Secure Multi-Party Computation of Boolean Circuits with Applications to Privacy in On-Line Marketplaces

Seung Geol Choi[*] Kyung-Wook Hwang[†] Jonathan Katz[*]
Tal Malkin[†] Dan Rubenstein[†]

Abstract

Protocols for generic secure multi-party computation (MPC) come in two forms: they either represent the function being computed as a *boolean* circuit, or as an *arithmetic* circuit over a large field. Either type of protocol can be used for any function, but the choice of which type of protocol to use can have a significant impact on efficiency. The magnitude of the effect, however, has never been quantified.

With this in mind, we implement the MPC protocol of Goldreich, Micali, and Wigderson, which uses a boolean representation and is secure against a semi-honest adversary corrupting any number of parties. We then consider applications of secure MPC in *on-line marketplaces*, where customers select resources advertised by providers and it is desired to ensure privacy to the extent possible. Problems here are more naturally formulated in terms of boolean circuits, and we study the performance of our MPC implementation relative to existing ones that use an arithmetic-circuit representation. Our protocol easily handles tens of customers/providers and thousands of resources, and outperforms existing implementations including FairplayMP [3], VIFF [10], and SEPIA [7].

1 Introduction

Protocols for secure multi-party computation allow a set of parties $P_1, \ldots, P_n$ to compute some function of their inputs in a distributed fashion, while revealing nothing to a coalition of corrupted parties about any honest party's input (or any group of honest parties' inputs), beyond what is implied by the output. Seminal results in cryptography dating to the 1980s [28, 12, 11] show that *any* polynomial-time function can be computed securely in the presence of coalitions of up to $n-1$ corrupted parties. For many years, the perception was that these were to be viewed as pure theoretical results with little practical relevance. This changed (to some extent) with the advent of Fairplay [22], an implementation of Yao's protocol for secure two-party computation that demonstrated for the first time that generic protocols were within the realm of feasibility. Since then, several other implementations of generic secure two-party and multi-party protocols have been developed [3, 21, 5, 10, 24, 7, 14], and this is currently an active area of research.

In this work our focus is on generic protocols for secure *multi-party* computation (MPC) in the *semi-honest* setting. (In the semi-honest setting, parties are assumed to follow the protocol

[*]Dept. of Computer Science, University of Maryland. This work was supported by DARPA. Email: {sgchoi,jkatz}@cs.umd.edu

[†]Dept. of Computer Science, Columbia University. Email: {kwhwang@ee,tal@cs,danr@cs}.columbia.edu

but coalitions of malicious parties may attempt to learn additional information from the joint transcript of their execution of the protocol. By "generic" we mean protocols that can be used to securely compute arbitrary functions.) There are, broadly speaking, two approaches taken by protocols for secure MPC: they either represent the function being computed as a *boolean* circuit, or as an *arithmetic* circuit over a (cryptographically) large field $\mathbb{F}$.[1] Although any function f can be computed using either type of protocol, the choice of representation affects the size of the circuit implementing f, and hence the overall efficiency of a secure protocol for computing f. The magnitude of the effect, however, has never been measured experimentally.

Most existing implementations of secure MPC rely on an arithmetic-circuit representation of the function, with VIFF [10] and SEPIA [7] serving as two prominent examples whose code is available for download. We are aware of only one existing implementation — FairplayMP [3] — of secure MPC using boolean circuits. As we will see, for certain problems a boolean-circuit representation is much more natural, and so it is important to have protocols of both types available. Indeed, the motivation for our work came from trying to apply secure MPC to privacy-preserving computation in *on-line marketplaces*, where customers select resources advertised by providers and it is desired to ensure privacy to the extent possible. (See the following section for details.) In doing so, we found that existing implementations of secure MPC were unsuitable or too inefficient for our purposes. Moreover, FairplayMP, VIFF, and SEPIA require an honest majority, even though resilience against an arbitrary number of corruptions is theoretically attainable.

1.1 Our Contributions

We implemented the classical MPC protocol of Goldreich, Micali, and Wigderson [12] (the *GMW protocol*), which uses a boolean-circuit representation for the function being computed and is secure against a semi-honest adversary controlling any number of corrupted parties. In our implementation, described in Section 2, we employ several optimizations to improve efficiency. Our code is publicly available[2] and we expect that, as with other systems, it will be useful in future work on privacy-preserving distributed computation.

With our system in place, any privacy-preserving multi-party computation can be solved, in principle, by defining an appropriate circuit for the task at hand. We designed circuits addressing three different (but related) problems in the context of on-line marketplaces where, generally speaking, *providers* advertise *resources* to be selected and subsequently utilized by *customers*, and the function of the marketplace is to match customers with providers in a way that optimizes some value under a certain set of constraints. We look at the following specific examples:

- **P2P content-distribution services** provide a marketplace where *content* is the resource, and providers advertise availability of content at peers [9, 17]. Here, a customer may want to determine which peer hosting the desired content is the best choice (e.g., closest, or having minimal delay) for retrieving that content.

- In **cloud computing** providers are cloud platforms (e.g., Amazon EC2, Microsoft Azure, etc.), resources are the services (e.g., storage, bandwidth, or processing) offered by each provider, and customers want to find the provider(s) offering services matching their needs at the cheapest price [1, 8, 27, 25, 26, 19].

[1] Of course, a boolean circuit can be viewed as a circuit over the field $\mathbb{F} = GF(2)$. The distinction is that protocols using arithmetic circuits require $1/|\mathbb{F}|$ to be negligible in order for security and correctness to hold.

[2] `http://www.ee.columbia.edu/~kwhwang/projects/gmw`.

- A **mobile social network** can be viewed as a marketplace where users are both customers and resources, and the provider helps users locate and connect to other users who share similar interests.

Formal definitions of the problems in each of the above settings are given in Section 3, and in Section 4 we describe optimized circuits solving each of them. For these problems, we find that it significantly helps to be able to work with boolean circuits rather than arithmetic circuits.

Finally, in Section 5 we evaluate the performance of our MPC protocol as applied to one of the above problems. (Since they have similar circuits, the other two problems should display similar results.) Our work shows that our protocol can be used to efficiently and securely implement a distributed marketplace with tens of providers/customers and thousands of resources over a wide-area network. Our implementation outperforms systems such as VIFF [10] and SEPIA [7], in part because we use boolean circuits rather than arithmetic circuits as those systems do.[3] Recall that another advantage of our protocol is that it provides security against any number of corruptions, whereas the cited implementations [3, 10, 7] require an honest majority.

1.2 Other Related Work

There are several existing implementations of secure two-party computation [22, 21, 24, 13, 14]. These are all specific to the two-party setting and do not yield protocols for three or more parties. Interestingly, and in contrast to other multi-party implementations [3, 10, 7] that *only* handle three or more parties, the GMW protocol we implement can handle any number of parties. For the two-party case, however, we expect our implementation to be roughly a factor of two slower than the best available system [14].

Other implementations of secure multi-party computation include [6, 4, 16]. The code for SIMAP [6] is not publicly available, and anyway SIMAP appears to be superseded by VIFF. Sharemind [4] handles three parties only. The work of Jakobsen et al. [16] is interesting since it achieves resilience against an arbitrary number of *malicious* corruptions. Their implementation is based on arithmetic circuits and, as reported by the authors, has worse performance than VIFF (though with better resilience).

2 MPC Implementation

We provide an overview of the GMW protocol, and details of our implementation. As stated earlier, the GMW protocol provides security against a semi-honest adversary corrupting any number of parties. (We refer to [11] for formal definitions of security.) Assuming semi-honest behavior is reasonable in settings where the codebase is difficult to change without detection, where software attestation can be used to convince other parties that correct software is being run, or where parties are trusted but must ensure secrecy of data for policy reasons or because of concerns about future break-ins.

[3] FairplayMP [3] also uses boolean circuits, but did not support multiple input values per party and crashed on the input sizes used. See Section 5 for further discussion.

2.1 Overview of the GMW Protocol

1-out-of-4 oblivious transfer. *Oblivious transfer* (OT) is a key building block of the GMW protocol. A 1-out-of-4 OT protocol is a two-party protocol in which there is a sender holding values (x_0, x_1, x_2, x_3) and a receiver holding an index $i \in \{0, \ldots, 3\}$; the receiver learns x_i, but neither the sender nor the receiver learn anything else; i.e., the receiver learns nothing about any other values held by the sender, and the sender learns nothing about the receiver's index.

Details of our OT implementation are given in Section 2.2.

The GMW protocol. The GMW protocol assumes the function f to be computed is represented as a boolean circuit consisting of XOR and AND gates or, equivalently, gates for addition and multiplication modulo 2. Let n denote the number of parties. In the GMW protocol the parties maintain random n-out-of-n shares $(s_{w1}, \ldots, s_{wn})$ of the value s_w on each wire w in the circuit; that is, party P_i holds share s_{wi} and all shares are random subject to $s_w = \bigoplus_i s_{wi}$. Setting up such shares on the input wires is easy: party P_i can provide input s_w on wire w by choosing random s_{wj} for $j \neq i$, sending s_{wj} to P_j, and locally setting $s_{wi} = s_w \oplus \left(\bigoplus_{j \neq i} s_{wj}\right)$. Shares on internal wires of the circuit can be computed inductively in the following way:

XOR gates. Say w is the output wire of an XOR gate with input wires u and v, and the parties have shares $(s_{u1}, \ldots, s_{un})$ and $(s_{v1}, \ldots, s_{vn})$ of s_u and s_v, respectively. Then each party P_i locally computes $s_{wi} = s_{ui} \oplus s_{vi}$, and one can observe that $(s_{w1}, \ldots, s_{wn})$ is a valid sharing of $s_w = s_u \oplus s_v$.

AND gates. Say w is the output wire of an AND gate with input wires u and v, and the parties have shares $(s_{u1}, \ldots, s_{un})$ and $(s_{v1}, \ldots, s_{vn})$ of s_u and s_v, respectively. Note that

$$\begin{aligned} s_w = s_u \cdot s_v &= \left(\sum_{i=1}^{n} s_{ui}\right) \cdot \left(\sum_{i=1}^{n} s_{vi}\right) \\ &= \sum_{i=1}^{n} s_{ui}s_{vi} + \sum_{i<j} (s_{ui}s_{vj} + s_{uj}s_{vi}). \end{aligned}$$

Each party P_i can compute $s_{ui}s_{vi}$ locally. As for the remaining term, each pair of parties P_i, P_j computes a random additive share of $s_{ui}s_{vj} + s_{uj}s_{vi}$ in the following way. P_j chooses a random bit $c_j^{\{i,j\}}$, and computes four values

$$c_j^{\{i,j\}}, \quad c_j^{\{i,j\}} \oplus s_{uj}, \quad c_j^{\{i,j\}} \oplus s_{vj}, \quad c_j^{\{i,j\}} \oplus s_{vj} \oplus s_{uj}$$

corresponding to the four possible values of P_i's shares s_{ui}, s_{vi}. Party P_i then acts as a receiver in 1-out-of-4 OT, with index determined by the actual values of its shares s_{ui}, s_{vi}, to obtain the appropriate value from P_j that we denote by $c_i^{\{i,j\}}$. Note that $c_i^{\{i,j\}} + c_j^{\{i,j\}} = (s_{ui}s_{vj} + s_{uj}s_{vi})$. Finally, each party P_i computes

$$s_{wi} = s_{ui}s_{vi} + \sum_{j \neq i} c_i^{\{i,j\}}.$$

It can be verified (see [11]) that $(s_{w1}, \ldots, s_{wn})$ is a (random) sharing of $s_w = s_u \cdot s_v$.

Observe that evaluation of XOR gates is essentially free, whereas evaluating AND gates requires $\binom{n}{2}$ invocations of 1-out-of-4 oblivious transfer.

Once a sharing $(s_{w1}, \ldots, s_{wn})$ of an output wire w is obtained, the value s_w can be reconstructed by having each party privately send its share to all other parties. It is also possible for only some specific party to learn a given output value by sending shares to that party only. We note that this is the only step in the protocol where private channels are needed, and then only if more than one party is to learn a given output value.

2.2 Oblivious Transfer Protocols

As noted in the previous section, oblivious transfer is a key building block of the GMW protocol; it is also the most computationally expensive part of the protocol, since it is the only aspect of the protocol that relies on public-key techniques. As described above, the GMW protocol requires one invocation of 1-out-of-4 OT per (ordered) pair of parties each time an evaluation of an AND gate is performed, and so m executions of OT (per ordered pair of parties) to evaluate a circuit containing m AND gates. We can improve the overall efficiency, however, using two techniques:

- Using *OT pre-processing* [2], each pair of parties can perform m oblivious transfers *on random inputs* at the outset of the protocol, and then (very efficiently) use the pre-computed values thus obtained to achieve the functionality of oblivious transfer on their actual inputs when evaluating an AND gate. Thus, all the oblivious transfers that will be needed throughout the entire protocol can be run *in parallel* at the beginning of the protocol.

- Using *OT extension* [15, 20], it is possible to achieve the functionality of m invocations of 1-out-of-4 OT at essentially the cost of k invocations of 1-out-of-4 OT of m-bit strings, where k is a statistical security parameter. (More precisely, the marginal cost for each additional OT is just a small number of hash computations.) Security here is based on the assumption that the hash function is *correlation robust* [15].

Combining these optimizations, each pair of parties needs only run k (parallel) invocations of some "base" OT protocol (for m-bit strings) at the outset of the GMW protocol; these can be converted to $m \gg k$ OT executions (on bits) using OT extension; these m "pre-processed" OTs can then be used, as needed, during the rest of the protocol. It remains only to specify the "base" 1-out-of-4 OT protocol we use.

We take as our base OT protocol the one by Naor and Pinkas [23], secure under the decisional Diffie-Hellman (DDH) assumption in the random-oracle model. Their protocol (actually, a version implementing k parallel executions of their protocol) is described in Figure 1 for completeness.

2.3 Implementation Details

We implemented the GMW protocol in C++. It takes as input a file containing a description of a boolean circuit for the function f of interest. (All parties are assumed to be running with identical copies of the circuit.) See Appendix A for an example. Unlike FairplayMP [3], we do not provide a mechanism for compiling a high-level language into a boolean circuit.

Parallelism. Nowadays, it is common for computers to have multiple cores. We use multi-threaded programming so as to take advantage of the available parallelism. In particular, each OT execution is performed by a separate thread.

In the OT extension protocol, we optimize execution time by having parties send values as soon as they can be computed, rather than waiting for the other party to finish sending. (This does not

k parallel invocations of 1-out-of-4 OT

Let $g, \mathbb{G}$, and q be fixed, where $\mathbb{G}$ is a cyclic group of prime order q, and g is a generator of $\mathbb{G}$. Let $H : \{0,1\}^* \to \{0,1\}^m$ be a hash function.

Inputs.

S holds $\{(x_0^j, x_1^j, x_2^j, x_3^j)\}_{j \in [k]}$ with $x_i^j \in \{0,1\}^m$.

R holds $(r_1, \ldots, r_k)$ where $r_j \in \{0, \ldots, 3\}$.

The protocol.

1. **S** chooses $\alpha \leftarrow \mathbb{Z}_q$ and computes $c_0 = g^\alpha$, and also chooses $c_1, c_2, c_3 \leftarrow \mathbb{G}$. It sends $c_0, \ldots, c_3$ to **R**.

For $j \in [k]$ the parties do:

2. **R** chooses $\beta_j \leftarrow \mathbb{Z}_q$. If $r_j = 0$ then it sets $d_j = g^{\beta_j}$; else, it sets $d_j = c_{r_j}/g^{\beta_j}$. Finally, **R** sends d_j to **S**.
3. **S** computes $e_0 = d_j^\alpha$ and $e_i = (c_i/d_j)^\alpha$ for $i \in \{1,2,3\}$. Then **S** sends $\bar{x}_i^j = H(e_i, j, i) \oplus x_i^j$ to **R** for $i \in \{0, \ldots, 3\}$.
4. **R** computes $c_0^{\beta_j} = e_{r_j}$, and then outputs $x_{r_j}^j = \bar{x}_{r_j}^j \oplus H(e_{r_j}, j, r_j)$.

Figure 1: The Naor-Pinkas OT protocol.

affect security, since this occurs at fixed times that are independent of the parties' inputs.) We also parallelize the GMW protocol itself by computing all multiplication gates at the same level of the circuit in parallel.

Random oracle. We use SHA-1 to implement a random oracle H with arbitrary output length by defining

$$H(M) = \text{SHA-1}(\mathsf{seed}, 0) || \text{SHA-1}(\mathsf{seed}, 1) || \cdots ,$$

where $\mathsf{seed} = \text{SHA-1}(M)$. Note that seed need only be computed once. We use the SHA-1 implementation of PolarSSL (`http://polarssl.org`).

Oblivious transfer. For our base OT protocol we use the Naor-Pinkas protocol (see Figure 1) with group $\mathbb{G}$ a subgroup of $\mathbb{Z}_p$ of prime order q, and $p = 2q + 1$ with p prime. In our default implementation, p is a 512-bit integer.[4] We modified the modular-arithmetic module of NTL (`http://www.shoup.net/ntl`) to be thread-safe, and use it in implementing the base OT protocol.

Recall we use OT extension to improve efficiency. By default, we use statistical security parameter $k = 80$ in our implementation. Messages are transmitted in chunks of reasonable size to obtain a balance between the idle time and the number of socket calls.

[4]This value of p is artificially low and does not provide sufficient security for practical applications. However, since the time for running the initial oblivious transfers is a small fraction of the overall running time, we do not expect that using larger p will significantly change our results.

3 Problem Definitions

We introduce three problems in the context of on-line marketplaces where, generally speaking, *providers* advertise *resources* to be selected and subsequently utilized by *customers*, and the function of the marketplace is to match customers with providers so as to optimize some value under a certain set of constraints. As highlighted in the Introduction, we look at examples in the settings of P2P content distribution, cloud computing, and mobile social networks.

As a toy example, consider a customer who wishes to buy a car (resource) from one of several dealers (providers). The customer is interested in several different models of cars (but not all models); the different providers offer a variety of models (not all of which interest the customer); and each provider prices each model independently. The customer wishes to find an acceptable car at the lowest cost, without revealing the set of models he or she is interested in; the providers do not want to reveal their prices. Secure MPC allows the customer to learn the identity of a provider selling an acceptable model at the lowest price, with the customer learning no other prices (or which models are sold by each provider), and with the providers learning only of the customer's willingness to buy some particular model at the given price.

More generally, and a bit more formally, let R be some set of resources. The input of each provider P_i is a collection of values for the resources in some subset $R_i \subseteq R$; i.e., P_i's input is of the form $\{v_r^i\}_{r \in R_i}$. (If desired, it is also possible for each P_i to use some default value $v_r^i = \perp$ for $r \notin R_i$; in that case, we may simply write P_i's input as $\{v_r^i\}_{r \in R}$.) We look at marketplaces where the computation can be broken into the following two steps, which will be executed as a *single* secure computation (so only the final output is revealed, not the intermediate results after the first step):

1. First, for each provider P_i and resource $r \in R_i$, compute a *scoring function* $\mathsf{sc}_r^i = \mathsf{Score}(i, r, v_r^i, x_n)$, where x_n denotes the private input of the customer. (In the running toy example, each model is scored by its offered price if the model is of interest to the customer, and by ∞ otherwise.)

2. Next, apply a *best-match function* $\mathcal{B}$ to the set of sc_r^i values to obtain a result that is given to the customer. (In the toy example, $\mathcal{B}$ outputs (i, r, sc_r^i) with minimum sc_r^i.)

We allow the scoring function to be completely general. For the best-match function we consider two possibilities: either $\mathcal{B}$ returns a single (i, r) maximizing/minimizing sc_r^i (with ties broken arbitrarily, and with or without including sc_r^i as part of the output), or $\mathcal{B}$ returns the set of all (i, r) for which the score sc_r^i exceeds/is lower than some threshold. In the following subsections, we instantiate this general framework in several specific scenarios. However, it should be clear that our approach is quite general.

3.1 P2P Content-Distribution Services

In our P2P content-distribution setting, content is replicated across various P2P servers or source peers (such as seeders) whose pairwise communications are measured (and perhaps even controlled) by network providers such as ISPs. Before a peer starts downloading content, he or she would like to find out the best source peer (with respect to network bandwidth, end-to-end delays, throughput, and so on) from which to receive the content.

Here the providers are the ISPs and the resources are the source peers themselves (which for simplicity we identify with their indices). Let R be the set of source peers, with $|R| = k$. We assume that the ISP to which each source peer is bound is public knowledge, so ISP P_i is associated with some set of peers R_i. The input of each ISP/provider P_i is the measured bandwidth v_r^i for each peer/resource $r \in R_i$. The customer knows which peers have a replica of the item it wishes to retrieve, and holds as secret input a vector $x_n = (b_1, \ldots, b_k)$ where $b_r = 1$ iff peer/resource r has the desired content, and $b_r = 0$ otherwise. The objective is for the customer to find the best (e.g., highest-bandwidth) peer among those holding the desired content, without revealing which source peers have the content; the ISPs do not want to reveal the bandwidth of their peers.

Here the scoring function can be defined as:

$$\mathsf{Score}\left(i, r, v_r^i, (b_1, \ldots, b_k)\right) = \begin{cases} v_r^i & \text{if } b_r = 1 \\ 0 & \text{otherwise} \end{cases},$$

and the best-match function $\mathcal{B}$ returns i, r maximizing sc_r^i. (In fact, it suffices to return r here since the provider to which r is bound is irrelevant and anyway known.)

3.2 Cloud Computing

In this setting, providers offer various service packages and the customer wants to select the service package meeting its needs at the lowest available price. The service packages offered by the providers are the resources here, and each such resource r has a value $v_r = (q_r, p_r)$ that is composed of its service quality q_r and price p_r. (For simplicity, we treat service quality as a one-dimensional quantity, e.g., CPU cycles. Our treatment can easily be generalized.) The customer holds input (q, p), where q represents a minimum acceptable service quality and p is a maximum budget. Two scenarios can be considered: either the customer wants to find the cheapest resource r with $q_r \geq q$, or the highest-quality resource r with $p_r \leq p$; each of these cases is treated below. In either case, the customer never reveals its budget or its service requirements to any of the providers, nor do the providers reveal to the customer (or to each other) what service packages they are offering.

Lowest-price selection. In this formulation, the customer seeks the package that satisfies its requirements at the lowest price. Here we may define the scoring function as:

$$\mathsf{Score}\left((i, r, q_r^i, p_r^i), (q, p)\right) = \begin{cases} p_r^i & \text{if } q_r^i \geq q \text{ and } p_r^i \leq p \\ \infty & \text{otherwise} \end{cases}.$$

The best-match function $\mathcal{B}$ returns an i, r minimizing sc_r^i.

Highest-quality selection. Here the customer seeks the package that meets its budget while giving the highest quality service. Now we may define the scoring function as:

$$\mathsf{Score}\left((i, r, q_r^i, p_r^i), (q, p)\right) = \begin{cases} q_r^i & \text{if } q_r^i \geq q \text{ and } p_r^i \leq p \\ -\infty & \text{otherwise} \end{cases},$$

and the best-match function returns an i, r maximizing sc_r^i.

3.3 Mobile Social Networks

Here we consider a scenario where a user in a social network wants to identify nearby users who share common interests. Now the resources and providers are just the set R of all users (and the customer is one of the users as well), and the value of each "resource" (i.e., user) is that user's location and set of interests.

We assume that each user knows only about its own location and interests. Thus for each $r \in R$ we define $v_r^r = (\ell_r, H_r)$, where ℓ_r is the location of user r, and H_r is the set of that user's interests (perhaps represented as a bit-vector). The customer's input consists of (ℓ, H, δ) where ℓ is the location of the customer, H is the set of interests she wants a potential match to share, and δ is the distance radius in which she wants to search. We consider a few alternatives for what the customer wants as output.

Find all close matches. In this formulation, the customer wants to find all users within distance δ who share interests H. We may then define

$$\begin{aligned}&\mathsf{Score}\left((r, \ell_r, H_r),\ (\ell, H, \delta)\right)\\ &\quad = \begin{cases} 1 & \text{if } H \subseteq H_r \text{ and } |\ell_r - \ell| \leq \delta \\ 0 & \text{otherwise} \end{cases},\end{aligned}$$

and the best-match function returns the set of all r such that $\mathsf{sc}_r^r = 1$.

Find closest match. Here the customer wants to find the *closet* user who matches her interests. Now, define

$$\begin{aligned}&\mathsf{Score}\left((r, \ell_r, H_r),\ (\ell, H, \delta)\right)\\ &\quad = \begin{cases} |\ell_r - \ell| & \text{if } H \subseteq H_r \text{ and } |\ell_r - \ell| \leq \delta \\ \infty & \text{otherwise} \end{cases}\end{aligned}$$

The best-match function returns an r minimizing sc_r^r.

Find best resource. Now the customer would like to obtain the resource within radius δ that shares as many interests as possible. We thus define

$$\begin{aligned}&\mathsf{Score}\left((\ell_r, H_r),\ (\ell, H, \delta)\right)\\ &\quad = \begin{cases} |H_r \cap H| & \text{if } |\ell_r - \ell| \leq \delta; \\ -\infty & \text{otherwise} \end{cases},\end{aligned}$$

and the best-match function returns r maximizing sc_r^r.

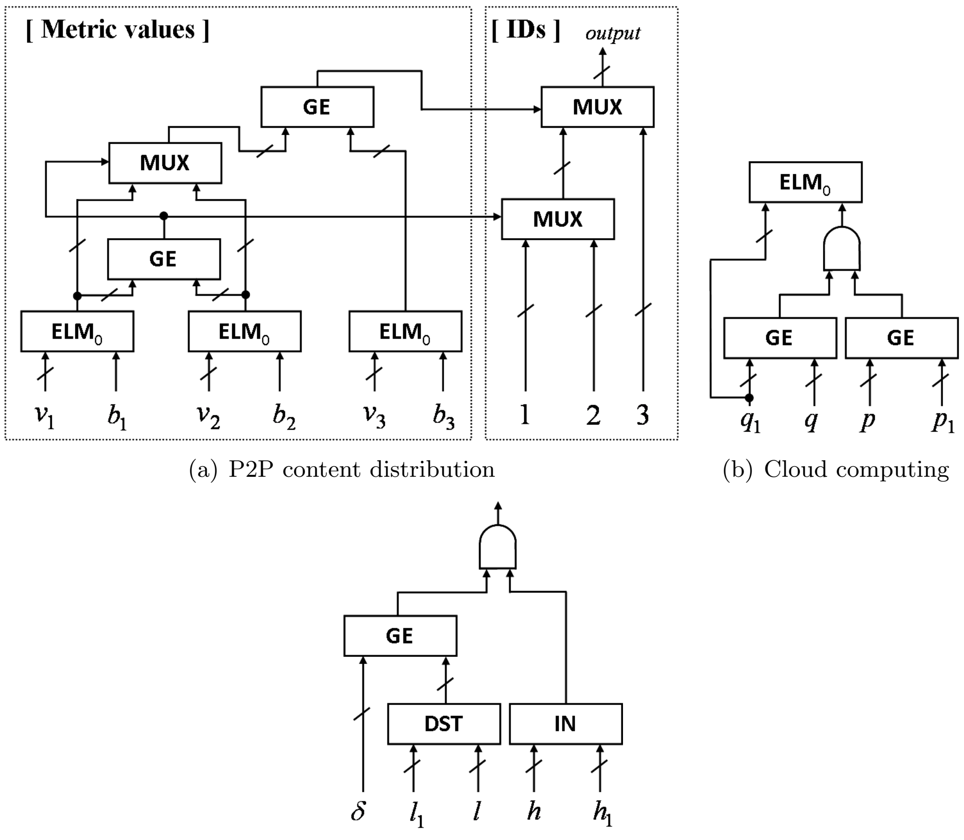
4 Boolean Circuit Constructions

We describe circuits solving the problems described in the previous section. Since XOR gates are essentially "free" to evaluate in the GMW protocol, while evaluating each AND gate requires cryptographic computations, our aim is to minimize the number of AND gates in the circuits.

In our boolean circuits, integers are treated as binary strings of some length ℓ. The length must be set sufficiently long to handle any value the integer might take. We use 0^ℓ and 1^ℓ in place of $-\infty$ and ∞ when computing scores.

General circuit layout. Recall that our online marketplace problems all consist of the following two main operations.

1. (Function Score.) The resources are scored according to the customer's interest.



(a) P2P content distribution

(b) Cloud computing

(c) Mobile social networks

Figure 2: Boolean circuits for the online marketplace problem with one customer, one provider, and three resources $R = \{1, 2, 3\}$: (a) the entire circuit diagram for the P2P content-distribution problem, (b) the additional circuitry required to score resources for the cloud computing application (highest-quality selection), and (c) the additional circuitry required for the mobile social networks application (find-all-close-matches).

2. (Function $\mathcal{B}$.) The scores are compared to find the "best" one.

In Figure 2, we show boolean circuits as a block diagrams. Figure 2(a) shows the entire process required for the P2P content-distribution problem, and Figures 2(b) and 2(c) indicate the components additionally needed for the other two applications. (For simplicity, the circuits in the figure consider the case when there are three resources.)

Circuit sizes. Let k be the total number of resources that the providers advertise, and let ℓ be the number of bits required to represent the score of each resource. For all the online marketplace problems, the total number of the gates required is $\mathcal{O}(k \cdot (\ell + \log_2 k))$.

4.1 Building Blocks for Circuit Constructions

We first describe some general building blocks that all our circuits will share. Generally, we use the circuits given by Kolesnikov et al. [18] who also aimed to minimize the number of AND gates.

Gate ADD. Given two ℓ-bit integers $a = (a_1, \ldots, a_\ell)$ and $b = (b_1, \ldots, b_\ell)$, this gate outputs the $(\ell + 1)$-bit integer $a + b$. The circuit description is as follows:

$$\begin{aligned}
&c_1 = 0\\
&\text{for } i \in [\ell] : c_{i+1} = c_i \oplus ((a_i \oplus c_i) \odot (b_i \oplus c_i))\\
&\text{for } s \in [\ell] : s_i = a_i \oplus b_i \oplus c_i\\
&\mathsf{ADD}(a, b) = (s_1, s_2, \ldots, s_\ell, c_{\ell+1})
\end{aligned}$$

Gate SUB. Given two ℓ-bit integers a and b with $a \geq b$, this gate outputs the ℓ-bit integer $a - b$.

$$\begin{aligned}
&c_1 = 1\\
&\text{for } i \in [\ell - 1] : c_{i+1} = a_i \oplus ((a_i \oplus c_i) \odot (b_i \oplus c_i))\\
&\text{for } s \in [\ell] : d_i = a_i \oplus b_i \oplus c_i \oplus 1\\
&\mathsf{SUB}(a, b) = (d_1, d_2, \ldots, d_\ell)
\end{aligned}$$

Gates GT and GE. Given two ℓ-bit integers a and b, gate GT (resp., gate GE) outputs 1 iff $a > b$ (resp., $a \geq b$) and 0 otherwise. The circuit description is as follows:

$$\begin{aligned}
&c_1 = 0\\
&\text{for } i \in [\ell] : c_{i+1} = a_i \oplus ((a_i \oplus c_i) \odot (b_i \oplus c_i))\\
&\mathsf{GT}(a, b) = c_{\ell+1}\\
&\mathsf{GE}(a, b) = 1 \oplus \mathsf{GT}(b, a)
\end{aligned}$$

Gate MUX. Given two ℓ-bit strings a and b, and a bit s, this gate outputs a if $s = 0$ and outputs b otherwise. The circuit description is as follows:

$$\begin{aligned}
&\text{for } i \in [\ell]\text{: } m_i = b_i \oplus ((s \oplus 1) \odot (a_i \oplus b_i))\\
&\mathsf{MUX}(a, b, s) = (m_1, m_2, \ldots, m_\ell)
\end{aligned}$$

Gate ELM. Given an ℓ-bit string a and a bit b, gate ELM_0 (resp., gate ELM_1) outputs a if $b = 1$ and 0^ℓ (resp., 1^ℓ) otherwise. The circuit description is as follows:

$$\begin{aligned}
&\mathsf{ELM}_0(a, b) = (a_1 \odot b, \ldots, a_\ell \odot b)\\
&\mathsf{ELM}_1(a, b) = \mathsf{MUX}(1^\ell, a, b).
\end{aligned}$$

4.2 P2P Content-Distribution Services

See Section 3.1 for a description of this problem.

Scoring. For this application, the scoring function Score just checks if $b_r = 1$. If so, it outputs the value v_r^i; otherwise, it outputs 0. This can be handled by a single ELM_0 gate:

$$\mathsf{sc}_r^i = \mathsf{ELM}_0(v_r^i, b_r).$$

Best-matched resource. The best-match function should output i, r maximizing sc_r^i. (Actually, it suffices to output r as discussed in Section 3.1.) For each r there is a unique (publicly known) provider P_i for which $v_r^i \neq 0$; thus, if there are k resources (i.e., source peers) overall, then we need only compare k values of sc_r^i in order to determine the maximum value. We can do this using $k-1$ comparisons. For example, in the three-resource case in Figure 2(a) there are three resources at the bottom level; at the next level, the winner between the first and the second is compared with the third to determine the maximum at the final level.

To achieve our goal of outputting the *index* of the resource with the maximum score, not only the scores but also the indices should be involved in this tournament. Therefore, at the bottom level of the circuit we actually have pairs (r, sc_r) (where $\mathsf{sc}_r = \mathsf{sc}_r^i$ for the unique P_i such that $v_r^i \neq 0$) that are fed into the tournament tree. When comparing (a, sc_a) and (b, sc_b), we first compute $c = \mathsf{GE}(\mathsf{sc}_a, \mathsf{sc}_b)$. The value propagated to the next level is then given by

$$\mathsf{MUX}((b, \mathsf{sc}_b), (a, \mathsf{sc}_a), c).$$

Thus, each pairwise comparison in the tournament consists of one GE gate and a MUX gate. The output is the index of the final winner.

By comparing the final score with 0^ℓ and using ELM_0 with the index of the winner, the best-match function $\mathcal{B}$ can be modified so that it outputs nothing if the maximum score is 0.

Number of AND gates. The number of AND gates used in the circuit is approximately $k \cdot (3\ell + \lceil \log_2 k \rceil)$.

4.3 Cloud Computing

Recall that the score of each resource with index r is (p_r, q_r) and that the input of the customer is (p, q).

Scoring resources. For both lowest-price selection and highest-quality selection, we need to check two things:

- Check if $q_r^i \geq q$; this is handled by $\mathsf{GE}(q_r^i, q)$.
- Check if $p_r^i \leq p$; this is handled by $\mathsf{GE}(p, p_r^i)$.

For highest-quality selection, the following expression can be used for scoring:

$$\mathsf{sc}_r^i = \mathsf{ELM}_0(q_r^i, \mathsf{GE}(q_r, q) \odot \mathsf{GE}(p, p_r^i)).$$

Figure 2(b) shows the block diagram for computing sc_r^i. For lowest-price selection, the scoring expression is as follows:

$$\mathsf{sc}_r^i = \mathsf{ELM}_1(p_r^i, \mathsf{GE}(q_r^i, q) \odot \mathsf{GE}(p, p_r^i)).$$

Best-matched resource. In the case of highest-quality selection, the circuit logic is exactly the same as in the case of the P2P content-distribution example. The case of lowest-price selection is also essentially the same; the only difference is to select the resource with the minimum score rather than the maximum score.

Number of AND gates. The number of AND gates is approximately $k \cdot (5\ell + \lceil \log_2 k \rceil)$ for both problems.

4.4 Mobile Social Networks

Recall that the input of provider P_r is its location ℓ_r and set of interests H_r. Likewise, the input of the customer is also its location and interest along with a distance limit δ. (See Section 3.3.)

Scoring resources. To handle the scoring function Score, we need additional building blocks.

- When computing $|\ell_r - \ell|$, we assume each location is two-dimensional. For simplicity, we measure the distance between $\ell_r = (\ell_r^1, \ell_r^2)$ and $\ell = (\ell^1, \ell^2)$ by

$$|\ell_r^1 - \ell^1| + |\ell_r^2 - \ell^2|.$$

  We define a DST gate to compute the above. Its circuit description is as follows:

$$\begin{array}{ll} c^1 = \mathsf{GE}(\ell_r^1, \ell^1) & c^2 = \mathsf{GE}(\ell_r^2, \ell^2) \\ a^1 = \mathsf{MUX}(\ell^1, \ell_r^1, c^1) & a^2 = \mathsf{MUX}(\ell^2, \ell_r^2, c^2) \\ b^1 = \mathsf{MUX}(\ell^1, \ell_r^1, 1{\oplus}c^1) & b^2 = \mathsf{MUX}(\ell^2, \ell_r^2, 1{\oplus}c^2) \end{array}$$
$$\mathsf{DST}(\ell_r, \ell) = \mathsf{ADD}(\mathsf{SUB}(a^1, b^1), \mathsf{SUB}(a^2, b^2))$$

- To compare interests, each set (i.e., H and H_r) is represented as a bit-vector of length ℓ such that its ith bit is set when it contains the ith interest. Let $H = (\alpha_1, \ldots, \alpha_\ell)$ and $H_r = (\beta_1, \ldots, \beta_\ell)$. We define a gate $\mathsf{CAP}(H, H_r)$ which outputs $H \cap H_r$, represented also as an ℓ-bit string:

$$\mathsf{CAP}(H, H_r) = (\alpha_1 \odot \beta_1, \ldots, \alpha_\ell \odot \beta_\ell).$$

  Then we define CNT gate that counts the numbers of 1s in a bit-string a. We assume the length of a is a power of 2 for simplicity. Roughly speaking, this use a binary tree where each internal node at the ith level is an ADD gate with i-bit integer inputs, and each leaf is a bit of a (e.g., a_1 or a_2). The output of the top level ADD gate will be the output of CNT.

$$\mathsf{CNT}(a_1, \ldots, a_\ell) = c_{1,\ell}$$

  Here, $c_{s,e}$ is inductively defined as follows:

$$c_{s,e} = \begin{cases} \mathsf{ADD}(c_{s,m}, c_{m+1,e}) & \text{if } s > e; \\ a_s & \text{if } s = e \end{cases}$$

  where $m = \lceil (e-s)/2 \rceil$. Then, we have

$$|H \cap H_r| = \mathsf{CNT}(\mathsf{CAP}(H, H_r)).$$

  We also define a gate $\mathsf{IN}(H, H_r)$ which outputs 1 iff $H \subseteq H_r$, and 0 otherwise:

$$\begin{array}{l} \text{for } i \in [\ell]\text{: } s_i = 1{\oplus}(\alpha_i \odot (1{\oplus}\beta_i))) \\ \mathsf{IN}(H, H_r) = s_1 \odot s_2 \odot \cdots \odot s_\ell. \end{array}$$

Now we show how to compute the Score functions. For the problem of find-all-close-matches, the value I_r is computed as follows:

$$\mathsf{sc}_r = d \odot \mathsf{IN}(H, H_r),$$

where $d = \mathsf{GE}(\delta, \mathsf{DST}(\ell_r, \ell))$. The value sc_r is a boolean value indicating whether the provider r is close and has matching interests. For the problem of find-close-match, the score sc_r is computed as follows:

$$\mathsf{sc}_r = \mathsf{ELM}_1(\mathsf{DST}(\ell_r, \ell), \mathsf{IN}(H, H_r)).$$

For the problem of finding the best resource, the score sc_r is computed as

$$\mathsf{sc}_r = \mathsf{ELM}_0(\mathsf{CNT}(\mathsf{CAP}(H, H_r)), d).$$

Best-matched resource. For the problem of finding all close matches, the output of the circuit is simply $(I_1, \ldots, I_k)$. The other two cases can be handled as in the case of the content-distribution problem by finding the index with the maximum score. Additional information (i.e., the maximum score itself) can be computed by using a MUX gate at the final comparison stage.

Number of AND gates. For the problem of find-all-close-matches, the number of AND gates is approximately $12\ell k$. For the find-close-match and find-best-resource problems, the number of AND gates is approximately $k \cdot (14\ell + \lceil \log_2 k \rceil)$ and $k \cdot (11\ell + (\ell/2 + 3)\lceil \log_2 \ell \rceil + \lceil \log_2 k \rceil)$, respectively.

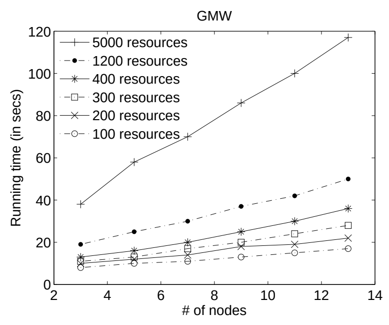
5 Performance Evaluation

We evaluate the performance of our implementation in both a local-area network (LAN) and a wide-area network (using PlanetLab, `http://www.planet-lab.org/`), and compare it to existing systems for secure MPC. In our experiments we consider only the P2P content-distribution problem formulated in Section 3.1 with $\ell = 16$, but since circuits for the other two problems are similar (in terms of both circuit depth and the number of AND gates), we expect the results to be similar for those problems. We let `GMW` refer to our solution for this problem, obtained by applying our GMW implementation to the circuit described in Section 4.2. All reported measurements are based on averages over 10 runs of the experiment in question.

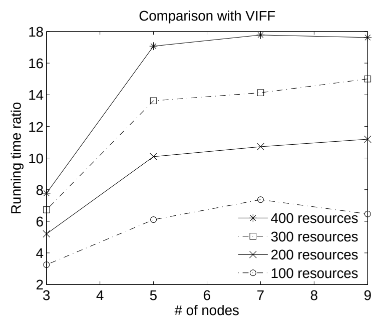
5.1 Local-Area Network

Our first set of experiments is performed in a cluster consisting of multiple Linux host nodes, each containing two Intel Xeon 2.80GHz CPUs and 4GB RAM. We use one host per participant in the protocol, so that an experiment with n providers requires $n + 1$ host machines. We set up our experiments so the customer chooses half of the resources offered by each provider to be "of interest". Note that the client's inputs do not affect performance in any way, since the same underlying circuit is evaluated regardless of the customer's input; indeed, if performance were affected by the customer's input then the protocol could not be secure!

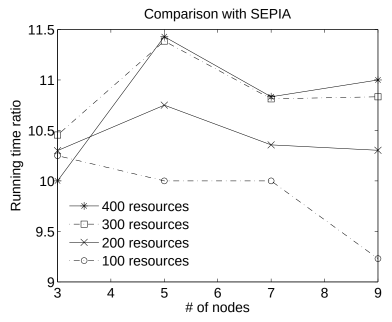
We ran experiments using from 3 to 13 nodes, and 100 to 5,000 resources. (This represents the aggregate offered by all providers.) For this problem, the number of AND gates being evaluated depends on the number of resources only (it is independent of the number of nodes) and ranges from about 5,500 AND gates (for 100 resources) to roughly 305,000 AND gates (for 5,000 resources). The running time is plotted in Figure 3(a), and the total bandwidth (between all parties) is shown in Figure 4(a).



(a) GMW



(b) Ratio of running times ($\frac{\texttt{VIFF}}{\texttt{GMW}}$)



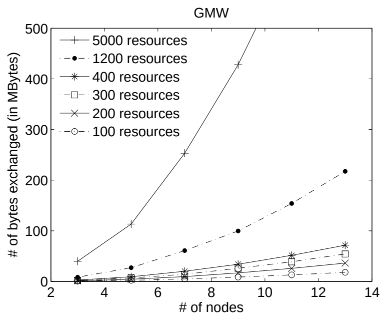
(c) Ratio of running times ($\frac{\texttt{SEPIA}}{\texttt{GMW}}$)

Figure 3: Running times in a LAN.

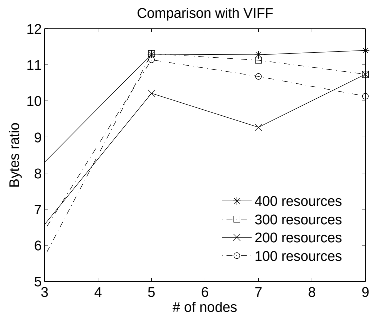
For a fixed number of resources, the bandwidth grows quadratically with the number of nodes; this is because each pair of parties communicates for every AND gate being evaluated. The running time scales linearly with the number of nodes since all nodes work in parallel, and the work per node increases in direct proportion to the number of other nodes with which it communicates. Although difficult to see from the plots, for a fixed number of parties the running time and bandwidth increase roughly linearly in the number of resources k; this is because the circuit size grows roughly linearly in k (actually, it grows as $k \log k$ but the effect of the additional logarithmic term is difficult to detect).

We also measured the marginal time to evaluate a single AND gate (i.e., the time required to evaluate an additional AND gate, once the number of AND gates is large). We use marginal time because there is a fixed cost for the initial set of oblivious transfers performed by the parties, but then oblivious-transfer extension is used to get additional OTs at much lower cost (see Section 2.2). The measured marginal cost per AND gate ranged from 50 μs (for 3 parties) to 340 μs (for 13 parties).

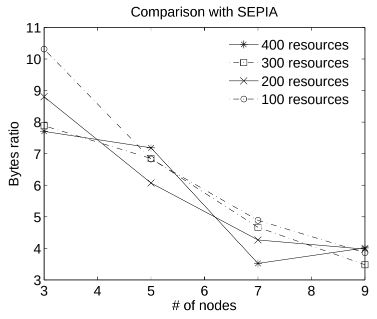
Comparison to existing work. We applied other existing implementations of secure MPC to the same problem. Unfortunately, despite contacting the authors we were unable to get a working implementation using FairplayMP [3] since we found that it did not support providing users with multiple inputs, and it would crash (when parties were provided with a single input) on inputs more than 16 bits long. However, we were able to compare our protocol with implementations in (the semi-honest version of) VIFF [10] and SEPIA [7]. We ran both VIFF and SEPIA over insecure (i.e., non-SSL-protected) channels even though private channels are needed to ensure security against an eavesdropping adversary for those protocols. (In contrast, for GMW a secure channel is not needed if



(a) GMW



(b) Ratio of bytes transferred ($\frac{\texttt{VIFF}}{\texttt{GMW}}$)



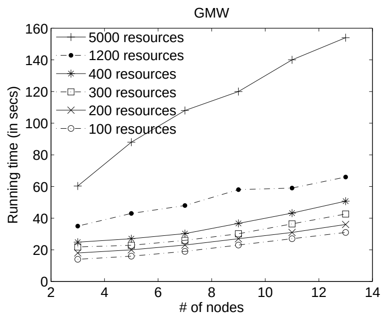
(c) Ratio of bytes transferred ($\frac{\texttt{SEPIA}}{\texttt{GMW}}$)

Figure 4: Total bytes transferred among all nodes.

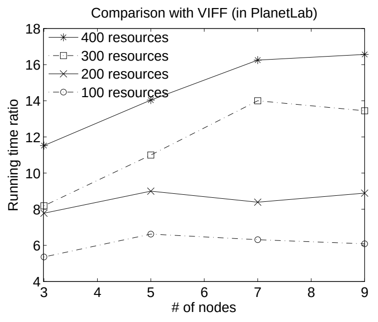
only one party learns the output; when multiple parties learn the output, only the final messages from the parties need to be encrypted.) In SEPIA, parties provide their inputs to "privacy peers" that run a secure computation protocol on their behalf; when we refer to "nodes" in SEPIA we mean the number of privacy peers.

In contrast to GMW, VIFF and SEPIA utilize arithmetic circuits where each wire carries an element of a large field $\mathbb{F}$ (with $\log |\mathbb{F}| \approx 64$ in each case), and gates perform addition or multiplication in $\mathbb{F}$. (Similar to the GMW case, addition is essentially "for free" whereas multiplication is "expensive".) The boolean circuit we used for GMW is easily adapted for VIFF/SEPIA as follows:

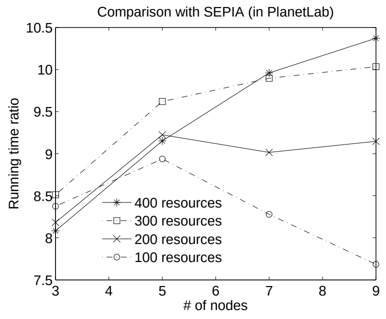
- Boolean values are easily represented as elements of $\mathbb{F}$ since $0, 1 \in \mathbb{F}$. Note AND$(a,b) = ab$ even when multiplication is done over $\mathbb{F}$, as long as $a, b \in \{0, 1\}$.
- ℓ-bit integers can be represented as elements of $\mathbb{F}$, since $|\mathbb{F}| \gg 2^\ell$ for the value of ℓ we use. Because of this, addition and subtraction gates are now trivial to implement, since they correspond exactly to addition and subtraction over $\mathbb{F}$.
- VIFF and SEPIA already provide comparison gates.
- MUX gates are also easily implemented over an arithmetic field, since $\mathsf{MUX}(a, b, s) = a(1 - s) + bs$ when $s \in \{0, 1\}$. We also express $\mathsf{ELM}_1(a, b) = \mathsf{MUX}(v, a, b)$, where $v \in \mathbb{F}$ is chosen sufficiently large. Finally, we have $\mathsf{ELM}_0(a, b) = ab$ when $b \in \{0, 1\}$.
- XOR can be computed as $\mathsf{XOR}(a, b) = a + b - 2ab$.



(a) Running time of GMW



(b) Ratio of running times ($\frac{\texttt{VIFF}}{\texttt{GMW}}$)



(c) Ratio of running times ($\frac{\texttt{SEPIA}}{\texttt{GMW}}$)

Figure 5: Running times in PlanetLab.

In Figures 3(b) and 3(c) (resp., Figures 4(b) and 4(c)) we compare GMW's running time (resp., bandwidth utilization) to that of VIFF and SEPIA. Since the running times of VIFF and SEPIA are comparatively long, we only ran experiments with up to 400 resources and up to 9 nodes. In those ranges of the parameters, GMW completes in under 20 seconds while VIFF and SEPIA take an order of magnitude longer; the relative performance of GMW becomes even better as the number of resources is increased. The results demonstrate that our implementation scales significantly better. It is also worth recalling that GMW withstands a larger number of corruptions than either VIFF or SEPIA.

5.2 Wide-Area Network

Last, we explore the effects of communication latency by running our protocol in a wide-area network (WAN) via PlanetLab (`http://www.planet-lab.org/`). In the PlanetLab settings we explored, the maximum round trip time (RTT) was more than 200 ms. The test nodes in PlanetLab have various hardware specs; the least powerful node had two Intel Core2Duo 2.33GHz CPUs and 2.0GB memory while the most powerful one had four Intel Xeon 2.83GHz CPUs and 3.7GB memory.

Figure 5 shows that GMW's running time increases by 17–64% relative to the time required on a LAN. As we increase the number of participating nodes, the results increase linearly as in the LAN, even though nodes' configurations are not homogeneous, suggesting that performance is mostly affected by communication latency. GMW maintains stable performance regardless of network conditions and heterogeneous hardware specifications, consistently outperforming VIFF and SEPIA indicated in Figures 5(b) and 5(c).

6 Conclusions

We have shown an implementation of the GMW protocol for secure multi-party computation. Our implementation is distinguished from existing implementations of multi-party computation in two important ways:

- Our implementation supports boolean circuits, rather than arithmetic circuits as in [10, 7].
- The protocol provides security against a semi-honest adversary corrupting any number of parties, rather than requiring an honest majority as in [3, 10, 7].

We have also shown that our implementation outperforms previous work [10, 7], at least for certain classes of problems that are more amenable to being solved using boolean circuits rather than arithmetic circuits. Finally, our work shows that applying secure multi-party techniques to networking problems is feasible.

Acknowledgments

The contents of this paper do not necessarily reflect the position or the policy of the US Government, and no official endorsement should be inferred.

References

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the clouds: A Berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, UC Berkeley, 2009.

[2] D. Beaver. Precomputing oblivious transfer. In *Advances in Cryptology — Crypto '95*, volume 963 of *LNCS*, pages 97–109. Springer, 1995.

[3] A. Ben-David, N. Nisan, and B. Pinkas. FairplayMP: A system for secure multi-party computation. In *15th ACM Conf. on Computer and Communications Security*, pages 257–266. ACM Press, 2008.

[4] D. Bogdanov, S. Laur, and J. Willemson. Sharemind: A framework for fast privacy-preserving computations. In *13th European Symposium on Research in Computer Security (ESORICS)*, volume 5283 of *LNCS*, pages 192–206. Springer, 2008. See `http://sharemind.cyber.ee`.

[5] P. Bogetoft, D. Christensen, I. Damgård, M. Geisler, T. Jakobsen, M. Krøigaard, J. Nielsen, J. Nielsen, K. Nielsen, J. Pagter, M. Schwartzbach, and T. Toft. Secure multiparty computation goes live. In *Financial Cryptography and Data Security*, volume 5628 of *LNCS*, pages 325–343. Springer, 2009.

[6] P. Bogetoft, I. Damgård, T. Jakobsen, K. Nielsen, J. Pagter, and T. Toft. A practical implementation of secure auctions based on multiparty integer computation. In *Financial Cryptography and Data Security*, volume 4107 of *LNCS*, pages 142–147. Springer, 2006.

[7] M. Burkhart, M. Strasser, D. Many, and X. Dimitropoulos. SEPIA: Privacy-preserving aggregation of multi-domain network events and statistics. In *19th USENIX Security Symposium*, pages 223–240. USENIX Association, 2010.

[8] R. Buyya, D. Abramson, and S. Venugopal. The grid economy. *Proc. IEEE*, 93(3):698–714, 2005.

[9] Y. Chen, R. H. Katz, Y. H. Katz, and J. D. Kubiatowicz. Dynamic replica placement for scalable content delivery. In *IPTPS*, pages 306–318, 2002.

[10] I. Damgård, M. Geisler, M. Krøigaard, and J. Nielsen. Asynchronous multiparty computation: Theory and implementation. In *12th Intl. Conference on Theory and Practice of Public Key Cryptography — PKC 2009*, volume 5443 of *LNCS*, pages 160–179. Springer, 2009. See `http://viff.dk`.

[11] O. Goldreich. *Foundations of Cryptography, vol. 2: Basic Applications.* Cambridge University Press, Cambridge, UK, 2004.

[12] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game, or a completeness theorem for protocols with honest majority. In *19th Annual ACM Symposium on Theory of Computing (STOC)*, pages 218–229. ACM Press, 1987.

[13] W. Henecka, S. Kögl, A.-R. Sadeghi, T. Schneider, and I. Wehrenberg. TASTY: Tool for automating secure two-party computations. In *17th ACM Conf. on Computer and Communications Security (CCCS)*, pages 451–462. ACM Press, 2010.

[14] Y. Huang, D. Evans, J. Katz, and L. Malka. Faster secure two-party computation using garbled circuits. In *20th USENIX Security Symposium* (to appear), 2011.

[15] Y. Ishai, J. Kilian, K. Nissim, and E. Petrank. Extending oblivious transfers efficiently. In *Advances in Cryptology — Crypto 2003*, volume 2729 of *LNCS*, pages 145–161. Springer, 2003.

[16] T. Jakobsen, M. Makkes, and J. Nielsen. Efficient implementation of the Orlandi protocol. In *8th Intl. Conference on Applied Cryptography and Network Security (ACNS)*, volume 6123 of *LNCS*, pages 255–272. Springer, 2010.

[17] J. Kangasharju, J. Roberts, F. T. R, K. W. Ross, S. Antipolis, and S. Antipolis. Object replication strategies in content distribution networks. In *Computer Communications*, pages 367–383, 2001.

[18] V. Kolesnikov, A.-R. Sadeghi, and T. Schneider. Improved garbled circuit building blocks and applications to auctions and computing minima. In *Cryptology and Network Security (CANS)*, volume 5888 of *LNCS*, pages 1–20. Springer, 2009.

[19] P. R. Lewis and P. Marrow. Evolutionary market agents for resource allocation in decentralised systems. In *Parallel Problem Solving From Nature*, pages 1071–1080, 2008.

[20] B. Li, H. Li, G. Xu, and H. Xu. Efficient reduction of 1-out-of-n oblivious transfers in random oracle model. Cryptology ePrint Archive, Report 2005/279, 2005.

[21] Y. Lindell, B. Pinkas, and N. Smart. Implementing two-party computation efficiently with security against malicious adversaries. In *6th Intl. Conf. on Security and Cryptography for Networks (SCN)*, volume 5229 of *LNCS*, pages 2–20. Springer, 2008.

[22] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay — a secure two-party computation system. In *13th USENIX Security Symposium*, pages 287–302. USENIX Association, 2004.

[23] M. Naor and B. Pinkas. Computationally secure oblivious transfer. *J. Cryptology*, 18(1):1–35, 2005.

[24] B. Pinkas, T. Schneider, N. Smart, and S. Williams. Secure two-party computation is practical. In *Advances in Cryptology — Asiacrypt 2009*, volume 5912 of *LNCS*, pages 250–267. Springer, 2009.

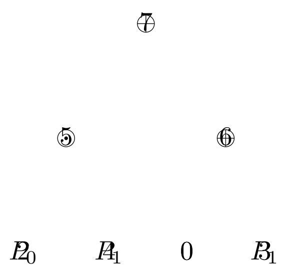
[25] B. Schnizler, D. Neumann, D. Veit, and C. Weinhardt. Trading grid services - a multi-attribute combinatorial approach. *European Journal of Operational Research*, 187(3):943–961, June 2008.

[26] Z. Tan and J. R. Gurd. Market-based grid resource allocation using a stable continuous double auction. In *GRID*, pages 283–290, 2007.

[27] R. Wolski, J. S. Plank, J. Brevik, and T. Bryan. Analyzing market-based resource allocation strategies for the computational grid. *International Journal of High Performance Computing Applications*, 15:258–281, 2001.

[28] A. C.-C. Yao. How to generate and exchange secrets. In *27th Annual Symp. on Foundations of Computer Science (FOCS)*, pages 162–167. IEEE, 1986.

A Circuit Example



```
n 2
d 7 5 2
i 0 2 2
i 1 3 4
o 0 1 0
o 1 7 7
v 0 1
v 1 1
g 0 0 -1 -1 0
g 1 0 -1 -1 0
g 2 0 -1 -1 1 5
g 3 0 -1 -1 1 6
g 4 0 -1 -1 1 5
g 5 1 2 4 1 7
g 6 2 0 3 1 7
g 7 2 5 6 0
```

Figure 6: Circuit Example

Our implementation of the GMW protocol takes as input (at each party running the protocol) three files that contain configuration information, the input of the party in question, and a description of a boolean circuit for the function f of interest. (All parties are assumed to be running with identical copies of the circuit.)

In Figure 6 we show an example circuit along with its description using our representation. The circuit description uses the following format:

- The first line of the file has the form `n` X, where X denotes the number of parties participating in the protocol.
- The second line of the file contains a `d` followed by several numbers providing an overall description of the circuit: the total number of wires in the circuit, the number w of the first non-input wire (i.e., wires 0 to $w-1$ are input wires), and the number of XOR gates in the circuit.
- For each party, there is a line in the file containing an `i` followed by the party's id, the number of the first input wire belonging to that party, and the number of the last input wire belonging to that party. (We assume wires are numbered such that every party provides inputs on a consecutive set of wires.)
- For each party, there is a line in the file containing an `o` followed by the party's id, the number of the first output wire belonging to that party, and the number of the last output wire belonging to that party. (We assume wires are numbered such that every party receives outputs on a consecutive set of wires.) If a party receives no output, the number of the last output wire for that party is set to 0.
- For each party, there is a line in the file containing a `v` followed by an integer denoting the number of bits that should be used to represent each item in that party's input file. (E.g., if the input file of party 0 contains a '4' then this value will be represented as the 3-bit integer '100' if this line of the file contains '`v` 0 3', but will be represented as the 5-bit integer '00100' if this line of the file contains '`v` 0 5'.) Each bit in the ultimate representation of the integer will correspond to one of the input wires of the party.
- The remaining lines of the file describe the gates in the circuit. For each gate, we list (a) the number of the output wire of this gate (which also serves as the gate id); (b) the gate type, which can be either input (0), AND (1), or XOR (2); (c) the numbers of the left and right input wires (set to -1 if these are input gates); and (d) the out-degree of the gate. If the out-degree is non-zero, then the ids of the gates that receive the output of the current gate are listed. Gate ids 0 and 1 are reserved for the constants 0 and 1, respectively.