)$ will return 1.

3.2 The Setting and Adversary Model

The Parties. We have a server S who provides a service for running efficient secure two-party computation on mobiles. Any player can register, even anonymously, to the server, providing it with a bound on the maximal number of AND gates it will need in the online stage, and then receive a commodity file that is stored on the mobile device. Later, any two players who have downloaded such commodity files can execute the secure computation using their mobiles.

We assume that the server is reasonably powerful, e.g., a standard desktop/server PC, while the players are much weaker mobile devices with restricted computing and memory resources.

The communication model. We assume that the players have high-bandwidth connection to the server in the offline stage. (Recall that a player can connect to the server at its convenience as there is no dependency to other players.) Later, in the online stage, the players have a low-bandwidth connection to the server but have a direct connection between themselves. We assume that the communication of the online stage is visible only to the players who run it.

2 Note, however, that our protocol does require fixing some bound on the size of the function in use in the offline stage.

The Adversary model. The adversary can control a subset of the participants causing them to behave maliciously, both in the offline stage and the online stage. (We consider *static* adversaries and not adversaries who choose which players to corrupt during the protocol execution.)

Our definition of security is in the standard ideal/real-world paradigm and follows the definitions typically used for secure MPC.

The main two differences with standard definitions are that, (i) in our setting, we do not allow the adversary to simultaneously corrupt the server S and a (non-server) party. As discussed above, this captures a form of non-collusion between the server and the parties, and (ii) while there is a pool of parties, not all of them have inputs to the function being computed, and it is the adversary in the real execution that decides who the two input-contributors are. This is to capture the fact that the two online participants need not be known in the offline stage when commodity files are downloaded, and the adversary can dynamically make this decision.

In the definition that follows, the corrupted parties could be malicious, covert or semi-honest, but in our constructions, we only consider a semi-honest or a covert server.

Real-model execution. The real-model execution of protocol Π takes place between a subset of parties $(P_1, \ldots, P_n)$, the server S and an adversary $\mathcal{A}$ that is allowed to corrupt an *admissible* subset of the parties. An admissible subset of parties can either be $\{S\}$ or any subset of $\{P_1, \cdots, P_n\}$.

At the beginning of the execution, the adversary $\mathcal{A}$ receives an admissible set I that indicates which parties it corrupts. Parties then interact with each other and the server according to the offline stage of the protocol (no real inputs yet). Denote party P_i's input by x_i, its random coins by r_i and its auxiliary inputs by z_i. The server S only has a set of random coins r_s and an auxiliary input z_s.

Eventually, two parties a and b decide to compute a function of their joint-inputs. Note that only P_a and P_b will have inputs to the function f, and that f will be chosen by them.

For an honest party participating in the online phase, its output is the output of computation and for a corrupted party, its output is chosen by the adversary. Note that the non-participating players do not have an output if they are honest but can have output if they are corrupted. We let honest servers' output be (a, b) capturing the fact that the server learns the two participants in the online stage. We denote P_i's output by OUT_i and Server's output by OUT_s.

The output of the real-model execution of Π between parties $(P_1, \ldots, P_n, S)$ in the presence of an adversary $\mathcal{A}$ is defined as:

$$\text{REAL}_{\Pi,I,\mathcal{A}}(k, F, X, Z, R) \stackrel{def}{=} \left\{\text{OUT}_1, \ldots, \text{OUT}_n, \text{OUT}_s\right\},$$

where $X = (x_1, \ldots, x_n)$, $Z = (z_1, \ldots, z_n, z_s)$, $R = (r_1, \ldots, r_n, r_s)$, and F is the set of functions chosen by the players to evaluate.

Ideal-model execution. The ideal-model execution, the parties have access to a trusted party $\mathcal{F}$. Honest parties interact with $\mathcal{F}$ directly and output the output they receive from it. The messages of the corrupted parties to $\mathcal{F}$ are chosen by the adversary, as well as their outputs at the end of the execution. As before, the simulator receives an admissible set I that indicates which parties it corrupts. It is allowed to either corrupt S or any subset of the players.

The trusted party $\mathcal{F}$ implements the following functionality: Any player P_i can send to the trusted party the message (f, P_i, P_j, x_i, bit) where bit is a bit. (This bit determines whose input is the first.) Once the trusted party has two messages $(f, P_a, P_b, x_a, 0)$ and $(f, P_b, P_a, x_b, 1)$, it sends (a, b) to the server and computes $f(x_a, x_b)$, sends it to the adversary $\mathcal{A}$ if one of the players is corrupted, aborts if was requested to (as in standard 2PC security definitions), or else sends $f(x_a, x_b)$ to the honest players; If both players are honest, it simply sends the outputs to both parties. This suffices if we only consider a semi-honest server. For a covert server, we need a small modification.

Security against *covert* [AL10] adversaries guarantees that a cheating adversary that diverts from the specified protocol is caught (deterred to cheat) by the honest parties with some probability, which is called *deterrence factor*. If the deterrence factor is one, then any cheating adversary is always detected. In order to handle a covert server we need to slightly modify the above ideal execution. For each pair of players who ask to compute f, the server/adversary can ask $\mathcal{F}$ to cheat. Then, the trusted party will flip a bias coin that is head with probability $1/t$ and is tail otherwise. If the coin turns up head it allows the adversary pick a function f' which will be used instead of f. Else, $\mathcal{F}$ sends a *server cheated* message to the two players instead of the output of f.

For all honest parties P_i, let OUT_i denote the output returned to P_i by the trusted party. For all corrupted parties let OUT_i be some value output by $\mathcal{A}$. The output of an ideal-model execution between parties $(P_1, \ldots, P_n, S)$ in the presence of an adversary $\mathbf{Sim}$ is defined as

$$\text{IDEAL}_{\mathcal{F},I,\mathbf{Sim}}(k, F, X, Z, R) \stackrel{def}{=} \left\{\text{OUT}_1, \ldots, \text{OUT}_n, \text{OUT}_s\right\}$$

where $X = (x_1, \ldots, x_n)$, $Z = (z_1, \ldots, z_n, z_s)$, $R = (r_1, \ldots, r_n, r_s)$ and F is the set of functions chosen by the players to evaluate.

We now present our formal definition of security which, intuitively, guarantees that an execution of protocol Π in the real model is indistinguishable from an execution in an ideal model with a trusted party $\mathcal{F}$.

Definition 3.1 (Security)**.** *An n-player sever-aided secure computation protocol Π is secure if for all polynomial-size adversaries $\mathcal{A}$ corrupting an admissible subset of the parties I, there exists a polynomial-size adversary* $\mathbf{Sim}$ *such that*

$$\left\{\text{REAL}_{\Pi,I,\mathcal{A}}(k,F,X,Z,R)\right\}_k \overset{c}{\approx} \left\{\text{IDEAL}_{\mathcal{F},I,\mathbf{Sim}}(k,F,X,Z,R)\right\}_k$$

for all X, Z, and where R is chosen uniformly at random, and for all sets of functions F

4 Our Protocol

Let Alice and Bob be the two players that wish to execute a secure computation, and let S be the service provider. We describe the protocol in three phases, starting with a simple protocol that is only secure against semi-honest player. We then show how to protect against malicious players, and finally present our main protocol removing the assumption that the server knows the two players a priori.

4.1 A Semi-honest Protocol

In the simplest variant of the protocol, we assume that Alice and Bob follow the steps of the protocol and hence behave semi-honestly. We also assume that the server S knows the two players from the very beginning.

The protocol is a natural combination of the GMW protocol [GMW87] and Beaver's multiplication triplets [Bea92]. Lets begin by reviewing the GMW protocol. In this protocol, players jointly evaluate the circuit, gate by gate, where at each step of the evaluation, the players start with secret-shares of the input wires to the gate and end with secret-shares of the output wire of the gate. At the very end, parties reconstruct the values of the output wires in order to learn the output of the circuit.

Let's denote Alice's XOR shares of the input wires to gate g by a_1, a_2 and Bob's shares by b_1, b_2. Our goal is to compute the shares a_3, b_3 such that $a_3 \oplus b_3 = g(a_1 \oplus b_1, a_2 \oplus b_2)$ such that Alice learns a_3 and Bob learns b_3.

First, note that a NOT gate can be easily implemented by flipping the share of one of the players, and thus we focus only on how to compute XOR and AND gates throughout this paper.

In case g is an XOR gate, the computation is rather simple: the players locally compute the values $a_3 = a_1 \oplus a_2$ and $b_3 = b_1 \oplus b_2$. Indeed, $a_3 \oplus b_3 = a_1 \oplus a_2 \oplus b_1 \oplus b_2$ as required. However, for AND gates, the evaluation is more involved. In the original GMW protocol without preprocessing, to evaluate AND gates, players execute a short protocol that uses oblivious transfer.

[Bea92] showed that if in an offline phase, players obtain shares of three bits u, v, w such that $uv = w$ (so that Alice knows the shares u_1, v_1, w_1 and Bob knows the shares u_2, v_2, w_2), in the online phase the evaluation of the gate can be performed without any oblivious transfers as follows: Alice sends $p_1 = a_1 \oplus u_1, q_1 = a_2 \oplus v_1$. Bob sends $p_2 = b_1 \oplus u_2, q_2 = b_2 \oplus v_2$. Then, they both locally compute the values $p = p_1 \oplus p_2 = a_1 \oplus b_1 \oplus u$ and $q = q_1 \oplus q_2 = a_2 \oplus b_2 \oplus v$. Alice sets $a_3 = pq \oplus qu_1 \oplus pv_1 \oplus w_1$ and Bob sets $b_3 = qu_2 \oplus pv_2 \oplus w_2$. Observe that as expected,

$$\begin{aligned} a_3 \oplus b_3 &= pq \oplus qu_1 \oplus pv_1 \oplus w_1 \oplus qu_2 \oplus pv_2 \oplus w_2 \\ &= (a_1 \oplus b_1 \oplus u)(a_2 \oplus b_2 \oplus v) \oplus (a_2 \oplus b_2 \oplus v)u_1 \\ &\quad \oplus (a_1 \oplus b_1 \oplus u)v_1 \oplus w_1 \oplus (a_2 \oplus b_2 \oplus v)u_2 \\ &\quad \oplus (a_1 \oplus b_1 \oplus u)v_2 \oplus w_2 \\ &= (a_1 \oplus b_1 \oplus u)(a_2 \oplus b_2 \oplus v) \oplus (a_2 \oplus b_2 \oplus v)u \\ &\quad \oplus (a_1 \oplus b_1 \oplus u)v \oplus w \\ &= (a_1 \oplus b_1)(a_2 \oplus b_2). \end{aligned}$$

To summarize, the semi-honest protocol works as follows. In the offline phase, the semi-honest server S generates n random Beaver triplet shares for the n non-XOR gates in the circuit. It then sends each parties' shares to them. In the online phase, parties first share their inputs to the circuit and then evaluate the circuit gate-by-gate using the Beaver triplets for evaluating AND gates as described above. At the end of the protocol, parties reveal their shares of the output wires to the circuit for each party to learn its output.

It is easy to show that this protocol is secure as long as S is semi-honest and Alice and Bob are also semi-honest. The protocol is very efficient with only 4 bits communicated per AND gate in the circuit.

4.2 A Malicious Protocol

The protocol described above is secure only against semi-honest players. For example, a corrupted Alice may send an incorrect p_1, q_1, or cheat in its local computation leading to incorrect values for output wires. These behaviors would all go undetected. Forcing the players to behave honestly can be done using *generic zero-knowledge proofs*, however, the efficiency of the resulting protocol in this case would become impractical.

[NNOB12] proposed a different approach for handling malicious players; The main idea is to authenticate all the

bits that are sent during the protocol in a very efficient way, specifically, using a message authentication code (MAC) that supports simple homomorphic operations. In the offline stage, Alice and Bob generate many Beaver triplets (as described above) that are also authenticated using the MAC scheme described earlier. Later, during the online stage, all the bits that are transferred have a tag associated with them that prove they were computed correctly. Since all the operations that the players do for evaluating the gates require only XOR operations, the derived MACs can be computed using homomorphic XOR operations only.

We now describe the resulting protocol in more detail.

Offline Stage. In our setting, the server S performs the offline stage and sends to each party its part of the commodity file. In particular, S generates two MAC keys Δ_a, Δ_b. It sends Δ_a to Bob, and Δ_b to Alice.

For each of Alice's input wires, S generates a random bit u, and a random base BASE_u. It then sends u and $\text{TAG}_u = \text{BASE}_u \oplus u\Delta_u$ to Alice and BASE_u to Bob. Symmetrically, he repeats the same process for Bob's input wires using Δ_b.

Then, for each AND gate it generates the triplet shares $(u_1, v_1, u_2, v_2, w_1, w_2)$ such that $(u_1 \oplus u_2)(v_1 \oplus v_2) = (w_1 \oplus w_2)$. It also generates random MAC bases BASE_{u_i},BASE_{v_i}, BASE_{w_i}. It then sends to Alice u_1, v_1, w_1 and the corresponding MAC tags TAG_{u_1},TAG_{v_1},TAG_{w_1} (where $\text{TAG}_{u_1} = \text{BASE}_{u_1} \oplus u_1\Delta_a$, etc.), and sends to Bob the MAC bases BASE_{u_1},BASE_{v_1}, BASE_{w_1} . Symmetrically, it sends u_2, v_2, w_2 along with the corresponding MAC tags TAG_{u_2}, TAG_{v_2}, TAG_{w_2} to Bob, and the corresponding mac bases to Alice.

Online Stage. The players initialize their inputs as follows. For Alice's input bit a, Alice picks one of the authenticated random bits a' and the corresponding tag $\text{TAG}_{a'}$ she received from the server. She sends to Bob the value $b = a \oplus a'$ and sets $\text{TAG}_a = \text{TAG}_{a'}$. Bob who holds $\text{BASE}_{a'}$ computes $\text{BASE}_a = \text{BASE}_{a'} \oplus b\Delta_a$. Now, Alice's authenticated share of her input bit is a, TAG_a while Bob holds the corresponding base BASE_a. Bob's authenticated share will simply be the bit 0 and the tag 0^λ while Alice sets the corresponding base to be Δ_b. It is easy to check that both shares are correctly authenticated. Parties follow these steps for all of Alice's input bits, and similarly for Bob's input bits with the roles switched.

We proceed to describing how to evaluate a gate given shares of its inputs. Let Alice hold XOR shares of the input wires to gate g, i.e., a_1, a_2 and the corresponding tags $\text{TAG}_{a_1}, \text{TAG}_{a_2}$, and let Bob hold the bases for those i.e. BASE_{a_1}, BASE_{a_2}. Similarly for Bob's input shares he holds b_1, b_2, $\text{TAG}_{b_1}, \text{TAG}_{b_2}$ while Alice holds BASE_{b_1}, BASE_{b_2}. *The invariant we want to keep after evaluating each gate is that Alice holds her share of the output wire a_3, the tag TAG_{a_3} and the base for Bob's tag BASE_{b_3}, and similarly for Bob.*

If g is an XOR gate, Alice simply computes $a_3 = a_1 \oplus a_2$ and lets $\text{TAG}_{a_3} = \text{TAG}_{a_1} \oplus \text{TAG}_{a_2}$. Bob runs MacAdd to compute BASE_{a_3}. A similar process is performed for b_3 with the roles switched.

If g is an AND gate, Alice and Bob will use the authenticated triplets. In particular, Alice sends $p_1 = a_1 \oplus u_1, q_1 = a_2 \oplus v_1$ to Bob, and the corresponding tags $\text{TAG}_{p_1} = \text{TAG}_{a_1} \oplus \text{TAG}_{u_1}$, $\text{TAG}_{q_1} = \text{TAG}_{a_2} \oplus \text{TAG}_{v_1}$. Bob runs MacAdd to calculate BASE_{p_1}, BASE_{q_1}, and uses them to check that TAG_{p_1} and TAG_{q_1} are valid tags of p_1 and q_1. Parties repeat similar steps where Bob sends $p_2 = b_1 \oplus u_2, q_2 = b_2 \oplus v_2$ along with their tags, and Alice verifies them.

Alice and Bob locally compute the values $p = p_1 \oplus p_2 = a_1 \oplus b_1 \oplus u$ and $q = q_1 \oplus q_2 = a_2 \oplus b_2 \oplus v$. Alice sets $a_3 = pq \oplus qu_1 \oplus pv_1 \oplus w_1$ and Bob sets $b_3 = qu_2 \oplus pv_2 \oplus w_2$. Alice lets $\text{TAG}_{a_3} = q\text{TAG}_{u_1} \oplus p\text{TAG}_{v_1} \oplus \text{TAG}_{w_1}$ and Bob lets $\text{BASE}_{a_3} = pq\Delta_a \oplus q\text{BASE}_{u_1} \oplus p\text{BASE}_{v_1} \oplus \text{BASE}_{w_1}$. (Note that here we let Alice's tag for pq be 0^n and let Bob compute the corresponding base $pq\Delta_a$). Similar steps are repeated for Bob's share b_3.

Note that the naive way of checking correctness of p_i, q_i bits that parties send/receive is to do it on-the-fly by asking them to send the corresponding tags and have the recipient verify the tags. But this would require $O(k)$ bits of communication per AND gate. So instead, each party computes the tags itself and adds it to a hash chain it maintains locally. i.e., both Alice and Bob add their version of TAG_{p_1}, TAG_{q_1}, TAG_{p_2}, TAG_{q_2} they compute to their local hash chain and parties test equality of their final hash chain at the end of the protocol, hence only exchanging $O(k)$ bits for the whole circuit.

For example, Bob can compute TAG_{p_1} on his own by letting $\text{TAG}_{p_1} = \text{BASE}_{p_1} \oplus p_1\Delta_a$ and add it to his version of the hash chain. If the value of TAG_{p_1} Alice computed above is not the same, the parties' hash chains would be different and final equality check would detect this.

This protocol is secure agaisnt malicious players (we do not prove this as we will prove security for our main protocol), and it only requires communicating 4 bits for each AND gate (similar to the semi-honest case).

But note that for the above protocol to work, the server must know in the offline phase who the two online players are, in order to generate the correlated triplets consistently. An important goal of our main construction is to remove this requirement.

4.3 The Main Construction

In the previous protocol, S sent to each party its part of the authenticated triplets. But in our desired setting, when S is generating the commodity file for Alice, Bob is not yet in the pic-

ture. A naive way of addressing this issue is to have S generate the authenticated triplet shares as before, send Alice's portion to her, and store Bob's portion on the server side. In the online phase, when Bob is identified, S can send his portion of the commodity file to him. This approach's main drawbacks is that it require $O(k)$ communications per gate between the server and Bob in the online phase, making the online phase expensive. It also requires the server to allocated additional storage for Bob's portion of the commodity until it is used in the future.

Decoupling the offline stage. Our main idea is to ensure that Bob's portion of the commodities can all be generated using a single random PRF key K_a. If so, to send Bob's commodity file to him, we only need to send him the key K_a. To achieve this, we change how the authenticated bit triplets are generated. Lets assume that each triplet has a unique public identifier id where the idea is to generate Bob's portion using $PRF_{K_a}(id \circ c)$ where c is an increasing counter and $\circ$ is the string concatenation operator. (We require $id \circ c$ to be unique and so we realize this by having id be a unique string of a fixed length.)

Initially, Alice requests a commodity file from S. At this stage, the online Bob is not determined. S generates two MAC keys Δ_a, Δ_b, and a PRF key K_a. S then generates each triplet share as follows:

- The server sets $u_2|U_2 \leftarrow \text{PRF}_{K_a}(id \circ 1)$, $v_2|V_2 \leftarrow \text{PRF}_{K_a}(id \circ 2)$, and $w_2|W_2 \leftarrow \text{PRF}_{K_a}(id \circ 3)$. The bits u_2, v_2, w_2 will be Bob's shares in the triplet and the λ-bit strings U_2, V_2, W_2 are the tags associated with those bits. This allows one to generate all of Bob's triplet shares and the associated tags given K_a.
- It then picks Alice's shares u_1, v_1 at random and sets $w_1 = (u_1 \oplus u_2)(v_1 \oplus v_2) \oplus w_2$.
- It sets the MAC-bases for Alice's shares to be $\text{BASE}_{u_1} = \text{PRF}_{K_a}(id \circ 4)$, $\text{BASE}_{v_1} = \text{PRF}_{K_a}(id \circ 5)$, $\text{BASE}_{w_1} = \text{PRF}_{K_a}(id \circ 6)$, and computes the corresponding tags using these bases, e.g., $\text{TAG}_{u_1} = \text{BASE}_{u_1} \oplus u_1\Delta_a$.
- It sets the MAC-bases for u_2, v_2 and w_2 to be $\text{BASE}_{u_2} = U_2 \oplus u_2\Delta_b$, $\text{BASE}_{v_2} = V_2 \oplus v_2\Delta_b$, and $\text{BASE}_{w_2} = W_2 \oplus w_2\Delta_b$. In other words, the bases are chosen such that U_2, V_2, W_2 are the correct tag values for u_2, v_2, w_2, respectively.

Now, the server sends Δ_b, all of Alice's authenticated shares, and all the MAC-bases for Bob's shares to Alice. (That constitutes Alice's commodity files.) Later, when Alice and Bob meet and want to evaluate the circuit, Bob contacts the server and asks for K_a and Δ_a. It uses K_a to regenerate its authenticated shares and all the MAC-bases of Alice's shares. Then, they evaluate the circuit as described in Section 4.2.

So far we have decoupled Bob from the processing stage, and only require Alice to participate in the offline/preprocessing stage. In fact, neither the server nor Alice need to know who is Bob in the preprocessing stage. Also observe that the online stage is quite efficient, as it only requires exchanging 4 bits and computing a small number of PRF outputs by Bob, per AND gate in the circuit.

Making the protocol symmetric. The only drawback of the above solution is that it is not symmetric with respect to the two players (Alice and Bob). Bob's online computation is higher while Alice's storage requirements are more since she needs to store her commodity file. Our last modification is to allow Alice and Bob to both participate in the preprocessing stage, each downloading a file that corresponds to half of the circuit i.e. $n/2$ AND gates, and later run the online stage with those files, allowing them to evaluate n AND gates. The challenge is that the two commodity files will be generated using different and unrelated keys (since it is not known who would be the online 2PC participant), and hence we need an efficient way of making the keys consistent in the online stage.

The idea is as follows. When the server generates Alice's commodity file, it uses two random MAC keys Δ_a and Δ'_a (Δ'_a is used in place of Δ_b above). It does the same with Bob's commodity file, using two random keys Δ_b and Δ'_b (Δ'_b replacing Δ_a in commodity generation). The rest of the commodity file generation process remains the same.

In the online stage, when the server knows who the players are, it sends to Alice the value $\Delta_a \oplus \Delta'_b$ and to Bob the value $\Delta_b \oplus \Delta'_a$. In addition, it sends the values K_a to Bob and K_b to Alice. Note that now, Bob can use $\Delta_b \oplus \Delta'_a$ to adjust its MAC tags so they use the key Δ_b. It can do the same for shares/tags derived from K_a, e.g., it sets TAG_{u_2} to be $U_2 \oplus u_2(\Delta_b \oplus \Delta'_a)$. At the end of this process, all the authenticated bits the players have are authenticated using Δ_a and Δ_b, no matter which commodity file they were associated with.

This concludes description of the main ideas behind the protocol. A complete description follows.

4.4 Detailed Protocol Description

The Offline Stage: Generating Commodity Files

Let n be the maximal number of AND gates in the circuit to be computed, l be the maximal input length, and let PRF be a pseudorandom function, represnted as a random oracle, and let H be a collision-resistant hash function represented as a random oracle. We explain the commodity file generation process for Alice, but an identical algorithm can be used to generate the commodity for Bob as well. In particular, Alice and Bob

can alternate roles from one AND gate to another, hence only receiving commodities corresponding to $n/2$ AND gates. We stress that at this stage the server does not know that Alice and Bob want to talk to each other and hence their commodity files are not correlated in any way.

1. Server S generates a random identification number ID_a identifying the file and writes ID_a to the file. This number should be random and unpredictable so it can be used as a form of authentication. Conveniently, we assume that the identification numbers are random strings of λ-bit length. In practice, they can be derived from $\text{PRF}_{mk}(i)$, where mk is a private master-key of length κ that belongs to the sever and i is an increasing index.
2. S picks three keys $\Delta_a, \Delta'_a \in \{0,1\}^\lambda$ and $K_a \in \{0,1\}^\kappa$. Δ'_a is used as the "offline" key for MACing the shares for Alice's online counterpart; Δ_a is the key for MACing Alice's shares herself, and K_a is used as a key to PRF and for example will be used to generate new MAC-bases for Alice's tags. I.e., $\mathsf{MAC}(id, b) = \text{PRF}_{K_a}(id) \oplus b\Delta_a$.
3. For $i = 1$ to l:
   - **Authenticated random bit (ARB):** S picks a random bit r and writes to Alice's file r and the tag for r i.e. $\text{TAG}_r = \mathsf{MAC}(i \circ 0, r)$, where $\circ$ is a concatenation operator that treats the first input as a $\log(l+1+2n)$-bit string (padded with zeros if needed) to prevent collisions (i.e., $i \circ w = i' \circ w'$ such that $i \neq i'$). Note that the use of 0 is arbitrary and any distinct value can be used instead.
4. For $i = l+1$ to $l+1+n/2$:
   - **Authenticated shared-AND triplet (ASAT):** S generates six random bits $u_1, v_1, u_2, v_2, w_1, w_2$ satisfying the equation $(u_1 \oplus u_2)(v_1 \oplus v_2) = w_1 \oplus w_2$ in the following way: It sets $u_2|U_2 \leftarrow \text{PRF}_{K_a}(i \circ 1)$, $v_2|V_2 \leftarrow \text{PRF}_{K_a}(i \circ 2)$, and $w_2|W_2 \leftarrow \text{PRF}_{K_a}(i \circ 3)$ where u_i, v_i, w_i are bits and U_i, V_i, W_i are strings of λ bits. It then picks u_1, v_1 at random and sets $w_1 = (u_1 \oplus u_2)(v_1 \oplus v_2) \oplus w_2$. S writes to the file u_1, v_1, w_1 and their tags $\text{TAG}_{u_1} = \mathsf{MAC}(i \circ 4, u_1)$, $\text{TAG}_{v_1} = \mathsf{MAC}(i \circ 5, v_1)$, $\text{TAG}_{w_1} = \mathsf{MAC}(i \circ 6, w_1)$.
     It also writes MAC-bases for u_2, v_2 and w_2 to Alice's file, i.e. $\text{BASE}_{u_2} = U_2 \oplus u_2\Delta'_a$, $\text{BASE}_{v_2} = V_2 \oplus v_2\Delta'_a$, and $\text{BASE}_{w_2} = W_2 \oplus w_2\Delta'_a$. (Alice will use these base values and Δ_b to check the other player's MACs in the online phase. Note that neither Alice nor the server know she is talking to Bob yet and hence this connection needs to be made in the online stage where the parties adjust things so that Δ_b is used instead of Δ'_a.)
5. S sends the commodity file to Alice and stores the tuple $(ID_a, K_a, \Delta_a, \Delta'_a)$ locally. Note that the server needs to store only these keys, and not the files it generates.

The same process is repeated for Bob where fresh new random keys Δ_b, K_b and Δ'_b are used to generate Bob's commodity file. (Recall that there are actually three options: (1) Only Alice gets a commodity file for $\geq n$ AND gates; (2) Only Bob gets a file; (3) Both players get files for which the sum of AND gates is $\geq n$. We focus on the third option here.)

The Online Stage: Alice and Bob Interacting

Alice has ID_a and its associated commodity file, and Bob has ID_b and its associated commodity file.

1. **Pairing devices for secure computation:** Alice and Bob decide on the circuit C they wish to evaluate and agree on a session identifier ID_s. Alice sends (ID_a, ID_s) to the server while Bob sends (ID_b, ID_s). S verifies that ID_a, and ID_b are stored in its database (and otherwise reports an error to the players and aborts the protocol). S sends K_b, Δ_b, and $\Delta'_b \oplus \Delta_a$ to Alice, and K_a, Δ_a, and $\Delta'_a \oplus \Delta_b$ to Bob. (Note that this can even be done with a single SMS since these messages are only $2\lambda + \kappa$ bits.)
2. **Initializing hash chains:** Both players initialize hash chains for checking all the MAC tags at the end of the protocol. Specifically, both players set $ch_a = ch_b = 0^\kappa$. During the protocol, each player will update *both* ch_a and ch_b, privately, and at the end of the protocol, Alice would send ch_a to prove her messages, and Bob would send ch_b to prove his.
3. **Preparing input wires:** Let a be Alice's input bit for the input wire $i \leq l$. (Same process is done for Bob's inputs.) Alice reads the ith ARB a' and its tag $\text{TAG}_{a'}$ from her commodity file and sends $b = a \oplus a'$ to Bob. She stores the tuple $(i, a, \text{TAG}_{a'})$. Bob stores the pair $(i, \text{PRF}_{K_a}(i \circ 0) \oplus b\Delta_a)$. (As discussed earlier, note that Bob flips the MAC based on the value of b, while Alice now has the tag of a' for authenticating her actual input bit a.) Bob also stores his share of the bit, which is simply 0 and the tag 0^λ. Alice stores Δ_b as the corresponding base for this bit.
4. **Evaluating an XOR gate:** Let's assume Alice's inputs to the gate are $a_1, a_2, \text{TAG}_{a_1}, \text{TAG}_{a_2}, \text{BASE}_{b_1}, \text{BASE}_{b_2}$ while Bob's inputs are $b_1, b_2, \text{TAG}_{b_1}, \text{TAG}_{b_2}, \text{BASE}_{a_1}, \text{BASE}_{a_2}$. The goal is for Alice to learn $c_a, \text{TAG}_{c_a}, \text{BASE}_{c_b}$ and for Bob to learn $c_b, \text{TAG}_{c_b}, \text{BASE}_{c_a}$ where

$$c_a \oplus c_b = (a_1 \oplus a_2 \oplus b_1 \oplus b_2)$$

   Each party XORs his own inputs and the corresponding tags/MAC-bases locally. I.e. Alice learns $c_a = (a_1 \oplus a_2)$, $\text{TAG}_{c_a} = (\text{TAG}_{a_1} \oplus \text{TAG}_{a_2})$ and $\text{BASE}_{c_b} = (\text{BASE}_{b_1} \oplus \text{BASE}_{b_2})$, while Bob does the same using his own inputs.
5. **Evaluating an AND gate:** Lets assume the parties are computing the ith AND gate in the circuit, and we are using Alice's commodities to evaluated this gate.

Set $j = l + 1 + i$. Alice's inputs to the gate are $a_1, a_2, \text{TAG}_{a_1}, \text{TAG}_{a_2}, \text{BASE}_{b_1}, \text{BASE}_{b_2}$ while Bob's inputs are $b_1, b_2, \text{TAG}_{b_1}, \text{TAG}_{b_2}, \text{BASE}_{a_1}, \text{BASE}_{a_2}$. The goal is for Alice to learn $c_a, \text{TAG}_{c_a}, \text{BASE}_{c_b}$ and for Bob to learn $c_b, \text{TAG}_{c_b}, \text{BASE}_{c_a}$ such that

$$c_a \oplus c_b = (a_1 \oplus b_1)(a_2 \oplus b_2)$$

.

(a) Alice reads the ith ASAT u_1, v_1, w_1 and $\text{TAG}_{u_1}, \text{TAG}_{v_1}, \text{TAG}_{w_1}$, and $\text{BASE}_{u_2}, \text{BASE}_{v_2}, \text{BASE}_{w_2}$. Bob does not read any commodities, but computes the following online $u_2|U_2 \leftarrow \text{PRF}_{K_a}(i \circ 1)$, $v_2|V_2 \leftarrow \text{PRF}_{K_a}(i \circ 2)$, and $w_2|W_2 \leftarrow \text{PRF}_{K_a}(i \circ 3)$, and $\text{TAG}_{u_2} = U_2 \oplus u_2(\Delta'_a \oplus \Delta_b)$, $\text{TAG}_{v_2} = V_2 \oplus v_2(\Delta'_a \oplus \Delta_b)$, $\text{TAG}_{w_2} = W_2 \oplus w_2(\Delta'_a \oplus \Delta_b)$.

(b) Alice sends $p_1 = a_1 \oplus u_1$ and $q_1 = a_2 \oplus v_1$ to Bob. She then computes $\text{TAG}_{p_1} = \text{TAG}_{a_1} \oplus \text{TAG}_{u_1}$ and $\text{TAG}_{q_1} = \text{TAG}_{a_2} \oplus \text{TAG}_{v_1}$, and updates her hash chain by computing $ch_a = \text{H}(ch_a|\text{TAG}_{p_1}|\text{TAG}_{q_1})$.

(c) Bob sets $\text{BASE}_{u_1} = \text{PRF}_{K_a}(i \circ 4)$, $\text{BASE}_{v_1} = \text{PRF}_{K_a}(i \circ 5)$ and $\text{BASE}_{w_1} = \text{PRF}_{K_a}(i \circ 6)$. He computes $\text{TAG}_{p_1} = \text{BASE}_{a_1} \oplus \text{BASE}_{u_1} \oplus p_1\Delta_a$ and $\text{TAG}_{q_1} = \text{BASE}_{a_2} \oplus \text{BASE}_{v_1} \oplus q_1\Delta_a$, and updates his hash chain by computing $ch_b = \text{H}(ch_b|\text{TAG}_{p_1}|\text{TAG}_{q_1})$.

(d) Bob sends $p_2 = b_1 \oplus u_2$ and $q_2 = b_2 \oplus v_2$ to Alice. He then computes $\text{TAG}_{p_2} = \text{TAG}_{b_1} \oplus \text{TAG}_{u_2}$ and $\text{TAG}_{q_2} = \text{TAG}_{b_2} \oplus \text{TAG}_{v_2}$, and updates his hash chain by computing $ch_b = \text{H}(ch_b|\text{TAG}_{p_2}|\text{TAG}_{q_2})$.

(e) Alice computes $\text{TAG}_{p_2} = \text{BASE}_{b_1} \oplus \text{BASE}_{u_2} \oplus p_2\Delta_b$ and $\text{TAG}_{q_2} = \text{BASE}_{b_2} \oplus \text{BASE}_{v_2} \oplus q_2\Delta_b$, and updates her hash chain by computing $ch_a = \text{H}(ch_a|\text{TAG}_{p_2}|\text{TAG}_{q_2})$.

(f) Alice and Bob individually compute $p_1 \oplus p_2$, $q_1 \oplus q_2$ and $\text{TAG}_p = \text{TAG}_{p_1} \oplus \text{TAG}_{p_2}$, $\text{TAG}_q = \text{TAG}_{q_1} \oplus \text{TAG}_{q_2}$.

(g) Alice sets $c_a = pq \oplus qu_1 \oplus pv_1 \oplus w_1$, and $\text{TAG}_{c_a} = q\text{TAG}_{u_1} \oplus p\text{TAG}_{v_1} \oplus \text{TAG}_{w_1}$. Bob lets $\text{BASE}_{c_a} = pq\Delta_a \oplus q\text{BASE}_{u_1} \oplus p\text{BASE}_{v_1} \oplus \text{BASE}_{w_1}$.

(h) Bob sets $c_b = qu_2 \oplus pv_2 \oplus w_2$ and $\text{TAG}_{c_b} = q\text{TAG}_{u_2} \oplus p\text{TAG}_{v_2} \oplus \text{TAG}_{w_2}$. Alice sets $\text{BASE}_{c_b} = q\text{BASE}_{u_2} \oplus p\text{BASE}_{v_2} \oplus \text{BASE}_{w_2}$.

6. **Checking MACs:** Alice sends her ch_a (used to verify her messages), and Bob checks that it is equal to the version of ch_a it computed locally. Similarly, Bob sends his ch_b to Alice who compares it against the version of ch_b it computed locally. Parties abort if the check fails.
7. **Revealing output:** Each party reveals its share of the output wires and the corresponding tag. (If there is a problem with the MACs, the other party aborts.)

4.5 Security Analysis

Before starting the proof, we note that while we use $\text{PRF}_k(\cdot)$ using the notation for a pseudorandom function in the protocol, to prove security of our protocol when the adversary dynamically chooses the online participants, we need $\text{PRF}_k(\cdot)$ to behave like a random oracle. If the online participants are fixed by the adversary a priori, then a PRF property would be sufficient. We focus on the proof of security for the stronger adversary model.

Theorem 4.1. *If the pseudo random function* $\text{PRF}_k(\cdot)$ *and the collision-resistant hash function* H *are implemented using a random oracle, then the protocol of Section 4.4 is secure in the presence of a semi-honest server or any subset of malicious players.*

We need to consider two main cases in order to cover all admissible adversaries. The first case is when the adversary corrupts the server (who is semi-honest) while other players are honest, and the second case is when either the adversary has corrupted exactly one of Alice or Bob (who can be malicious) but the other party and the server are honest, or when both Alice and Bob are corrupted by the adversary. Note that the adversary may corrupt a set of players but only needs to decide in the online phase which ones participate in the protocol.

First Case - Adversary Corrupts the Server. Recall that in this case, the server is semi-honest and thus does not deviate from the protocol specifications. For any adversary $\mathcal{A}$ corrupting the server in the real world, the simulator $\mathbf{Sim}$ in the ideal world runs $\mathcal{A}$ internally. $\mathbf{Sim}$ receives commodity files for all players from $\mathcal{A}$ in the offline phase. It also obtains identities of the two participating parties (Alice and Bob) from the trusted party. The only message sent to $\mathcal{A}$ is the IDs for Alice's and Bob's commodity files in the beginning of the online phase. $\mathbf{Sim}$ sends the same IDs he received in the offline phase as the honest parties would do. $\mathbf{Sim}$ then outputs whatever $\mathcal{A}$ outputs. It is trivially the case that $\mathcal{A}$'s view in the real and ideal world are identical. Given that the server is semi-honest, parties learn the correct output due to correctness of the protocol in presence of honest parties. Hence the REAL and IDEAL distributions are identical.

Second case - Adversary Corrupts a Subset of Players. For any adversary $\mathcal{A}$ in the real protocol, the simulator $\mathbf{Sim}$ in the ideal world runs $\mathcal{A}$ internally. In the very beginning, $\mathbf{Sim}$ needs to emulate the server. It generates the commodity files similar to how the honest server would do, except that whenever $\text{PRF}_k(\cdot)$ is used to generate randomness, $\mathbf{Sim}$ uses a uniformly random string instead. In the online phase, the adversary $\mathcal{A}$ announces which one of the parties in its list will partic-

ipate in the online phase. If both Alice and Bob are corrupted, then the rest of the simulation is simple: **Sim** sends the appropriate IDs, Δs for both parties, chooses random K_a, K_b and sends them to $\mathcal{A}$. **Sim** then responds to all queries to the oracles $\text{PRF}_{K_a}(\cdot)$, $\text{PRF}_{K_b}(\cdot)$ such that they are consistent with the commodity files it generated earlier. It outputs whatever $\mathcal{A}$ does.

The more involved case is when only one party is corrupted. Without loss of generality we assume it is Bob. **Sim** sends the appropriate values $ID_b, \Delta_b, \Delta'_b$, generates a random K_b and sends it to $\mathcal{A}$ as well. As before, we assume that $\text{PRF}_{K_b}(\cdot)$ is a random oracle (e.g. $H(K_b, \cdot)$ where H is hash function model as a random oracle). **Sim** responds to all queries to this oracle, keeping them consistent with the commodity files he had generated offline.

During the input-preparation step of the online stage, $\mathcal{A}$ sends a one-time pad encrypted ciphertext for each bit of Bob's input. **Sim**, who knows the encryption pads (since it generated the commodity files), decrypts them to learn Bob's input x_b and sends it to the trusted party to receive the output of the computation $z = f(x_a, x_b)$. **Sim** also emulates honest Alice using a random input x'_a, following the exact steps of the protocol until the very end where the output gates are computed. For the output gates, **Sim** adjusts Alice's shares so the output wires open to the correct value z. To be more precise, the adjustment is $f(x'_a, x_b) \oplus z$. **Sim** also needs to adjust the MAC tags before adding them to the hash chain it is compiling. But it can do so since it knows the MAC keys, and the corresponding bases.

If $\mathcal{A}$ aborts at any stage during the protocol, **Sim** aborts as well outputting whatever $\mathcal{A}$ outputs. Also if the chain ch_b generated by $\mathcal{A}$ is not equal to the one **Sim** generated itself, it will emulate honest Alice aborting and outputs whatever $\mathcal{A}$.

We now show that the ideal execution describe above is indistinguishable from a real execution. We describe the sequence of hybrids needed for the second case where only one corrupted party participates, and simply note that the Hybrid 2 below is identical to the ideal execution for the case where both Alice and Bob are corrupted, hence Hybrids 3, 4 and 5 are not necessary in that case.

Hybrid 0: This world is the real execution were both Alice and Bob use their real inputs, and interact with the real server.

Hybrid 1: We replace the real server with the **Sim** doing exactly what the real server does. The Hybrid 0 and Hybrid 1 are identical.

Hybrid 2: In this hybrid, **Sim** behaves as prescribed in the simulation above in generating the commodity files, i.e. instead of using PRF_k, it generates random strings on its own for computing the commodity files offline. It follows the real server's behaviors otherwise.

Probability of distinguishing Hybrids 1 and 2 is less than advantage of an adversary in breaking the random oracle PRF_k (i.e. distinguishing it from a random function) which is negligible.

In case both players are corrupted, the simulator programs the random oracle so that once they adversary receives the PRF keys, the commodity files will be consistent with the PRF outputs.

Hybrid 3: Similar to Hybrid 2, but output fail if the values added to hash chain are not the same between Alice and Bob but the output chain for both is the same. The probability of distinguishing Hybrids 2 and 3 is less than advantage of an adversary in finding collisions for the hash function H (which is implemented using a random oracle).

Hybrid 4: Similar to Hybrid 3, except that if Bob's hash chain passes the final check but any intermediate bits exchanged or the final output bits are incorrect, the simulator aborts. The probability of distinguishing Hybrids 3 and 4 is less than probability of an adversary breaking the MAC since Bob needs to forge a tag for the incorrect bit in order to pass hash chain check.

Hybrid 5: the same as Hybrid 4, except that instead of real inputs for Alice, we use random inputs for her. This is the ideal execution described above.

Hybrids 4 and 5 are identically distributed, since all bits received by Bob for intermediate gates are encrypted using a one-time pad (due to the uniformly random pads in the commodity files), and their distribution in both hybrids is uniformly random. The bits sent for the output wires, on the other hand, are distributed to yield the correct output z in both hybrids.

4.6 Handling A Covert Server

In some cases the assumption that the server is semi-honest is plausible. However, in case the players wish to get a stronger security guarantee from the server, then they can download and validate several commodity files before using one for the actual evaluation. We show that this approach provides security in the presence of a covert server. (We stress that we still allow the adversary to either corrupt the server or players, but not both).

In more details, say that Alice wishes to verify that her commodity file is correctly constructed with probability $1 - \frac{1}{t}$. She asks the server for t different commodity files, chooses $t - 1$ of them at random and asks the server to open them. Then, she uses the remaining commodity file in the protocol from Section 4.4.

Unfortunately, this does not suffice. First, a corrupted server can provide an invalid PRF key for the commodity file

in the online phase. (The above cut-and-choose only verifies that the commodity files were generated properly in the offline phase, but does not guarantee that Bob obtains the right PRF key in the online phase.) This is resolved by adding to the commodity file a commitment to the PRF keys. (This is also checked if the commodity files is opened.) In the online stage, Alice sends the commitment from her commodity file to Bob, who then asks the server to decommit and prove that it is indeed the PRF key Bob received. The rest of the protocol remain as before and hence is not repeated here.

A second difficulty is regarding proving the protocol using simulation. This is resolved with standard techniques for proving security of cut-and-choose based protocols: In the RO model, Alice commits on the commodity file she wish to use using the RO, so that the simulator can extract it. Then, the server sends the t files, along with commitments on their PRG seeds, again, using the RO to allow extraction. Alice decommits her choice, and the server opens the checked files as before (while Alice makes sure that the committed PRF keys are consistent with the files). These steps allow the simulator to extract Alice's challenge before sending the commodity files, and similarly, allow the simulator to learn how many files are invalid before asking the server to open any of them. (A slightly less efficient construction can be realized in the standard model using an extractable commitment.)

Theorem 4.2. *If the pseudo random function* $\mathrm{PRF}_k(\cdot)$ *and the collision-resistant hash function* H *are implemented using a random oracle, then the protocol outlined above is secure in the presence of a covert server or any subset of malicious players.*

Since we work in the random oracle model, the proof for the case of malicious players is identical to that of previous protocol, and hence is not repeated here (we do not even need to extract the player's challenge as we can change the commodity files by programming the random oracle). Next, we provide the simulation for the case where the server is corrupted but the players are honest.

For any adversary $\mathcal{A}$ corrupting the server in the real world, the simulator $\mathbf{Sim}$ in the ideal world runs $\mathcal{A}$. $\mathbf{Sim}$ receives t commodity files for each player from $\mathcal{A}$ in the offline phase. It extracts $\mathcal{A}$'s inputs to the RO and hence it learns the opening for all t commodity files for all players. We call a commodity file *bad* if the opening of the secrets does not explain the commodity file or the commitments associated with it.

$\mathbf{Sim}$ then obtains identities of the two participating parties (Alice and Bob) from the trusted party. There are several possibilities. (i) Either Alice or Bob receive more than one bad commodity file. In this case, $\mathbf{Sim}$ sends a **cheat** message to the trusted party, and emulates that party catching the server cheat and aborting, and outputs whatever $\mathcal{A}$ does. (ii) neither Alice nor Bob receives any bad files. In this case, the rest of simulation becomes identical to the semi-honest case above so we do not repeat it here. (iii) Either Alice or Bob receives one bad file but neither one receive two or more bad files. In this case, $\mathbf{Sim}$ sends a **corrupted** message to the trusted party. The TTP flips a $1/t$-biased coin. If it turns up tail, it sends **detected** to the simulator. $\mathbf{Sim}$ simulates the honest player catching the server cheating and aborting, and outputs whatever $\mathcal{A}$ does. If it turns up head, TTP sends **undetected** to $\mathbf{Sim}$. $\mathbf{Sim}$ then sends the function that receives the players inputs, runs the online stage with the invalid commodity files and outputs the result. (The commodity files are hardcoded in the function description.) Last, it outputs whatever $\mathcal{A}$ does. The outputs of this simulated execution is identical to the one in the real world, as the output is computed with the same inputs and invalid commodity files.

It is easy to see that in the random oracle model, the REAL and IDEAL executions are identically distributed.

4.7 On Fairness

The protocol from Section 4.4 does not achieve *fairness* since a corrupted player can abort after receiving the honest player's final message (which allows only the corrupted player to learn the output of the computation). As shown in [KMR11], fairness can be achieved in the server-aided setting. The crucial observation in [KMR11] is that if the server is the party who sends the last message to the players (i.e., the message that allows learning the output), then fairness can be achieved.

We can achieve fairness in our protocol in a similar way as well. We briefly review the modifications we need to make to our protocol for this purpose. Instead of sending the shares and MACs of the output wires to each other, the players send them to the server, who checks the MACs and sends the output to both players, and hence fairness is achieved. However, now the server learns the output of the computation, so another modification is needed. Instead of sending the actual output bits, the players agree on a random mask m, XOR the output of the computation with m (which can be done for free using XOR gates) and send the output to the server, which now learns the output XORed by m.

However, in order to verify an output bit of Alice, Bob has two send the two possible MACs to the server, which would reveal the value of her bit given its tag. (Recall that XORing the output with m is done by XORing the MAC bases for Alice's bits with Δ_a. Knowing the bases and the MAC tags would reveal Alice's bits.) Thus, in order to enable the server to verify the MACs of the shares without learning the actual MAC

bases/tags, the players agree on a random key k and send to the server the shares "encrypted" with a deterministic encryption using key k. (E.g., if Alice has an output bit b and its tag TAG_b, she sends b and an encryption of the tag $AES_k(\text{TAG}_b)$, while Bob sends two encryptions, one of the MAC for bit 0 and one for bit 1. The server simply compares the shares without knowing k.) Since the output is XORed with m, these steps are needed only for the output bits of one of the players. The resulting protocol is simple to perform and requires only 3 additional encryptions per output bit.

A disadvantage of the above protocol is that now the communication between the players and the server in the online stage depends on the output length. First, it reveals the output length to the server (though in most case, this is probably not too problematic). Second, in case the output is long, this might be inefficient since we assumed the communication between the players and the server is limited in the online phase (even though we need only one additional round of communication). In theory, this can be solved by using $f'(k_0, k_0, x, 1) = [Enc_{k_0 \oplus k_1}(f(x_0, x_1)), k_0 \oplus k_1]$ such that Alice inputs a secret random key k_0 and her input x_0, Bob inputs a secret random key k_1 and his input x_1, and Enc is a deterministic encryption; The players learn the output of $Enc_{k_0 \oplus k_1}(f(x_0, x_1))$ while the server learns $k_0 \oplus k_1$ and sends it to both players. Note that now the communication between the players and the server is small and depends only on the security parameter. However, this solution is much less efficient since now the players must also evaluate the circuit for Enc. We leave the question of designing a more efficient solution without this drawback for future work.

5 Prototype Implementation and Evaluation

We implemented a prototype of our protocol for Android smartphones. In this section we briefly describe the architecture of the prototype, and then show an extensive experimental study of its running with different circuits.

5.1 Architecture and Optimizations

The Android application was written in JAVA. As a pseudo-random function we use AES-128 in ECB mode (since inputs to the PRF are at most 1 block long), and as the collision-resistant hash function we use SHA-1. These primitives are implemented using the JAVA Crypto and Security libraries. (During the development we tested several other Android crypto libraries, such as SpongyCastle and Conceal. Our tests showed that the JAVA standard library outperforms the other libraries.) The mobile app is single-threaded.

For memory-saving purposes we use a *binary format representation* of tokens and circuit files. We implemented a tool that receives a human readable circuit representation (using the format of [ST15]) and outputs the smaller binary format our mobile app works with.

Our protocol requires transmitting only two bits per AND gate. We use *bit masking* (multiple bits in a byte) for storing those bits and for reducing the network communication size. In addition, in order to reduce the number of communication rounds, the AND gates are processed *layer-by-layer* instead of gate-by-gate. Namely, we store the values to be transmitted for all AND gates in the current layer and then send/receive them all at once. This allows to us to avoid the TCP overheads. Information regarding circuit layers, the number of AND gates for each layer, etc. is obtained form the circuit file. In order to save on heap memory usage and reduce the garbage collector overheads, our app does not read the entire circuit and commodity files at once, but instead, it retrieves only information about the next layer to be processed. While this approach increases the number of I/O operations, the resulting reduction of memory usage significantly improved performance. (Note that during evaluation, the prototype keeps track only of live wires. Values of wires that are not used in the rest of the evaluation are deleted.)

The user-interface of the mobile app is simple to use: The user can see the currently stored commodity files and their properties (e.g., number of AND gates supported). He can download new commodity files from the server and store them locally by specifying the number of AND gates/inputs or choosing a circuit file for which the commodity is to be generated. The app includes several circuits for testing purposes but allows for adding circuits of your own or downloading them from the server. When the user wishes to execute the online stage with another mobile, it chooses the circuit they wish to evaluate, interact with the server to get the required keys, etc., and then execute the online protocol. Figure 3 includes several snapshots of the user interface of the app.

5.2 Evaluation

Setup. For our experiments we used as the server, a machine with an Intel Core-i5 3570 3.4 GHz processor, and 8GB RAM, which was running Windows 7 (x64). One player was a Samsung Galaxy S3 with dual-core Snapdragon S4 1.5Ghz CPU, 1.5Gb RAM memory (OS Android 4.4.2 KitKat), and another player was a Galaxy S2 with dual-core Snapdragon S3 1.5Ghz,

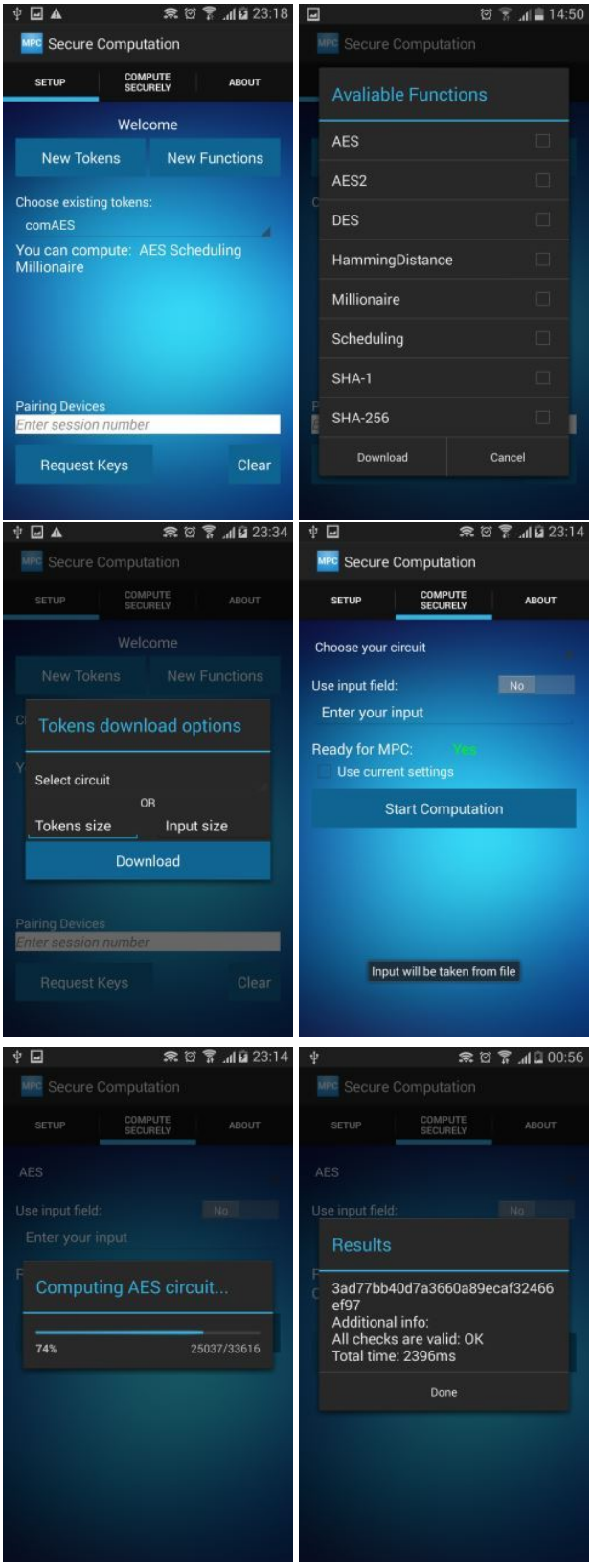

Fig. 3. Application screenshots. The first three screenshots are of the offline stage and the next ones are of the online stage.

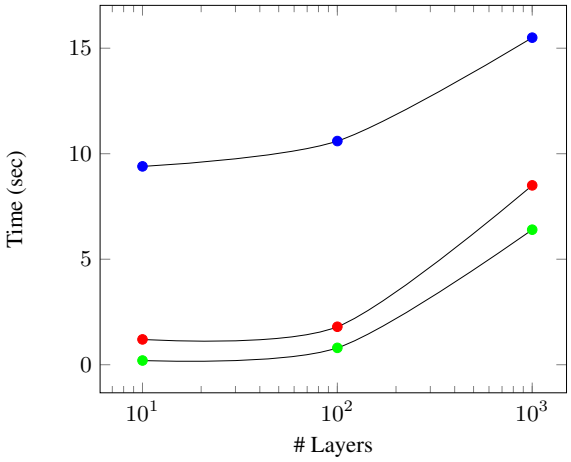

Fig. 4. The effect of the circuit depth on the online running time. The blue dots are for circuit size 100000 gates, the red are for size 10000 gates, and the green are for size 1000 gates.

788Mb RAM (OS Android 4.1.2 JellyBean). Communication with server was done via public Wi-Fi (with TLS as the underlying protocol), while communication between the mobile phones was done through Wi-Fi Direct.

We measured energy consumption during the setup phase using PowerTutor. Our application consumed about 2.6 W*s when downloading 2^{19} tokens (as compared to 1.25 W*s of [DSZ14], though our tokens are larger since we deal also with malicious players).

Synthetic Tests. It is clear that the performance of the online protocol depends on the depth of the circuit being evaluated. Therefore, we generated synthetic circuits consisting of only AND gates, with different depths and sizes.

See Figure 4 for the performance of the online stage for those circuits. Note that the depth has a similar effect on the running time regardless of the total size of the circuit. (The large gap between the blue and the red lines is mostly because of memory issues, as the larger circuit requires much more memory than the smaller one.)

Applications. We tested our prototype with several circuits that compute more useful functionalities. First, we evaluated it with AES, SHA1 and SHA256 which have become standard circuits for evaluating secure two-party protocols. In the SHA-1/SHA-256 circuits, Alice has the first half of the input and Bob has the second half. Second, we generated three circuits using the PCF framework of [KMSB13]: Millionaire, which receives two 32 bit integers and outputs which is larger; Scheduling, which receives two 32 bit integers and outputs their AND (this can be used to implement the scheduling functionality of [DSZ14]); and, Hamming Distance, which receives two 32 bit integers and outputs their hamming distance. (We note that these circuits are not optimized to be shallow,

Circuit	# ANDs	Depth	Offline	Pre-Online	Online
AES	6800	204	450	35	2.5
SHA-1	37300	10445	1400	120	69.5
SHA-256	90825	8071	3600	250	66.8
Millionaire	97	97	250	5	0.59
Scheduling	32	1	240	5	0.07
Hamming-Distance	8194	8194	574	30	47.8

Fig. 5. The running times of the different steps. Offline and Pre-Online are in milliseconds, and Online is in seconds.

thus performance of the online stage is not optimal.) We also tested with synthetic circuits with various depths and sizes to better study the behaviour of our construction as seen in Figure 4.

The results are in Figure 5. We measured three different times: Offline time is the time it takes a player and the server to run the offline stage of the protocol from Section 4. At the end of this stage, the player has the commodity file on its phone. Pre-online time is the time it takes the phone to parse the commodity file (and the circuit specification file) and construct its internal structures. This can be done before the players meet. Last, we measure the time it takes the two players to run the online stage of the protocol from Section 4 (given they both already finished the pre-online stage). Note that in this stage, the communication channel is quite slow and thus the majority of the time is spent in communication.

Better Amortized Efficiency. As discussed earlier, the depth of the circuit heavily influences the running time of the online stage. If the players are interested in evaluating the circuit with different inputs, then the overhead caused by the depth can be amortized so that the cost per each evaluation is much smaller than in the single case. The idea is to run all evaluations in parallel, so that the ith layer of all circuits is evaluated in parallel, resulting in the same number of rounds as in the single evaluation case. (The same idea was used also in [NNOB12].) We note that one can actually use different circuits (and not the same one several times). Currently, our prototype only supports running the same circuit many times in parallel.

In Figures 6 and 7 we show the effect of the amortization technique for the evaluation of AES and SHA-1. We can see that when we use even only two evaluations in parallel, performance is about 40% better. When the number of parallel evaluations is larger, the improvement is even larger. For example, evaluating AES five times in parallel requires 1.01 seconds per evaluation, while evaluating AES in the single evaluation case requires 2.5 seconds. Or, evaluating SHA-1 six times in parallel requires 28.3 seconds per evaluation, while in the single evaluation case it requires 69.5 seconds. (Recall that once we increase the number of gates per layer, our mobile app needs more memory for storing the values of the intermediate wires. This is the reason we see cases where using more evaluations in parallel actually hurts performance.)

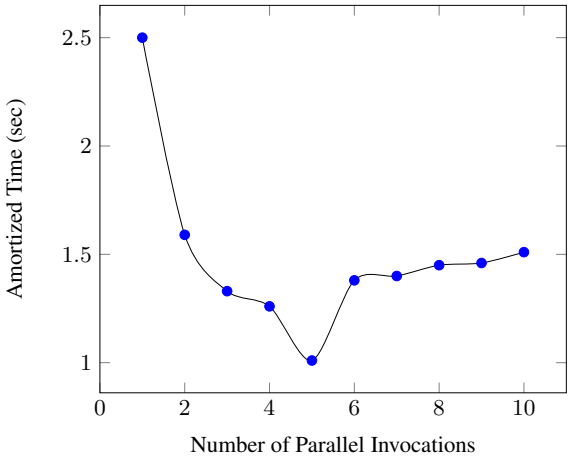

Fig. 6. The amortized (online) time required for computing a single AES circuit.

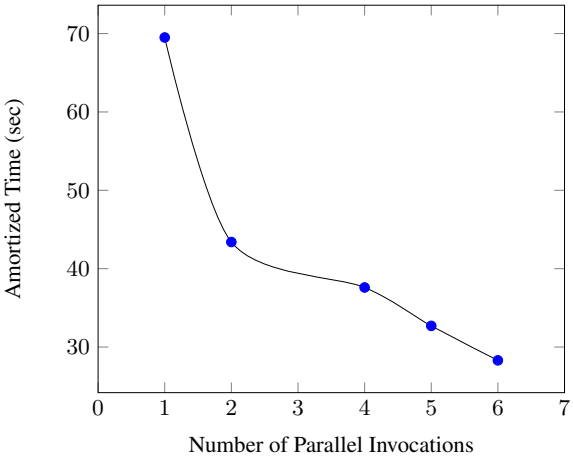

Fig. 7. The amortized (online) time required for computing a single SHA-1 circuit.

Comparison with Related Work. See [DSZ14] for a comprehensive review of related work. Here we repeat only the works that are most relevant to ours.

[HCE11] evaluates semi-honest gc-based secure two-party computation on smartphones. Going from semi-honest 2PC to fully-secure 2PC based on garbled circuits (i.e., using cut-and-choose) is still very expensive and currently seems impractical for running on smartphones. [KMR12] designs and evaluates new protocols in the server-aided setting. They show that AES can be evaluated in about 9 seconds, assuming the server and one of the players can communicate via fast network connection. Note that the goal of [KMR12] is to reduce the work of the players, whereas we would like to reduce their work, but rely on a slow network connection (as is often the

case with smartphones). In addition, in [KMR12] the server knows the circuit being evaluated, whereas in our setting it does not.

[Hua12] presents a protocol that is based on the ideas of [NNOB12] and a trusted server that generates commodity randomness as in our setting. As we follow the same ideas, our protocol is similar to theirs. However, in our setting the players do not have to know in the offline stage, whom would be the other player they will interact with in the online stage. ([Hua12] focuses on standard 2PC setting.) In addition, our prototype is designed for mobile phones, whereas the prototype of [Hua12] runs on desktops. In fact, our prototype achieves similar speed to that of [Hua12] even though it runs on mobile phones. (E.g., evaluating AES takes about 3 seconds for both prototypes.)

Last, [DSZ14] considers a setting in which one of the mobile phones is connected to a trusted smartcard that can generate correlated randomness for the players. As long as the smartcard is not tampered by its user, the protocol of [DSZ14] is secure against semi-honest players. The use of the smartcard allows to significantly reduce the cost of the GMW protocol, similar to our use of the server. It is hard to compare our results to theirs as we use different circuits and different settings. Still, we can provide a rough comparison for our scheduling circuit: The circuit of [DSZ14] has 56 gates whereas ours has 32. [DSZ14] requires about 1.6 seconds (in total) for evaluating those 56 gates, while our prototype requires about 1 second for evaluating 32 gates. Note that our protocol is secure against malicious players, while their implementation only achieves semi-honest security.

Acknowledgement. We would like to thank the Jonathan Katz and the anonymous reviewers for very helpful comments on the paper.

References

[AJLA+12] Gilad Asharov, Abhishek Jain, Adriana Lopez-Alt, Eran Tromer, Vinod Vaikuntanathan, and Daniel Wichs. Multiparty computation with low communication, computation and interaction via threshold fhe. In *Advances in Cryptology–EUROCRYPT 2012*, pages 483–501. Springer, 2012.

[AL10] Yonatan Aumann and Yehuda Lindell. Security against covert adversaries: Efficient protocols for realistic adversaries. *J. Cryptol.*, 23(2):281–343, April 2010.

[BCD+09] Peter Bogetoft, Dan Lund Christensen, Ivan Damgard, Martin Geisler, Thomas Jakobsen, Mikkel Kroigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, et al. Secure multiparty computation goes live. In *Financial Cryptography and Data Security*, pages 325–343. Springer, 2009.

[Bea92] Donald Beaver. Efficient multiparty protocols using circuit randomization. In *Proceedings of the 11th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '91, pages 420–432, London, UK, UK, 1992. Springer-Verlag.

[BG02] Boaz Barak and Oded Goldreich. Universal arguments and their applications. In *Computational Complexity, 2002. Proceedings. 17th IEEE Annual Conference on*, pages 162–171. IEEE, 2002.

[BHKR13] Mihir Bellare, Viet Tung Hoang, Sriram Keelveedhi, and Phillip Rogaway. Efficient garbling from a fixed-key blockcipher. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 478–492. IEEE, 2013.

[CADT14] Henry Carter, Chaitrali Amrutkar, Italo Dacosta, and Patrick Traynor. For your phone only: custom protocols for efficient secure function evaluation on mobile devices. *Security and Communication Networks*, 7(7):1165–1176, 2014.

[CHK+12] Seung Geol Choi, Kyung-Wook Hwang, Jonathan Katz, Tal Malkin, and Dan Rubenstein. Secure multiparty computation of boolean circuits with applications to privacy in on-line marketplaces. In *Topics in Cryptology–CT-RSA 2012*, pages 416–432. Springer, 2012.

[CLT14] Henry Carter, Charles Lever, and Patrick Traynor. Whitewash: Outsourcing garbled circuit generation for mobile devices. In *Proceedings of the 30th Annual Computer Security Applications Conference*, pages 266–275. ACM, 2014.

[CMTB13] Henry Carter, Benjamin Mood, Patrick Traynor, and Kevin Butler. Secure outsourced garbled circuit evaluation for mobile devices. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 289–304, Washington, D.C., 2013. USENIX.

[DCFGT12] Emiliano De Cristofaro, Sky Faber, Paolo Gasti, and Gene Tsudik. Genodroid: are privacy-preserving genomic tests ready for prime time? In *Proceedings of the 2012 ACM workshop on Privacy in the electronic society*, pages 97–108. ACM, 2012.

[DI05] Ivan Damgard and Yuval Ishai. Constant-round multiparty computation using a black-box pseudorandom generator. In *Advances in Cryptology–CRYPTO 2005*, pages 378–394. Springer, 2005.

[DIK+08] I. Damgard, Y. Ishai, M. Krøigaard, J.-B. Nielsen, and A. Smith. Scalable multiparty computation with nearly optimal work and resilience. In *Advances in Cryptology - CRYPTO 2008*, pages 241–261, 2008.

[DSZ14] Daniel Demmler, Thomas Schneider, and Michael Zohner. Ad-hoc secure two-party computation on mobile devices using hardware tokens. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 893–908, San Diego, CA, August 2014. USENIX Association.

[FKN94] Uri Feige, Joe Killian, and Moni Naor. A minimal model for secure computation. In *Proceedings of the twenty-sixth annual ACM symposium on Theory of computing*, pages 554–563. ACM, 1994.

[G+09] Craig Gentry et al. Fully homomorphic encryption using ideal lattices. In *STOC*, volume 9, pages 169–178, 2009.

[GMW87] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, STOC '87, pages 218–229, New York, NY, USA, 1987. ACM.

[HCE11] Yan Huang, Peter Chapman, and David Evans. Privacy-preserving applications on smartphones. In *USENIX Workshop on Hot Topics in Security*, 2011.

[HEKM11] Yan Huang, David Evans, Jonathan Katz, and Lior Malka. Faster secure two-party computation using garbled circuits. In *Proceedings of the 20th USENIX conference on Security*, SEC'11, pages 35–35, Berkeley, CA, USA, 2011. USENIX Association.

[HKK+14] Yan Huang, Jonathan Katz, Vladimir Kolesnikov, Ranjit Kumaresan, and Alex J Malozemoff. Amortizing garbled circuits. In *Advances in Cryptology–CRYPTO 2014*, pages 458–475. Springer, 2014.

[HSS+10] Wilko Henecka, Ahmad-Reza Sadeghi, Thomas Schneider, Immo Wehrenberg, et al. Tasty: tool for automating secure two-party computations. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 451–462. ACM, 2010.

[Hua12] Yan Huang. *Practical Secure Two-Party Computation*. PhD thesis, University of Virginia, 2012.

[IKO+11] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, Manoj Prabhakaran, and Amit Sahai. Efficient non-interactive secure computation. In Kenneth G. Paterson, editor, *EUROCRYPT*, volume 6632 of *Lecture Notes in Computer Science*, pages 406–425. Springer, 2011.

[IPS08] Yuval Ishai, Manoj Prabhakaran, and Amit Sahai. Founding cryptography on oblivious transfer — efficiently. In *Proceedings of the 28th Annual conference on Cryptology: Advances in Cryptology*, CRYPTO 2008, pages 572–591, Berlin, Heidelberg, 2008. Springer-Verlag.

[JS07] Stanislaw Jarecki and Vitaly Shmatikov. Efficient two-party secure computation on committed inputs. In *Proceedings of the 26th annual international conference on Advances in Cryptology*, EUROCRYPT '07, pages 97–114, Berlin, Heidelberg, 2007. Springer-Verlag.

[KMR11] S. Kamara, P. Mohassel, and M. Raykova. Outsourcing multi-party comptuation. Technical Report 2011/272, IACR ePrint Cryptography Archive, 2011. http://eprint.iacr.org/2011/272.

[KMR12] Seny Kamara, Payman Mohassel, and Ben Riva. Salus: a system for server-aided secure function evaluation. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 797–808. ACM, 2012.

[KMSB13] Ben Kreuter, Benjamin Mood, Abhi Shelat, and Kevin Butler. Pcf: A portable circuit format for scalable two-party secure computation. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 321–336, Berkeley, CA, USA, 2013. USENIX Association.

[KSS12] Benjamin Kreuter, Abhi Shelat, and Chih-Hao Shen. Billion-gate secure computation with malicious adversaries. In *Proceedings of the 21st USENIX conference on Security symposium*, Security'12, pages 14–14, Berkeley, CA, USA, 2012. USENIX Association.

[Lin13] Yehuda Lindell. Fast cut-and-choose based protocols for malicious and covert adversaries. In *CRYPTO*, pages 1–17, 2013.

[LP07] Yehuda Lindell and Benny Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In Moni Naor, editor, *EUROCRYPT*, volume 4515 of *Lecture Notes in Computer Science*, pages 52–78. Springer, 2007.

[LP11] Yehuda Lindell and Benny Pinkas. Secure two-party computation via cut-and-choose oblivious transfer. In Yuval Ishai, editor, *TCC*, volume 6597 of *Lecture Notes in Computer Science*, pages 329–346. Springer, 2011.

[LR14] Yehuda Lindell and Ben Riva. Cut-and-choose yao-based secure computation in the online/offline and batch settings. In *Advances in Cryptology–CRYPTO 2014*, pages 476–494. Springer, 2014.

[MF06] Payman Mohassel and Matthew K. Franklin. Efficiency tradeoffs for malicious two-party computation. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *Public Key Cryptography*, volume 3958 of *Lecture Notes in Computer Science*, pages 458–473. Springer, 2006.

[MGBF14] Benjamin Mood, Debayan Gupta, Kevin Butler, and Joan Feigenbaum. Reuse it or lose it: More efficient secure computation through reuse of encrypted values. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 582–596. ACM, 2014.

[MLB12] Benjamin Mood, Lara Letaw, and Kevin Butler. Memory-efficient garbled circuit generation for mobile devices. In *Financial Cryptography and Data Security*, pages 254–268. Springer, 2012.

[MNP+04] Dahlia Malkhi, Noam Nisan, Benny Pinkas, Yaron Sella, et al. Fairplay-secure two-party computation system. In *USENIX Security Symposium*, volume 4. San Diego, CA, USA, 2004.

[MR13] Payman Mohassel and Ben Riva. Garbled circuits checking garbled circuits: More efficient and secure two-party computation. In *CRYPTO*, pages 36–53, 2013.

[NNOB12] Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A new approach to practical active-secure two-party computation. In *Advances in Cryptology - Crypto 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 681–700. Springer, 2012.

[NO09] Jesper Buus Nielsen and Claudio Orlandi. Lego for two-party secure computation. In *Proceedings of the 6th Theory of Cryptography Conference on Theory of Cryptography*, TCC '09, pages 368–386, Berlin, Heidelberg, 2009. Springer-Verlag.

[PSSW09] Benny Pinkas, Thomas Schneider, Nigel P. Smart, and Stephen C. Williams. Secure two-party computation is practical. In *Proceedings of the 15th International Conference on the Theory and Application of Cryptology and Information Security: Advances in Cryptology*, ASIACRYPT '09, pages 250–267, Berlin, Heidelberg, 2009. Springer-Verlag.

[SS11] Abhi Shelat and Chih-Hao Shen. Two-output secure computation with malicious adversaries. In Kenneth G. Paterson, editor, *EUROCRYPT*, volume 6632 of *Lecture Notes in Computer Science*, pages 386–405. Springer, 2011.

[SS13] Abhi Shelat and Chih-hao Shen. Fast two-party secure computation with minimal assumptions. In *CCS*, pages 523–534. ACM, 2013.

[ST15] Nigel Smart and Stefan Tillich. Circuits of basic functions suitable for MPC and FHE. http://www.cs.bris.ac.uk/Research/CryptographySecurity/MPC/, 2015.

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science*, SFCS '86, pages 162–167, Washington, DC, USA, 1986. IEEE Computer Society.