Extension of Two-Party Computation to Multi-party Computation

Principles and Applications of Secure Multi-Party Computation

Philipp Müller

Fakultät für Informatik,
Technische Universität München
muellerp@in.tum.de

Abstract. We explain how n parties can evaluate a function $f(x_1, \ldots, x_n)$ such that each party privately holds one input parameter to the function. The function is evaluated in a manner that ensures that the privacy of each parties input is preserved and only a constant number of rounds is needed.

1 Introduction

Collaborative secure function evaluation has surprisingly many applications in practice. Whenever several participants want to compute some value from their respective inputs and privacy is a concern, we might need a technique that allows us to evaluate the proper function in a way that preserves privacy of the inputs.

1.1 Two motivational problems

We will now see two examples of what such problems might look like.

Secure auctioning Imagine the following scenario: Two people (A and B) want to buy some good C. Of course, the better offer will win, so both A and B are interested in who had the higher bid. However, A and B don't want each other to know the bids, i.e. A wants to keep its bid secret from B and vice versa.

We can model this problem as a function

$$f(a,b) = \begin{cases} -1, & \text{if } a > b \\ 0, & \text{if } a = b \\ 1, & \text{if } a < b \end{cases} . \tag{1}$$

Staying within this context, we want to know the function value $f(a,b)$, where a and b denote the bids of A and B, respectively, but A and B don't want a and b to become publicly known.

Of course, we might not only be interested in a function $f : \mathbb{R}^2 \to \{-1, 0, 1\}$, but in a function $g : \mathbb{R}^n \to \{1, 2, \ldots, n\}$ that can be used if not only two, but n people $(A_1, \ldots, A_n)$ are interested in some good C. The output of function g then denotes which person's offer was the best.

Office assignment problem This problem is taken from [1]. Imagine $2n$ people that must be distributed onto n offices, i.e. one office is shared by two people. Each person i writes a list with values ${}^iL^j \in [0,1]$ (for $j = 1 : n$). The value ${}^iL^j$ denotes how much person i likes person j. The higher the value the better.

Of course, they want to be distributed in such a way that their preferences are respected to a certain degree. They agree that for a room assignment M the value

$$\sum_{\{i,j\}\in M} \min({}^iL^j, {}^jL^i)$$

is a good measure for the goodness of the assignment M. So, the goal will be to maximize this measure, but – of course – no student wants to reveal his preferences (i.e. all the values ${}^iL^j$ shall stay private to person i).

1.2 Common ingredients of these two problems

The above two functions are both derived from the desire to compute some value from inputs that were contributed by different parties. Moreover, we – in particular – should look out that these inputs shall not become public.

Each participant wants to know the output, but does not want his input to the function to become public. Along the same lines, no participant shall gather information about the other participant's inputs.

At this point it is worth noting that even the function value itself *might* reveal some information about the inputs. For example, consider $mult(a, b) = a \cdot b$: If $mult(a, b) = 0$, one can conclude that one of the arguments must have been zero. Such conclusions – obviously – cannot be avoided since we are interested in the actual output of the function, that in this case might reveal information about the inputs. However, we want to keep "as much as possible" secret. To define what this means precisely is a crucial point that will be addressed later.

This kind of problem can be solved using *secure multi-party function evaluation.* Of course one might tailor a specific scheme for each function. However, it is conceivably to derive one technique that allows for arbitrary functions.

1.3 What we will construct

We will construct a framework that enables some participants (also called *players* or *parties*) to collaboratively evaluate a function such that each participant supplies one input to the function. Moreover, we will address the problem that the inputs shall not be revealed to the community.

Such a protocol for secure function evaluation usually evolves in *rounds.* In each round, the participants can communicate with each other. Between the rounds, they can carry out local computations.

One might be tempted to guess that "more complicated" functions take more time to be collaboratively evaluated. However, as we will see, this is not the case. The protocol we describe takes a constant number of rounds (independent of the function) and keeps the amount of communication between the single participants polynomially bounded.

2 Formal definitions for Secure Computation

We said that we want to compute functions in a way such that the inputs stay private to the respective players. We call that *secure*. While this colloquial explanation is quite intuitive, we will briefly examine what it means formally. The following definitions are along the lines of those given in [1] and [2].

2.1 Prerequisites

A word on the security parameter. A central value when considering cryptographic schemes or protocols is the so called *security parameter*, often denoted by k. In short, this is a constant that is fixed before setting up some cryptographic algorithm or protocol[1]. The run times of encryption/decryption or the running times needed by some adversary are then typically measured as a function in k.

Informally stated, typically the following is the case: As soon as k is greater than some particular value, the system constructed (according to this parameter) is considered secure (see [3]).

Definition 1 (Negligibility). *A function* $\epsilon : \mathbb{N} \to \mathbb{R}$ *is* negligible *if* $\epsilon(k) \in k^{-\omega(1)}$.

To bring the notion of negligibility in a format that is (due to Landau-Notation) possibly more well-known to most computer scientists, we can say, that a function $\epsilon(k)$ is negligible, if for every positive integer c there exists some k_0 such that for all $k > k_0$ we have $|\epsilon(k)| < k^{-c}$ (so: ϵ negligible $\Leftrightarrow \forall c > 0.\ \exists k_0.\ \forall k > k_0.\ \epsilon(k) < k^{-c}$, see [4]). That is, a function $\epsilon(k)$ is negligible if it vanishes faster than any polynomial-inverse.

A typical example for a function that vanishes faster than any polynomial-inverse is $f(k) = 2^{-k}$. This is a function that grows inverse-exponentially.

As you can see, we already specified ϵ as a function of k (which stands for the security parameter). This is because negligibility is a tool that is often applied to e.g. the success probability of an adversary's attack.

Alphabets and strings. Let us *informally* introduce alphabets and strings, since they are – at least at the level of definition – quite intuitive. We call some set Σ of letters an *alphabet*. If we take several letters from this alphabet and concatenate them, we obtain a *string* over the alphabet Σ. If there's no risk of confusion (which is almost always the case) we omit the phrase "over the alphabet Σ" and just call it "string". Moreover, we use no special sign for concatenation, but just write one letter after another.

There is exactly one string consisting of no letters at all (the empty string). By Σ^l we denote the set of all strings of length exactly l. We define the set of *all* strings over some alphabet Σ by

$$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \cdots = \bigcup_{i=0}^{\infty} \Sigma^i.$$

For our needs, we will consider $\Sigma = \{0, 1\}$, which is not a real restriction, because everything can be represented using only bytes (without excessively expanding the representation).

We call every subset of Σ^* a (formal) *language*.

Probability measures over languages. In order to understand the next definition, we have to think about the following random experiment: Take Σ^* and pick some string randomly from it. The so-called *probability measure* then "tells us" for each string how probable it is to pick exactly this string. That is, it maps each string (and – by extending it – every set of strings) to some probability between 0 and 1.

Definition 2 (Ensemble). *We call a family of probability measures* $\{\mathcal{A}_k\}$ *on* Σ^* *an* ensemble*, if only strings of length of at most* $q(k)$ *have positive probability with respect to* $\mathcal{A}_k$*, where* $q(k)$ *is some polynomial in* k.

[1] That is, you pick some security parameter, and afterwards the cryptosystem is set up according to this parameter.

The notation $\{\mathcal{A}_k\}$ should be understood as the set

$$\{\mathcal{A}_k \mid \mathcal{A}_k \text{ is a probability measure on } \Sigma^*, k \in \{0, 1, 2, 3, \dots, \}\}\,.$$

So, a probability measure in an *ensemble* "operates" only on a *finite subset S* of Σ^* (since it only "maps" strings of length of finite length onto non-negative probabilities), and "randomly picks" a string from this finite set S according to its distribution.

Definition 3 (Indistinguishability). *For $\mathcal{A}$ taken from some ensemble and C a boolean circuit (having sufficiently many inputs), $p_{\mathcal{A}}^{C}$ denotes the probability that C outputs 1 on an input randomly drawn according to the distribution given by $\mathcal{A}$.*

We say that ensembles $\{\mathcal{A}_k\}$ and $\{\mathcal{B}_k\}$ are computationally-indistinguishable *if for any polynomial-size circuit family $\mathcal{C} = \{C_k\}$, the function $\epsilon(k) := |p_{\mathcal{A}_k}^{C_k} - p_{\mathcal{B}_k}^{C_k}|$ is negligible.*

We call $\mathcal{A}$ and $\mathcal{B}$ statistically indistinguishable *if the function*

$$\epsilon(k) := \max_{S_k \subseteq \Sigma^*} |\Pr_{\mathcal{A}_k}[S_k] - \Pr_{\mathcal{B}_k}[S_k]|$$

is negligible, where $\Pr_{\mathcal{A}}[S]$ denotes the probability that we get a string within S if we draw from Σ^* randomly according to the distribution $\mathcal{A}$.

Essentially, if two ensembles $\mathcal{A}$ and $\mathcal{B}$ are *computationally* indistinguishable, then no efficient (i.e. no polynomial-time) algorithm can decide whether a sample given to it came from $\mathcal{A}$ or $\mathcal{B}$. In other words, any efficient algorithm's behaviour doesn't change significantly when given samples from $\mathcal{A}$ or $\mathcal{B}$, respectively. Note that statistical indistinguishability is a stronger property than computational indistinguishability in the sense that statistical indistinguishability implies computational indistinguishability.

For more information on indistinguishability, please refer to e.g. [5].

For us, indistinguishability is "just a tool" that helps us analyzing whether some malicious agent who is attacking our protocol can distinguish the protocols outcome from some sample obtained by some randomized algorithm. We will come to that later in more detail.

2.2 Model of computation

We assume a synchronous model. In this setting, the participants can either broadcast a message to everybody or use a private channel with one specified receiver. We imagine this as a network of processors whose computation is controlled by a common clock ticking (at time steps $0, 1, 2, \dots$). We assume local computation to be "instantaneous" compared to the clock ticks, i.e. whenever some participant computes something *locally*, we do not consider the exact running time of this operation, but we just assume that particular computation to take exactly the time between two rounds. Round i starts at tick i and lasts until tick $i + 1$.

Communication between participants. We assume a read-only common input tape for all participants that contains (among others) the security parameter k (in suitably encoded format). We assign each player a private work tape (read/write), a private read-only input tape and a private write-only output-tape, all of which are initially empty.

Moreover, we assume that between any two participants i and j there is a private channel $i \to j$ to securely send messages from i to j. This can be seen as a write-only tape for i and a read-only tape for j. These private channels are organized in a way such that the receiver can determine who sent the corresponding message.

Last, each player has a broadcast channel (write-only for the sending participant, readable by everybody).

It is convenient to assume that at each time step, there is a well-defined (possibly empty) message on each channel (i.e. in particular on broadcast and private channels).

Additionally, each player has access to a fair coin to support non-determinism.

Player i runs program P_i, so that we call the tuple $\mathcal{P} = (P_1, \ldots, P_n)$ a *protocol*. The *history* of player i consists of everything player i had access to (coin tosses, messages sent and received, private and common input).

Round and communication complexity. We stated that a protocol proceeds in rounds, and in each round participants might (or might not) communicate with each other. "In between two rounds", the participants have the possibility to do local computations. For collaborative function evaluation, the number of rounds together with the amount of communication is the decisive point. That is, when we consider a protocol's complexity, we have to distinguish between *round complexity* (i.e. the number of rounds needed by the protocol) and *communication complexity* (i.e. the amount or length of messages exchanged by the individual participants). Note that the number of rounds is usually the crucial size we try to keep low.

2.3 Adversaries

Power of adversaries. In particular, we are interested in what happens if not all players are following the protocol. We will have to explain how to proceed if some of the players are corrupted by an adversary (e.g. some malicious agent).

Therefore, we have to define what the participants of our protocol can do (and what they can't), and what a so-called *adversary* might do in order to compromise our privacy or disrupt correctness of our protocol. On the other hand, we – of course – need to describe how to tackle such adversaries.

In our case, we assume quite strong adversaries, that can "infect" players so that the adversary knows the player's entire "state" and controls all future actions of the infected players.

Moreover, we will consider adversaries that can infect players not only at some statically defined point of the protocol, but can act somewhat *dynamically.* The type of adversary we are considering is called a *uniform* adversary. That means, an adversary A is a probabilistic polynomial-time algorithm. An adversary infects or "corrupts" players. We consider adversaries that can do so at the beginning of each new round. Note that such an adversary can – after he has infected one player – decide whether to infect another or not.

However, we apply one restriction: The number of players an adversary can corrupt is bounded by some value. We call this value t. An adversary is called t-adversary if he corrupts at most t players.

After an adversary has infected a participant, it controls the player fully.

Note that we won't consider many distinct adversaries "within" one protocol, but we assume only *one* single adversary attacking our protocol. That is no real restriction, since we can interpret several malicious agents as "belonging" to one malicious instance only.

What does it mean to defeat an adversary? Imagine a black box that computes some function privately and securely and that is resistant against attacks from adversaries. We say that a protocol computes a function securely if it – under attack by the adversary – imitates this black box under attack as closely as possible. To formalize this in a mathematical way is a clearly non-trivial issue. We will address this later and see what it means in more detail. For now, we just say that an adversary should not be able to decide whether some information come from a randomized algorithm (a so-called *simulator*), or the information comes from a real protocol that he was trying to compromise.

We assume that the protocol ends as soon as all uncorrupted participants have terminated. Then the adversary terminates, too. At this time the tape of the adversary has some string on its output tape. If we consider the protocol $\mathcal{P}$, security parameter k and the private inputs $\overrightarrow{x} = (x_1, \ldots, x_n)$ as fixed instances, the adversary A defines a probability space $\mathbf{VIEW}_A^k(\overrightarrow{x})$ of adversary outputs. We can interpret the adversary's output as an encoding of its history (that is of coin flips, received and sent messages, etc.).

Moreover, each player has output a certain value. We indicate the output values of the uncompromised players by $\mathbf{OUTPUT}_A^k(\overrightarrow{x})$. When we are interested in a sample from this distribution, we just consider the outputs of non-infected players.

2.4 Security

As mentioned before, we want that "the adversary doesn't know whether what it sees comes from the protocol, or from some randomized algorithm (a so-called *simulator*)". That is, we want that the adversary's view can be *approximated* by a simulator. We ask what auxiliary tools this simulator can use to approximate the view. We will consider simulators that have access to a so-called *oracle*:

Definition 4 (t-bounded oracle). *A t-bounded $(\overrightarrow{x}, f)$-oracle is an oracle accepting two kinds of queries:*

Component query *A component query is an integer $i \in \{1, 2, \ldots, n\}$. It is only answered if t or fewer component queries were made so far. If this is the case, the oracle distinguishes between the following two cases:*
- *If there has been no output query so far, it is answered by x_i.*
- *If there has been already a proper output query (see below), namely $\overrightarrow{x}'_T$, the query is answered by $(x_i, f_i(\overrightarrow{x}_{\overline{T}} \cup \overrightarrow{x}'_T))$.*

Output query *An output query is a "tagged vector" $\overrightarrow{x}'_T$ (see below). It is answered by $f_T(\overrightarrow{x}_{\overline{T}} \cup \overrightarrow{x}'_T)$ if T consists precisely of the component queries made up to now and if there were not output queries so far. Further or improper output queries stay unanswered.*

The first thing we have to clarify is the meaning of a tagged vector. Consider vectors $\overrightarrow{v} = (v_1, \ldots, v_n)$ and $\overrightarrow{w} = (w_1, \ldots, w_n)$ and some set $T \subseteq \{1, 2, \ldots, n\}$. Then, the T-tagged vector $\overrightarrow{v}_T$ is defined by $\{(i, v_i) \mid i \in T\}$. That is, a tagged vector is a set of pairs each containing an index and an actual element. Please note that if each index is present exactly once in some tagged vector, it can easily be seen as a "usual" vector (simply by writing the elements at the specific positions). This means, that (with $\overline{T} = \{1, 2, \ldots, n\} \setminus T$) we can interpret $\overrightarrow{v}_T \cup \overrightarrow{w}_{\overline{T}}$ as the vector whose components are taken from $\overrightarrow{v}$ if the component index is contained in T, otherwise from $\overrightarrow{w}$.

Note that there might be component queries after an output query, as long as the number of component queries does not exceed t.

As you can see, an oracle is parametrized with t, $\overrightarrow{x}$ and f, i.e. it can be specifically designed for the function f we are interested in and the corresponding input values (encoded in $\overrightarrow{x}$). Moreover, it essentially allows up to t component queries. Remember that we didn't allow an adversary infecting as many players as he'd like to, but we limited the number of infected players to t.

We consider simulators S having access to a t-bounded $(\overrightarrow{x}, f)$-oracle. We denote the output of simulator S by $\text{Output } S^{O_t(\overrightarrow{x}, f)}(1^k)$. Note that this – again – forms a probability space. That is, fixing k, f and $\overrightarrow{x}$, and "generating" some random result by applying S with these parameters, we get a value according to some random distribution.

By QUERIES $S^{O_t(\overrightarrow{x},f)}(1^k)$ we denote a pair containing the following two things:

- The indices i for which there was *never* a component query, i.e. it contains the components that were never queried by S.
- The (single) output query that was made by S.

Since S is a simulator (a randomized algorithm), it might randomly and dynamically decide which components to query and *when* to do an output query. Thus, QUERIES $S^{O_t(\overrightarrow{x},f)}(1^k)$ forms a probability space, as well.

An element from this probability space is written $(G, \overrightarrow{x}'_T)$, where G denotes the indices that were *never* queried by S and the $\overrightarrow{x}'_T$ is the output query (remember: there can only be *one* output query). As you can see, G may be a proper subset of $\overline{T}$.

You see, we defined several probability spaces for adversaries as well as for simulators. We shall now see why we did so and what we consider "secure":

Definition 5 (Privacy, Correctness). *Let $f : (\Sigma^l)^n \to (\Sigma^l)^n$. A protocol $\mathcal{P}$ t-securely computes f if for all t-adversaries A there exists a simulator S (probably using a t-bounded oracle) such that the following hold:*

Privacy *For all $\overrightarrow{x} \in (\Sigma^l)^n$, the k-parametrized ensemble $\mathbf{VIEW}_A^k(\overrightarrow{x})$ and the ensemble* OUTPUT $S^{O_t(\overrightarrow{x},f)}(1^k)$ *are computationally indistinguishable.*

Correctness *For all $\overrightarrow{x} \in (\Sigma^l)^n$, the k-parametrized ensembles $\mathbf{OUTPUT}_A^k(\overrightarrow{x})$ and $[(G, \overrightarrow{x}'_T) \leftarrow$ QUERIES $S^{O_t(\overrightarrow{x},f)}(1^k) \mid f_G(\overrightarrow{x}_{\overline{T}} \cup \overrightarrow{x}'_T)]$ are statistically indistinguishable.*

We see that we require (for privacy) that – intuitively – the outcome of an adversary's computation can be *simulated* by some random algorithm (that has access to some oracle). The thought behind is the following: "If the adversary's computation could have been done by a simulator (not even knowing the protocol), then the adversary's attack couldn't be that useful."

The above definitions state that for any adversary A there is some simulator S that fulfils the privacy and correctness criteria for this particular adversary. In other words, it is enough if there is (at least) one simulator for each adversary, and different simulators might be used for different adversaries. That is, from the definition, one might tailor the simulator to the specific given adversary.

However, the proof in [1] leads to something stronger, namely *one* single simulator that works for *all* adversaries (or at least for all the ones that we allowed).

2.5 What can we use as a basis

Remember that our goal was to derive a protocol that allows us to securely evaluate any given function.

As we will see, we won't construct this protocol "from scratch", but we will use other, already established techniques to solve the problem. These are nowadays well-known concepts, that we present here briefly.

Secret sharing. If we want to compute a function collaboratively, it is, of course, advantageous if we are able to distribute information across players. As said before, each player will hold some private input (which we call – for now – her *secret*).

Therefore, it is useful that a single player can "split up" her secret into small pieces that can be distributed among all players.

The key idea is that this secret is split up in a way such that – essentially – none of the players can reconstruct the secret on his own (i.e. from his single part only), and the secret can only be reconstructed if a certain number of players cooperate and reveal their parts of the secret.

Section A.2 in the appendix provides a description of how this works for the interested reader.

We call the player holding the secret the *dealer* and call the distributed parts *shares.*

Verifiable secret sharing. We call secret sharing *verifiable* if the dealer is not able to distribute incorrect shares to the other players and even some dishonest players are tolerated. More precisely, it is ensured that even if the dealer is malicious, the other players can later reconstruct some well-defined secret.

We consider a dealer D holding a secret bit b who is engaged in a verifiable secret sharing scheme tolerating up to t faults. After the scheme, each player i holds his own private share b_i of b. Then, we require three properties:

1. With high probability (i.e. greater than $1 - 2^{-k}$, where k being the security parameter), there exists a unique value b' such that if the good players broadcast their private shares, each good player will locally compute b' from the broadcast values, regardless of the values broadcast by malicious players.
2. If D is good, then the computed value b' will be the correct value b.
3. If D is good, then the view of any adversary not corrupting D is independent of b. That is, there is some simulator *independent* of b that can simulate the adversary's view.

If player D splits up his secret b and distributes it, we say that D *commits* to b. After D committed to b, b is said to be *committed* or a *shared* secret. One important aspect of committing is the following: if a dealer commits to a certain value, then this value is not automatically known publicly (it is just shared). But the dealer cannot change his mind afterwards; that is, once he commits to a value, he is "bound" to that value. For more information on this topic, we refer to section A.1 or [3].

Even if we defined verifiable secret sharing for single bits, we can easily extend it to strings, if we simply share each bit of the corresponding string.

For more information on verifiable secret sharing, please refer to e.g. [7].

It has been shown that – in our model of computation – verifiable secret sharing can be implemented in a constant number of rounds in the presence of a t-adversary for $t < n/2$. If you are interested in the proofs of these statements, please consult [8,9].

Computing on shared bits. As we will later see, circuits are a basic construct when we evaluate a function. It has been proven that a boolean circuit (representing some function) with bounded fan-in and depth d can be evaluated securely and secretly in $O(d)$ rounds involving an amount of communication that is polynomial in k and in the size of the circuit. That is, we can assume that the inputs to the circuit and the result of the evaluation of the circuit are shared bits (and thus not publicly known). If you are interested in the proofs of these statements, please consult [8,9,10].

The problem with these protocols is that they do not necessarily work in a constant number of rounds. Our protocol will overcome this weakness, but relies – in some intermediate steps – on those protocols.

We note some important examples that can be securely and secretly evaluated in a constant number of rounds tolerating less than half of the players being infected by an adversary:

Unbounded fan-in XOR. If one examines the protocols developed in e.g. [11,8], it can be shown that the output of a XOR-gate operating on any number of bits can be evaluated securely and secretly in a constant number of rounds with polynomial amount of communication.

Constant-depth circuits It has been shown by [8] that any constant-*depth* circuit with bounded fan-in can be securely and secretly evaluated in a constant number of rounds with polynomial amount of communication.

Collaborative coin flipping. Flipping coins collaboratively can be seen as a special case of the previous problem, with some added randomness. It allows a group of players to obtain shared random bits (that then are known to *nobody*!). They can be constructed if each player commits a random bit. Then, the participants could – if they (or at least some specified subset of them) revealed their shares – recompute each single bit committed by all the participant. For a brief introduction on how this works, see section A.3.

Proving assertions to the community. In our protocol, players will need to assert that some of their local computations were done right. Proving correctness can be done fast, even if the corresponding computation might have involved deep circuits.

Informally, if we have a function $f : \Sigma^a \to \Sigma^b$ represented by a circuit C, and shared input bits $\sigma_1, \ldots, \sigma_a$, then a player can commit $f(\sigma_1, \ldots, \sigma_a)$ to prove that he computed f on the given bits correctly. This proof information-theoretically reveals nothing and requires $O(1)$ rounds and a polynomial amount of communication (in $|C|$ and k).

When the player proves correctness of his computation, she commits (additionally to the function value) some certificate containing the internal values of the wires.

Pseudorandom generators. As we will see later, we will use so-called pseudorandom generators in the protocol. A pseudorandom generator is a *deterministic* algorithm running in polynomial time that takes a "seed" (some truly random string) as input and composes a longer, pseudorandom string from it ("stretching").

Thus, the following shall hold: the output of a pseudorandom generator cannot be distinguished (using an efficient algorithm) from a truly random source (in the formal definition, we use negligibility to state this).

Note that a pseudorandom generator has the property, that we cannot deduce its input from its output[2].

Even if it is not known for sure that such generators exist, it has already been proven that they at least exist under very convenient complexity-theoretic assumptions. For more information on this topic see section A.4 or [12,13].

3 The protocol on a very high level

In order to understand how the protocol works, we now consider the involved strategy on a very high level. Later, we will give a more formal version (i.e. essentially an algorithm) for the protocol.

3.1 General idea

First of all, we are – as alluded before – assuming the function to be on hand in a suitable and unified format, namely in shape of a circuit. That is, we represent

[2] Functions that have this property are also called *one-way functions*.

a function as a collection of gates and wires. The input wires to the circuit will correspond to the (private) inputs $x_1, \ldots, x_n$, while the output wires determine the result $f(x_1, \ldots, x_n)$ we want to compute. Note that not each x_i is represented by one wire only, since it might well be the case that one single x_i consists of several bits. That means, that each input bit is assigned to one wire.

It is worth noting that we can without loss of generality assume the circuit to consist only of gates having at most 2 input wires. We can reduce any circuit having gates with more than two input wires to such a circuit. This will increase the size of the circuit, but only by a polynomial factor (in the size of the original circuit). For a short explanation, please refer to section A.5. This will simplify our construction. Moreover, we assume that the gates used in that circuit have a special format: Either they have only one output wire, or they have two output wires, but then only one input wire (we will later discuss why this is necessary). Last, we will assume no "cycles" in our circuit[3].

At the beginning, everybody knows the circuit, since the function we want to compute is known to all participants.

The first thing we are going to do is constructing a so-called *garbled circuit*. This is a circuit that resembles a computation which is related to the original function f in such a way that one can – using certain substitutions – compute the outcome of f without actually knowing the input (and intermediate) values to f. To be a bit more concrete, the garbled circuit will not receive the original inputs, but will receive inputs whose *meanings* (or *semantics*) are (mostly) unknown to the computing parties. That is, a participant can actually compute the correct output of the circuit, even though she does not learn the intermediate values. While this might sound strange to the reader, it is really just the idea at a very high level. We will shortly come to the details of this construction.

As we will see, we can use a trick for each gate that enables us to compute the output of the gate (whose semantics will also be unknown). In this way, we can propagate the computation up to the final gate. This will be the only gate whose outputs can be easily associated with their semantics.

In this way, we will be able to finally compute the final gate's outputs and semantics (and hence the function value), while keeping the input and intermediate values secret from anybody else.

Let us briefly summarize the idea as follows: we will construct a scheme that *hides* the actual bits within the circuits. The participants will still use some intermediate signals during the computation, but they will *not know* if a particular intermediate signal represents a bit 1 or a bit 0.

Let us for now consider a concrete example of how we can construct a garbled circuit from an original circuit. Later, we will define the procedure more formally.

3.2 Concretizing the idea – an example

Our example function and its associated circuit. To illustrate the transformation of a circuit into a suitable garbled circuit, we present an example which is taken from [1]. Consider the function

$$f : \begin{cases} \{0,1\}^3 \to \{0,1\} \\ (x,y,z) \mapsto (x \wedge y) \vee z \end{cases}$$

and imagine we have three participants (let's call them X, Y and Z for the corresponding inputs) that want to evaluate f for their respective inputs.

[3] That makes sense when we are representing functions – in the mathematical sense – by a circuit.

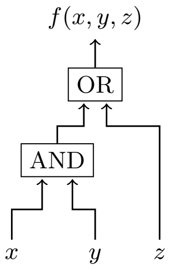


Fig. 1. Circuit computing $f(x,y,z) = (x \wedge y) \vee z$. Note that this (simple) circuit satisfies the requirements we proposed: No circuits, only gates having two inputs and exactly one output.

The function f can – and will – be represented straightforwardly by the circuit C shown in Figure 1. This circuit simply consists of two gates (and some wires connecting these gates properly), one of which is computing the AND of x and y, the other one taking the OR of z and the result given by the AND-gate. This circuit is deliberately simply for illustration purposes, but the concepts extend to more complicated circuits[4].

Remember now that we want to compute $f(x,y,z)$ in a way such that no participant can infer more information about the inputs than the result $f(x,y,z)$ itself implicitly reveals. Since we want *no* participant to gather information about the inputs of the other two players, it is not a good idea to leave the whole computation to one participant alone. Thus, we will compute the function *collaboratively* and thereby take care that no participant has to reveal his input to some other participant.

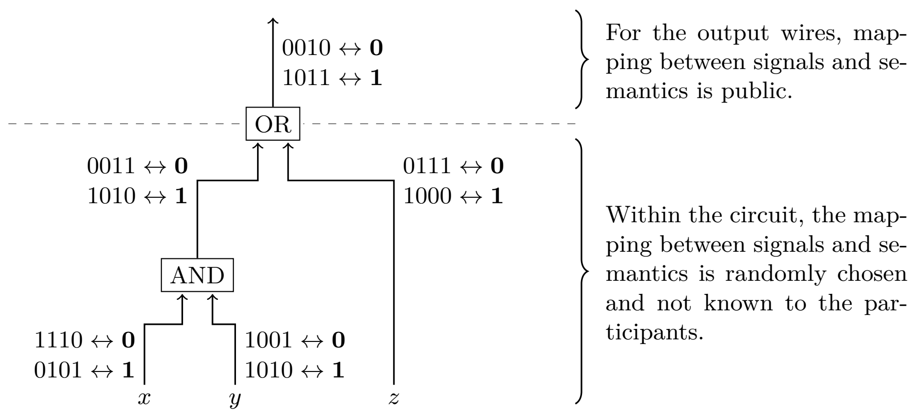
However, if we operate *directly* on the given circuit, our participants would have to supply their inputs without obfuscation (or something similar) so that the participants could see each others' inputs. That's why we resort to a *garbled circuit.*

The idea of a garbled circuit. As said above, to compute $f(x,y,z)$, we first construct a "garbled circuit". Note that when we evaluate the circuit C (the one depicted in Figure 1) for some inputs x, y and z, each wire has a certain value (0 or 1) that we call the wire's *semantics.* These semantics are represented by a special *signal* (the semantic **0** is represented by the signal 0, while semantic **1** is represented by signal 1). Once we supply the inputs to circuit C, all its internal semantics are known (or can be computed easily).

It is important that we can differentiate between the *semantics* and the *signal* of a wire. In "normal life" (i.e. when we evaluate circuits without restrictions on privacy of inputs), we will – for simplicity and convenience – associate signals that represent the semantics in some straight-forward way (e.g. signal 0 represents semantic **0**). But as we are especially interested in keeping information private, it should be clear that this might exactly be something that we do *not* want here!

That is, we could camouflage the semantics by using signals that cannot easily be mapped onto semantics. Therefore, we might even choose something different than "single letters" as signals, but whole strings. When doing so, we want to keep the mapping from signals to semantics (and vice versa) secret. That is, we allow (possibly more complicated) signals across the wires, and associate certain signals with semantics 0, and other signals with semantics 1. However, it shall – essentially – *not* be possible that any participant can deduce the semantics from

[4] As long as they have a certain structure (see 4.1)



0010 ↔ **0**
1011 ↔ **1**
OR
0011 ↔ **0**
1010 ↔ **1**
0111 ↔ **0**
1000 ↔ **1**
AND
1110 ↔ **0**
0101 ↔ **1**
1001 ↔ **0**
1010 ↔ **1**
x
y
z
For the output wires, mapping between signals and semantics is public.
Within the circuit, the mapping between signals and semantics is randomly chosen and not known to the participants.

Fig. 2. Associating random signals with semantics. The input wire for variable y associates signal 1001 with semantics **0** and signal 1010 with semantics **1**. While all input and internal signals and semantics cannot be mapped onto each other, the signals and semantics of the *output* wire are constructed in such a way that the signal's last bit coincides with the desired semantics. Note that the "structure" (i.e. placement of gates and wires) of the garbled circuit corresponds to that of the original circuit depicted in Figure 1.

the signals (except for the output wire, where we particularly want to be able to retrieve the semantics, since it represents the function value). On the other hand, it *shall* be possible, that the participants can correctly compute the signals needed for evaluation — despite not knowing the semantics.

Even if this looks like an impossible task, we remember that we proposed several auxiliary techniques in section 2.5 that *might* help us to achieve or goal. We will later see how they concretely do so.

Signals and semantics by example. We said that we map signals to semantics (and implicitly vice versa). Now, for example, a wire ω might "hold" signal 010001001 (note that this signal consists of several bits, i.e. a whole bit string), and we map this signal (secretly) to 0. This means, if we see that wire ω holds this signal, this means that it carries semantics 0 (but we possibly don't know this — we just know the *signal*).

Note that the same signal (010001001) might be mapped to 1 on another wire ω', i.e. we map signals to semantics for each wire[5]. The idea is now, that for all wires but the output wire, we don't publish the mappings between signals. Only the output wire's[6] mappings between signals and semantics are known, since we want to be able to compute the output.

We come now back to our circuit C and "install the new signals": Therefore, we "remove the original signals", and replace them by new (random) signals that are mapped onto semantics. We illustrate this concept in Figure 2. In our example, we see, that if the left incoming wire of the AND-gate holds signal 0101 (representing semantics **1**), and if the right incoming wire of the AND-gate holds 1001 (representing semantics **0**), we should get the output signal 0011 (representing semantics $\mathbf{0} = \mathbf{1} \wedge \mathbf{0}$).

[5] This might be needed if we have more wires than signals available.

[6] Here we are talking about the output wire of the whole circuit (and not of one single gate).

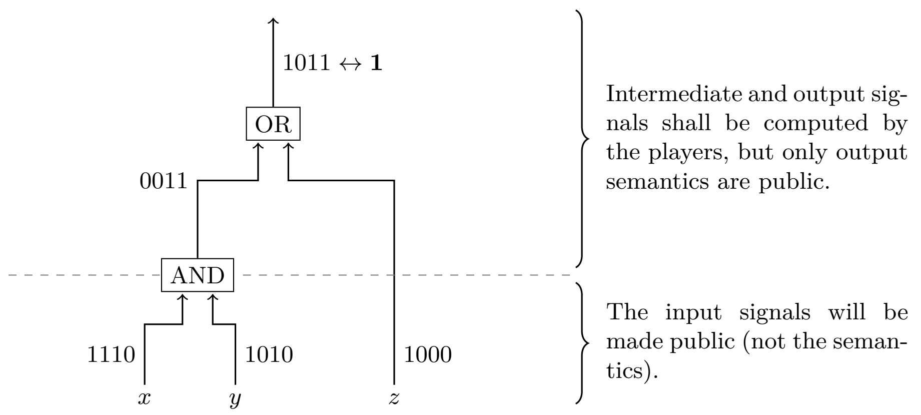

1011 ↔ **1**
OR
0011
AND
1110
1010
1000
x
y
z
Intermediate and output signals shall be computed by the players, but only output semantics are public.
The input signals will be made public (not the semantics).

Fig. 3. Our goal: The players shall – given a garbled circuit and garbled inputs – compute intermediate and output signals. From the output signals, they can deduce the output of the function. This example shows the garbled inputs for $x = \mathbf{0}, y = \mathbf{1}, z = \mathbf{1}$ (see figure 2 for reference). The players shall not be able to deduce the other signals for the wires. Since $f(0,1,1) = (0 \wedge 1) \vee 1 = 1$, each player shall – at last – learn the output signal 1011 which stands for semantics **1**.

Only the output wire's association between semantics and signals (i.e. 0010 ↔ **0** and 1011 ↔ **1**) is publicly known. They are constructed such that the last digit of the signal corresponds to the desired semantics. This is useful since we want all participants to simply be able to deduce the functions outcome from the output wires of the circuit.

All *internal* mappings from signals to semantics, are unknown to the participants. Please note the difference between the fact that the relation between internal signals and semantics is properly *defined*, and the fact that they are *not known* to the players. This is a central point for the protocol to work.

In a nutshell: What is our goal? We later want something like the following: Each player shall get the garbled circuit, and *only* the input signals (also in garbled format). From these, each player shall be able to compute the output values of the circuit. This is depicted in figure 3. On the way to this, the player shall not be able to deduce the semantics of input or intermediate wires[7].

Moreover, if a player has computed e.g. the even signal for a wire, she should not be able to deduce the odd signal for that wire.

Format of the signals. Note that all signals have the same length and are binary strings. Moreover, it is worth noting that for each wire, exactly one signal is ending in 1, while the other is ending in 0. We call the signals ending in 0 "even signals" and the signals ending in 1 "odd signals" (for obvious reasons). Even if we don't strictly need this convention, it is quite helpful and makes notational issues a bit easier.

Since we want to collaboratively compute the circuit, it should not be the case that e.g. one single player has to define all the signals, and possibly declare his defined signal public. We rather choose some method where all signals are composed of parts supplied by *all* participating parties. To be precise, we require that each

[7] If some player supplies some input, she of course knows the semantics of the corresponding input wire. We just want to ensure that no player learns the semantics of inputs that are not supplied by herself.

party contributes k bits to each signal (k denoting the security parameter). Moreover, we want to construct the signals randomly, since we wanted that concluding the semantics from a signal is not possible (for internal and input wires).

If each of the n players contributes k bits, these are nk bits. To these bits, we add one single bit (again collaboratively randomly chosen) indicating whether it is the even or odd signal (the so-called *parity*). These, in turn, makes signals of length $nk + 1$.

Having this construction in mind, we can interpret a signal (i.e. a bit string) of length $nk + 1$ as a concatenation of n bit strings each having length k and one last bit to determine whether the signal is odd or even.

So, the only remaining (but very central) question is: “How can we assure that the intermediate values (i.e. the internal signals) are computed correctly, even if we don’t know the association between signals and semantics?”

How can signals propagate along the gates? In order to compute the final output value, we will have to determine the output wire’s signal (and – since for this wire we know the mapping – its semantics). Therefore, we have to compute internal signals along the wires.

To do this, we employ an auxiliary table for each single gate that helps us computing the gate’s functionality. We denote the gate’s functionality by the symbol $\otimes$ (i.e. the symbol $\otimes$ represents e.g. OR, AND, XOR, NAND or another binary function on bits). This auxiliary table holds four (binary) strings, each of which has length equal to the length of the signals. We construct these bit strings for each gate g and call them call them $A^g_{00}, A^g_{01}, A^g_{10}$ and A^g_{11} (we use superscript g to specify the gates for which the strings are used).

So, we get a table consisting of 4 rows for a gate taking two inputs and having one output. Remember that we assumed our circuit consisting only of gates with two inputs. Since each input wire can carry either an odd or an even signal, that makes 4 possibilities for the input semantics in total.

As you might guess, that every single entry of these 4 entries in the table is used for one specific computation of the gate, i.e. for one specific combination of odd and even signals along the input wires. For example, if the two input wires of gate g are even, we have to use A^g_{00}. If the left incoming wire of gate g carries an odd signal, and the right incoming of gate g carries an even signal, we have to use A^g_{01} for the computation. That is, one has to consider the last bits of the signals carried by the two incoming wires, and choose the value from the table accordingly.

But what do the values of this table ($A^g_{00}, A^g_{01}, A^g_{10}$ and A^g_{11}) look like? Note that we said that the length of a signal is $nk + 1$. In order to understand how the table works, we interpret a single signal as n strings, each of length k, i.e. we split up the signal into several pieces. Note that, since the length of a signal was $nk + 1$, we discard the last bit when doing so.

As we mentioned before, we consider one signal (of length $nk + 1$) as n binary strings of length k and one additional parity bit (telling us whether the signal is odd or even). We now split up an incoming signal into n strings of length k and issue each of these strings as input to a pseudorandom generator G. This pseudorandom generator (taking a string of length k) stretches this input into a longer (binary) string (of length equal twice the length of one single garbled signal). This longer string is then split up into two halves that we call $G_0(s)$ and $G_1(s)$ (for the “zeroth” and the “first” half of the generator’s output).

Depending on the parity on the incoming signal, we either use G_0 or G_1. If the incoming signal is even, we use G_0, while we use G_1 if the incoming signal is odd. That is, we split up the signal into n bit strings, feed each of them into a random generator and use the respective halves of the outputs generated by the calls to our random generator G. We will come back to that later in more detail (i.e. we will explain how long the output of G is exactly, and which pieces exactly are given to G as inputs).

That essentially means: For a given even signal (i.e. a signal ending in 0), we compute $G_0(s)$ for all pieces s obtained by splitting up the signal. We proceed analogue for odd signals.

We take then the bit-wise exclusive-or (XOR) of all the values obtained by the pseudorandom generators and the appropriate table entry.

So, let us for now assume that we are considering – for a certain gate – the following input signals (for left and right input wire, respectively):

$$\sigma = \sigma_1 \dots \sigma_n a \quad \text{and} \quad \tau = \tau_1 \dots \tau_n b.$$

That is, each σ_i is a bit string of length k and the signal σ has parity a. Analogously the input signal τ consists of n parts having length k each (the single τ_i). Then, we later want to be able to compute the gate's output as follows:

$$\mathit{output} \leftarrow G_b(\sigma_1) \oplus \dots \oplus G_b(\sigma_n) \;\oplus\; G_a(\tau_1) \oplus \dots \oplus G_a(\tau_n) \;\oplus\; A^g_{ab} \tag{2}$$

You can treat the arrow-sign ($\leftarrow$) as an equality sign. We just chose this symbol here, because we wanted to emphasize that the output *is computed* from some other values. That is, we have to consider all input signals (i.e. all combinations of odd/even signals for this gate) and compute the corresponding table entry A^g_{ab}

Let us now think for a moment, why we exactly want to define the output like this:

In the above equation, *output* denotes the proper desired result for the specific gate. It should not be overly difficult to solve the above equation (resp. definition) for A^g_{ab} (remark: a and b were the parities of the input signals σ and τ, respectively).

If we do so, we can derive how to construct the values $A^g_{00}, A^g_{01}, A^g_{10}$ and A^g_{11} (by substituting a and b by appropriate values). We now solve equation (2) for A^g_{ab}:

$$A^g_{ab} = G_b(\sigma_1) \oplus \dots \oplus G_b(\sigma_n) \;\oplus\; G_a(\tau_1) \oplus \dots \oplus G_a(\tau_n) \;\oplus\; \mathit{output} \tag{3}$$

Even if this beast doesn't look that nice to us (and even if we didn't specify the term *output* exactly in the above equation), we already now see the following: Equation (3) consists essentially of terms obtained by a random generator (these are the terms $G_b(\sigma_i)$ and $G_a(\tau_j)$) and one term *output* that are XORed to obtain the result A_{ab}. Remember: In section 2.5 we talked about collaboratively evaluating the XOR-function. We said that we can do this in a constant number of rounds with polynomial amount communication. That means: If we know how to compute the term *output*, we can evaluate this huge XOR in a constant number of rounds with only communication only.

What you should recognize is the following: Equation (3) seems to contain no parametrization that "adjusts" the value A_{ab} to the kind of gate that the value will later be used for. This information is contained in the term *output*. Let us now see what this term is looks like exactly.

Remember that the term *output* should contain the result that one gets when evaluating the considered gate with inputs σ and τ, respectively. To be a bit more precise, *output* shall represent the signal that is "returned" by the gate for the given inputs.

Remember that we assigned signals *to each wire*, i.e. in particular to the output wire of the gate we're considering. Moreover, the left input wire holds the signal σ_a (i.e. a signal with parity a) and the right input wire holds the signal τ_b (i.e. a signal with parity b). As we stated before, for each wire, each of the two signals represents exactly one semantic (**0** or **1**). We now want the following: We want to determine the correct outgoing signal such that the *semantics* associated with the computed outgoing signal are the same as the *semantics* we would obtain if we would map σ_a and τ_b onto their respective semantics and evaluate the ungarbled (i.e. the original) gate for these input semantics. That is we essentially want to know: "What would the original gate return on inputs represented by the semantics of the input wires?" The obvious problem here is of course, that we do not know the mapping between signals and semantics, and we just have the signals.

Surprisingly, it is still possible to properly compute the outgoing signal such that its associated semantics represent the desired computation!

How to compute the value of the variable output. Before we go on, we need some additional definitions. First of all, let us give names to these wires: We call the input wires α and β (carrying signals σ and τ, respectively), and we call the outgoing wire γ (will then carry the signal we want to compute).

Let us call the even and odd signals for the gate's output wire s_0^γ and s_1^γ, respectively (remember: these are just signals, and we do not know the mapping to the associated semantics). That is, when we want to compute the gate's output, we have to decide whether we choose s_0^γ or s_1^γ as the resulting signal. So, essentially, we have to compute the subscript index.

Moreover, we will need the *semantics* for the incoming wires and outgoing wire.

We will now need a single bit for each of these three wires – for now called $\lambda_\alpha, \lambda_\beta, \lambda_\gamma \in \{0,1\}$ – that defines for the corresponding wire whether the even signal is associated with semantics $\mathbf{0}$ or vice versa. The subscript denotes the wire that is associated with that bit.

That is, if e.g. $\lambda_\alpha = 0$ (this bit is of course for wire α), we – on this wire – associate the even signal with semantics $\mathbf{0}$ and the odd signal with $\mathbf{1}$. If $\lambda_\alpha = 1$, it is just the other way around[8]. As you can see, one single bit suffices to fully define the semantics of a wire.

Note that using this bit, one can easily compute the semantics from a signal (for a specific wire). We simply take the last bit of the signal (the parity bit) and XOR it with the corresponding bit λ for this wire.

$$output := s^\gamma_{[(\lambda_\alpha \oplus a)\otimes(\lambda_\beta \oplus b)]\oplus\lambda_\gamma} \tag{4}$$

So, why does this make sense? Consider for now, just the subscript index

$$[(\lambda_\alpha \oplus a) \otimes (\lambda_\beta \oplus b)] \oplus \lambda_\gamma$$

and take it to pieces:

The term $(\lambda_\alpha \oplus a)$ is the *semantics* carried by the input wire α (we said that we can compute the semantics by taking the XOR of the λ-variable with appropriate index – in this case α – and the parity bit, which is – in this case – a). Analogously, $(\lambda_\beta \oplus b)$ is the semantics carried by input wire β.

Remember that we are considering one particular gate that computes the function denoted by the symbol $\otimes$. Thus, $(\lambda_\alpha \oplus a) \otimes (\lambda_\beta \oplus b)$ is exactly the semantics that we expect to obtain on wire γ when we would evaluate the original gate. Thus, if we want to have the associated signal to that semantics, we have to reconstruct it by XORing this value with λ_γ.

That is, if we define *output* this way, it gives us *exactly* the signal we need for the output wire.

Moreover note that we said that it is possible to evaluate a circuit with bounded fan in and depth d in $O(d)$ rounds with polynomial communication. Note that the

[8] The bit λ basically tells us how we can construct the semantics from the last bit of the outgoing signal.

circuit needed to compute this subscript has bounded fan-in and constant depth, thus can be computed in $O(1)$ rounds.

It remains to clear how the values $\lambda_\alpha, \lambda_\beta$ and λ_γ are constructed. Again, since we do not want any participant to compute something alone, we construct these values collaboratively. Therefore, the participants generate random bits and take the XOR of them. Again, this is a collaborative evaluation of the XOR function which – as we know – can be evaluated in a constant number of rounds with polynomial communication amount.

But when one considers equation (3), one might ask why we XOR this value with all the outputs generated by some pseudorandom generators. This is exactly what prevents all participants from determining the mapping between signals and semantics: If we generate the gate labels using pseudorandom generators, no one can conclude the mapping between signals and semantics, since it is scrambled (for a bit more on this, please consult section A.6).

So far, we have seen how we can compute the output *signal* of a certain gate, given its input signals (not its input *semantics*). That way, we can "go through" the circuit and evaluate one gate after another, finally arriving at the gates whose outgoing wires represent the functions output value.

If we now want to compute our original function, we can just evaluate the circuit associated with proper inputs. Since the mappings from from signals to semantics are known for the output wires, we can reconstruct the solution representing our function value.

4 A (more) formal depiction of the protocol

We saw in section 3.2 how we can transform any function (represented as a circuit) into a garbled circuit that can be used later to evaluate the function given certain inputs. We will now give a formal summary that explains how to construct a garbled circuit and how to evaluate it locally. This steps are in principle taken from [1], with some explanations added.

The protocol itself has two main steps. First, the participants *collaboratively* construct the garbled circuit needed for our function. In the second step, the participants *locally* do the (expensive) evaluation of the garbled circuit. But before we start with the actual construction, we have to ensure that the circuit we are starting with has a certain format. We will now discuss all these aspects in more detail.

4.1 Bringing the original circuit into a suitable format

We already mentioned in section 3.1 that we can assume the circuit is in a suitable format. We briefly recapitulate what the circuit should look like: It should have no "cycles", only gates with a fan-in/fan-out combination of 2/1 or 1/2. It is a well known fact that each circuit can be represented using only such gates without increasing the circuit's size too much (i.e. the size stays polynomial).

Introducing "splitters". As we will discuss later (and as it was shown roughly ten years after the protocols initial publication by [14]), we have to require the following for our circuit: No wire is used more than once as an input wire. This is necessary, because otherwise some gates may be related in a way that might reveal something about the semantics of a wire, wihch is what we wanted to prevent. For a short explanation why this is necessary, see section A.7 or have a look at [14].

To be able to ensure this, we need so-called *splitter* gates that take one signal as input and have two outgoing wires. The incoming signal is then just forwarded to each of the outgoing wires. While this might seem just to be a complication of matters, we will later inspect why this is necessary.

That is, whenever one input wire is used as input to more than one gate, we use splitters to split up the signal onto several distinct wires. Again, note that this doesn't increase the size of the circuit more than by a polynomial factor.

Keep in mind that our circuit C has now two kinds of gates: Gates taking two inputs and returning one output (the "ordinary gates"), and gates taking one input and returning two outputs (the splitters we had to introduce).

4.2 Constructing the garbled circuit

In order to understand how the protocol works, let us first consider the inputs to our problem:

Common input. The common input can be seen as a string:

$$c = 1^{k'}\#1^n\#1^\ell\#1^l\#C$$

This string is basically just used so that all participants know "what's going on". The parameter k' defines a lower bound for the security parameter (see below how we obtain the actual security parameter k from k'), while n is the number of participants, ℓ stands for the length of one party's input[9] and l is the length of the output value.

That is, we – formally – consider a function $f : (\Sigma^\ell)^n \to \Sigma^l$ accepting n inputs of length ℓ and returning one value of length l. We could assume simpler sets here, but we chose to use these to be consistent with [1]. C is a suitable circuit-representation of the function f.

Private inputs. Each participating party i has a private input $x_i \in \Sigma^\ell$ that shall not become known to other participants.

Computing gate labels and semantics. Let us in the following assume that C has Γ gates (numbered $1, 2, \ldots, \Gamma$) and W wires (numbered $1, 2, \ldots, W$, including input, internal and output wires). The input wires are the wires $1, 2, \ldots, n\ell$, output wires are assigned the numbers $W - l + 1, W - l + 2, \ldots W$.

What we will now do is computing the garbled circuit. That is, we will collaboratively compute gate labels for each gate and we will compute the semantics for each wire. Remember that it is crucial to keep the semantics secret so no participant can conclude the "real values" along the wires, but just sees the signals. Moreover, we will later just "publish" the correct signals for the input wires and the gate labels. We will compute these values collaboratively and randomly.

We remember that we needed four gate labels for each gate[10]. Moreover, we said that each player shall contribute k bits to each signal. Be careful here: Each player has to generate k bits for every wire *for the even and for the odd signal.* Additionally, we stated that we wanted all players to contribute one bit to the semantics λ^ω for some wire ω. That means, each player i has to generate a random bit string $r_i \in \Sigma^{2kW+W-l}$ (W is the number of wires in total).

The strings r_i will later be interpreted in the following way:

$$r_i = s^1_{0,i}, s^1_{1,i}, \ldots, s^W_{0,i}, s^W_{1,i},\ \lambda^1_i, \ldots, \lambda^{W-l}_i \tag{5}$$

[9] We assume all inputs from the parties to be of equal length.

[10] A "normal gate" needs four gate labels, because it has exactly two input wires that can carry even or odd signals each.

A splitter gate needs four labels as well, since it has one input and two outputs. Thus, the input signal can be even or odd, and we need these two possibilities for each of the two outgoing wires.

Each string $s^{\omega}_{p,i}$ is then a random bit string of length k generated by player i that "participates" in the parity-p-signal for wire ω (and thus, implicitly in the gate labels). The value λ^{ω}_{i} is some bit that will later "participate" in the *semantics*[11] of wire ω. See below how this is to be understood. Note that we do only need $\lambda^{1}_{i}, \ldots, \lambda^{W-l}_{i}$ since we wanted the semantics of the l output wires to be public.

Now, let the actual security parameter $k = \max(k',\ \sqrt[10]{|c|})$. This is a technical detail that is necessary due to our definition of adversaries. These adversaries are allowed to use polynomial time in $|c|$, and not in k. For more information please consult [1].

Now, the parties compute gate labels and (garbled) input signals with security parameter k – tolerating the presence of at most $\lfloor (n-1)/2 \rfloor$ adversaries (speak: they "information-theoretically $\lfloor (n-1)/2 \rfloor$-securely" compute all this) using the following technique[12]:

Private inputs. The private inputs $x_1, \ldots, x_n$ define the bits $b^1, \ldots, b^{n\ell}$ associated with each input wire according to $b^1 \ldots b^{n\ell} = x_1 \ldots x_n$. Each x_i supplies l bits (namely $b^{(i-1)\ell+1}$ to $b^{i\ell}$).

Defining the signals. We now define for each wire $\omega \in \{0, \ldots, W\}$ and for each parity $b \in \{0, 1\}$ the signals:

$$s^{\omega}_{b} := s^{\omega}_{b,1} \ldots s^{\omega}_{b,n} b \tag{6}$$

This means that for $b = 0$, we have the even signals, while $b = 1$ gives an odd signal. Note that only the input signals (representing the correct semantics) can later be revealed to the public safely. But remember: we won't publish the semantics!

The semantics of the wires. We come now to a central part of this construction: We define the semantics of each wire ω.

$$\lambda^{\omega} = \begin{cases} \lambda^{\omega}_{1} \oplus \cdots \oplus \lambda^{\omega}_{n} & \text{for } 1 \leq \omega \leq W - l \\ 0 & \text{for } W - l + 1 \leq \omega \leq W. \end{cases} \tag{7}$$

Please note that equation (7) contains just a *definition.* It is important that neither λ^{ω} nor λ^{ω}_{i} become public. We define a notational shortcut: We say that λ^{ω} is the semantics of signal s^{ω}_{0}, while $\overline{\lambda^{\omega}} := 1 - \lambda^{\omega}$ is the semantics of signal s^{ω}_{1}.

Signals along input wires. For each input wire ω of the circuit (i.e. for each $\omega \in \{1, 2, \ldots, n\ell\}$), the string σ^{ω} is given by

$$\sigma^{\omega} = s^{\omega}_{(b^{\omega} \oplus \lambda^{\omega})}. \tag{8}$$

To research what σ^{ω} means, we assume we have an input wire (in the sense that it is an input wire to the whole circuit) $\omega \in \{1, 2, \ldots, n\ell\}$ that has semantics λ^{ω}. Then, the subscript $(b^{\omega} \oplus \lambda^{\omega})$ denotes the parity that is needed for this particular input wire to correctly represent the semantics of the player's input. If we then take $s^{\omega}_{(b^{\omega} \oplus \lambda^{\omega})}$, we can be sure that we are issuing the correct input signal (one can simply test this with sample values for b^{ω} and λ^{ω}).

[11] Remember that we stated that if $\lambda^{\omega} = 0$ then the even signal is associated with semantics **0** and the odd signal is associated with semantics **1**. If $\lambda^{\omega} = 1$, it is just the other way around.

[12] To be more precise, this technique relies on the fact that the underlying "sub-protocols" used to perform it enable us to information-theoretically $\lfloor (n-1)/2 \rfloor$-securely compute gate labels and garbled input signals.

Note that even if the original input signal comes from one single player, the computation of $s^{\omega}_{(b^{\omega}\oplus\lambda^{\omega})}$ has to be done collaboratively, since it involves λ^{ω} (the semantics for wire ω) that shall not become public and since signals are composed of bits supplied by all players. However, that collaborative computation is no problem, since we already saw that we can compute the value:

$$(b^{\omega} \oplus \lambda^{\omega}) \overset{(7)}{=} (b^{\omega} \oplus \lambda^{\omega}_1 \oplus \dots \oplus \lambda^{\omega}_n) \tag{9}$$

is, as we see, essentially just a XOR of bits whose associated circuit can be secretly and securely evaluated in a constant number of rounds and with polynomial communication.

In this way, all players learn the value σ^{ω} without learning the value b^{ω} (which shall stay private to the player holding it). The players then only compute the signal associated with the obtained parity. This way, only this signal is learnt by all players.

Constructing the gate labels. It remains to explain how we can – given the input signal(s)[13] to a gate g – later be able to compute the output of this certain gate. Therefore, we define the gate labels (as explained in section 3.2).

We assume now that α, β denote input wires carrying signals σ^{α}_a and σ^{β}_b, respectively. The incoming signal(s) have the following format[14]:

$$\sigma^{\alpha}_a = s^{\alpha}_{a1} \dots s^{\alpha}_{an} a \quad \text{and} \quad \sigma^{\beta}_b = s^{\beta}_{b1} \dots s^{\beta}_{bn} b$$

This means: On wire α, there's the signal with parity a for that specific wire, consisting of the n sub-strings s^{α}_{ai} of length k each, and with the parity bit a. We have the corresponding parts for σ^{β}_b.

Remember: We have two kinds of gates in our circuit: The "ordinary gates" taking two inputs and returning one output and the "splitters" taking one input and having two output wires.

"Ordinary gates". We will start with the "ordinary gates". These gates get two input signals and return one output signal. For each ordinary gate g in the circuit (i.e. g is some number in $\{1, 2, \dots ; \Gamma\}$), we define the gate labels $A^g_{00}, A^g_{01}, A^g_{10}, A^g_{11}$ as follows: If gate g computes a binary function we denote by the symbol $\otimes$, and has left wire α, right wire β, and output wire γ (i.e. $\alpha, \beta, \gamma \in [1 \dots W]$), then we define for $a, b \in \{0, 1\}$ the gate label A^g_{ab} by

$$A^g_{ab} = G_b(s^{\alpha}_{a1}) \oplus \dots \oplus G_b(s^{\alpha}_{an}) \ \oplus \ G_a(s^{\beta}_{b1}) \oplus \dots \oplus G_a(s^{\beta}_{bn}) \ \oplus \ s^{\gamma}_{[(\lambda^{\alpha}\oplus a)\otimes(\lambda^{\beta}\oplus b)]\oplus\lambda^{\gamma}} \tag{10}$$

Splitter gates. We will now do the analogous thing for splitter gates (for reference, see [15]). Even if we didn't specify these gates in section 3.2, the underlying thoughts are the same. Remember that a splitter just forwards its input signal onto the two outgoing wires. We call the single input wire α and denote the output wires by γ_0 and γ_1. Then, the gate labels are as follows (for $a, b \in \{0, 1\}$):

$$A^g_{ab} = G_b(s^{\alpha}_{a1}) \oplus \dots \oplus G_b(s^{\alpha}_{an}) \oplus s^{\gamma_b}_{(\lambda^{\alpha}\oplus a)\oplus\lambda^{\gamma_b}}$$

As said before, G_0 and G_1 denote the "zeroth" and the "first" halves of the output of a pseudorandom generator G. This pseudorandom generator takes a binary

[13] Again without knowing the input *semantics*.

[14] Depending on the type of the gate, we have either two inputs (for ordinary gates) or only one input (for splitters).

string of length k (k being the security parameter) and stretches it into a longer string. The length of this longer string has to be fixed a priori, which is why we assume that the generator returns a binary string of length $2(\overline{n}k + 1) + 1$, with e.g. $\overline{n} = k^{10}$. The term $\overline{n}$ bounds the number of players that can participate in the protocol depending on the security parameter. Thus, $\overline{n} = k^{10}$ shouldn't be a terribly restrictive constraint, but can be increased at will (at least as long as it is a fixed power of k). For the reason we define A^g_{ab} exactly like that, please refer to section 3.2. For an intuition about why we need the pseudorandom generators, please refer to, e.g., section A.6.

Outcome of the first phase of the protocol. We have computed two important things collaboratively: The whole garbled circuit (consisting of gate labels and associated wires) and the input signals (σ^ω for $\omega \in \{1, 2, \dots, n\ell\}$).

The semantics (λ^ω for $\omega \in \{1, 2, \dots, W - l + 1\}$) are *not known* to the participants. It is crucial that this is the case, as otherwise the players could deduce e.g. the input values and internal values of wires. It is all the more surprising that we can evaluate the function correctly, even if we don't know the internal wires values.

What we've done now is the following: The parties collaboratively computed the garbled circuit (with security parameter k, and $\lfloor (n-1)/2 \rfloor$-securely). Note that we didn't exactly specify how e.g. the several XOR-operations are computed collaboratively, but we just said that *we are using* some techniques that enables us to do so. We saw in section 2.5 that such a protocol for XOR exists.

In this way, we can rely upon the existing protocols that serve as building blocks for our protocol. Moreover, this takes the burden of some proofs from us: We just know that there exist some protocols that we can use as building blocks for our protocol, and if these building blocks have specific properties, they automatically apply for our protocol.

In particular, we saw that every step described could be evaluated in a constant number of rounds and with polynomial overhead. Since the first phase has no loops or recursions, the overall complexity is still within a constant number of rounds and polynomial amount of communication.

It remains now for the single parties to evaluate the obtained garbled circuit. This is done in the second phase.

4.3 Locally evaluating the garbled circuit

We saw in the previous section how to construct the garbled circuit and will now examine how to evaluate the garbled circuit we obtained by the above routine.

As before, take a first look at the inputs:

Common input As before, we have a common string $c = 1^{k'}\#1^n\#1^\ell\#1^l\#C$. For the meanings of all the symbols, please consult section 4.2.

Input Now we have a garbled program $\hat{y}$, i.e. we have gate labels $A^g_{00}, A^g_{01}, A^g_{10}, A^g_{11}$ (each coming from the set Σ^{nk+1} and input signals $\sigma^\omega \in \Sigma^{nk+1}$ (for $g \in [1 \dots \Gamma]$ and $\omega \in [1 \dots nl]$).

We want to compute the string $y \in \Sigma^l$ that this garbled program evaluates to. As in the previous step, we set $k = max(k', \sqrt[10]{|c|})$. It is necessary for a player i to know the value k, so that the player knows how to dissect a signal into n binary strings of length k.

The following is carried out by each player i (since we want to evaluate locally). That is, each player has to "start at the input wires" (using the values σ^1 to $\sigma^{n\ell}$) and work his way up to the output wires to evaluate the function.

We are now going to see how we can obtain the signals for the outgoing wires of the gates. Therefore, we again distinguish between "ordinary" and splitter gates:

"Ordinary gates". We consider how to evaluate an ordinary gate g with left input wire α, right input wire β and output wire γ (i.e. $\alpha, \beta, \gamma \in [1 \dots W]$) given its input signals.
Let $\sigma^\alpha = \sigma_1^\alpha \dots \sigma_n^\alpha a$ be the signal for wire α, and $\sigma^\beta = \sigma_1^\beta \dots \sigma_n^\beta b$ for wire β. Then, we want to compute the resulting signal σ^γ for wire γ. Therefore we use the appropriate gate labels:

$$\sigma^\gamma = G_b(\sigma_1^\alpha) \oplus \cdots \oplus G_b(\sigma_n^\alpha) \;\oplus\; G_a(\sigma_1^\beta) \oplus \cdots \oplus G_a(\sigma_n^\beta) \;\oplus\; A_{ab}^g \tag{11}$$

Splitter gates. Similarly, we compute the outgoing signal from a splitter gate. As we said, in splitter gates we only have one incoming signal, namely $\sigma^\alpha = \sigma_1^\alpha \dots \sigma_n^\alpha a$, and we have two outgoing wires γ_0 and γ_1 which we want to compute the signals for. That is, for $i \in \{0, 1\}$ we compute (according to [15])

$$\sigma^{\gamma_i} = G_i(\sigma_1^\alpha) \oplus \cdots \oplus G_i(\sigma_n^\alpha) \;\oplus A_{ai}^g \tag{12}$$

Please note that equation (11) resembles equation (2), which was the starting point of our thoughts when it came to how we would like to construct the gate labels. Equation (12) was derived in an analogue way.

So, now we have seen how we compute the output of a certain gate. Thus, we can now "propagate up the circuit" and will reach the output wires. As soon as a player has computed a signal for each wire, she's basically done and can read the function output y:

$$y = \text{lsb}(\sigma^{W-l+1}) \dots \text{lsb}(\sigma^W) \tag{13}$$

In equation (13), lsb denotes the least significant (in our case the last) bit. We can be sure that $\text{lsb}(\sigma^{W-l+1}) \dots \text{lsb}(\sigma^W)$ is directly the output of the function, because we explicitly stated that the output wires ($W - l + 1$ to W) have semantics that are publicly known (namely $\lambda^{W-l+1} = \cdots = \lambda^W = 0$).

4.4 Review of the steps

Ensure that the circuit has a certain format. Let us briefly recapitulate what we have done. We first ensured that the circuit representing the desired function has a certain format (no cycles, only two-in-one-out- and one-in-two-out-gates, no wire used as input in more than one gate). It should be clear that transforming any circuit into a circuit satisfying these constraints can be done in polynomial time and moreover requires no communication between the players at all.

Collaboratively computing the garbled circuit. When the players computed the garbled circuit, we said that each player generates a random string of length $2kW + W - l$ (where k is the security parameter, W the number of wires). The obtained random bits are then used to define the semantics and signals for each wire. Note that the size of the gate labels and the signals is polynomial (in the size of the circuit and the security parameter). Moreover, when the signals and gate labels are computed, only a polynomial amount of information is needed to perform these computations.

Thus, because the players only need to communicate that amount of information, we have a polynomial amount of communication.

Evaluating the circuit locally. Each player obtains now the garbled circuit and the garbled inputs. According to the presented rules, each player locally evaluates the circuit and finally obtains the desired result. This last step requires no communication at all, it can be done by each player individually.

5 Outcome

As we saw, we could use (verifiable) secret sharing, collaborative evaluation of constant-depth circuits (in particular collaborative XOR), pseudorandom generators as a basis for a new protocol the securely and secretly evaluates a given function. These protocols are known to work in presence of dishonest players, as long as these players are a minority. Furthermore, they work in a constant number of rounds using a polynomial amount of communication. These techniques are then used to collaboratively compute a garbled circuit (with garbled inputs), that is in turn locally evaluated by each player individually.

Note that we didn't exactly specify which technique should be used for these actions. We just "ensured" that *there are* techniques that we can use.

That, in turn, simplifies proofs, since *we can assume* that for these building blocks we have come constant-round polynomial-communication protocols. The proof then builds upon these assumptions.

Even if we won't go through the proof, let us briefly restate what we have seen by now: It is possible to collaboratively evaluate a function f on inputs $x_1, \ldots, x_n$ such that player i supplies x_i and that all the inputs stay private to the respective players. This is even more surprising if one takes into account that we did *not* bound the size of the circuit in some way.

Proving the above statement requires very sophisticated arguments, which is why we won't do an analysis of the proof. For these reasons, we omitted a more formal in-depth-treatment of the protocol. Inclined readers find a more detailed proof in [1].

But reader beware: the learning curve for understanding these proofs is not small. A variety of definitions, formalizations, lemmata and theorems has to be studied before one can even start to think about the proof. For a relatively up-to-date treatment of garbled circuits that introduces necessary concepts, [4] might be a good start.

A Deferred Explanations

A.1 How a dealer is "bound" to a committed value

In section 2.5 we said that the dealer is bound to his shared value once he committed to it. Of course, there are several ways to implement this, but we will briefly describe one from [3].

We use a one-way collision-free hash function H (a one-way function is a function that can be easily evaluated, but whose inverse is very complicated or impossible to compute). Then, if we want to commit to some value x, we can send as a "promise" the value $y = H(x)$. Note that – informally stated – nobody is able (at least not in feasible time) to find another value $x \neq y$ with $H(x) = H(x') = y$.

In this way, we are bound to the value x, since we already published $y = H(x)$, but cannot find another value that yields y when we apply H to it. Thus, if we later publish our secret x, the other players simply need to check whether they obtain y when they apply H to it. This way, we cannot change our mind without being caught.

A.2 The basic idea of secret sharing

Consider the following basic idea for secret sharing. Assume you have some bit s that you want to share among three players such that no player con reconstruct the value on her own (but they should be able to reconstruct it altogether).

Then, you could toss two coins yielding random bits s_1 and s_2. Moreover, you construct a third bit $s_3 = s_1 \oplus s_2 \oplus s$. Note that then $s = s_1 \oplus s_2 \oplus s_3$. Then you give s_1 to player 1, s_2 to player 2 and so forth. Afterwards the three players can – collaboratively – reconstruct the value s, but none of them can do so on her own.

This is – of course – a very simple scheme, that only works if all players are honest and if *all* players join the collaborative computation.

However, there are more sophisticated techniques (e.g. relying on polynomials over finite fields like $\mathbb{Z}_p$ for some prime p) that allow a certain amount of dishonest players and still work.

For more information on that topic, please refer to [6].

A.3 Collaborative coin flipping – basic idea

Taking the XOR (evaluating securely and secretly) of all these bits returns the random bit generated by all the participants in common. This computation (XOR on any number of bits) can be done in a constant number of rounds and using only polynomial amount of communication. This can be seen, for example, by a deeper investigation of the arguments in [9] (according to [2]).

From now on, we use the symbol $\oplus$ to denote the XOR-computation. Even if XOR is usually defined on single bits only, we apply this function to bit strings bit-wise (e.g. $1100 \oplus 1001 = 0101$).

A.4 Some words on Pseudorandom Generators

It is a widely known fact that pseudorandom generators exist if and only if so-called *one-way* functions exist. These are functions that are easy to compute, but hard to invert. For more information on the relation between one-way functions and pseudorandom generators, please refer to [3,13,12].

Even now, whether such one-way functions exist is an open question. Thus, it is – as mentioned in section 2.5 – not proven that pseudorandom generators exist at all.

One remark that has to be made in this context is the following: The existence of one-way functions (or – equivalently – of pseudorandom generators) directly implies $\mathbf{P} \neq \mathbf{NP}$ ($\mathbf{P}$ and $\mathbf{NP}$ being the famous complexity classes, of course). This is simply, because the inverse of such a one-way function would be "hard to compute, but easy to verify", which is – informally – what $\mathbf{NP}$ is about.

Thus, a proof for the existence of pseudorandom generators is probably – to put it mildly – none of the simple-to-read ones.

A.5 Size of two-fan-in-circuits

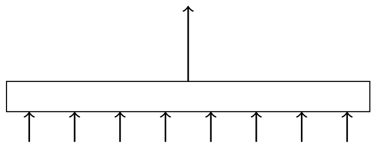
We can reduce any circuit to a circuit consisting only of gates having two input wires, increasing its size only by a polynomial factor.

To prove this, consider an AND-gate having $n > 2$ wires. Then, we can simulate this gate using only AND-gates with two input wires.

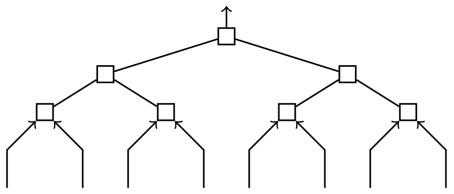
We do this in a manner similar to a complete binary tree (see Figure 4).

The bottom level consists of at most $\frac{n}{2}$ AND-gates. The second-to-bottom level consists of at most $\frac{n}{4}$ gates. The next level has at most $\frac{n}{8}$ gates, and so forth. Thus, the total number of gates used is at most $\sum_{i=1}^{\log n} \frac{n}{2^i} \leq n$.

Thus, for one AND-gate that has n inputs, we can construct a sub-circuit consisting of at most n gates such that this sub-circuit resembles the computation of the original AND-gate. The size increased by a polynomial factor (in the number of inputs of the original gate).



(a) A single AND-gate with eight inputs.



(b) Several 2-input AND gates simulate one 8-input AND gate.

Fig. 4. Simulating an n-input AND-gate by several 2-input AND-gates

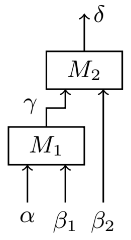


Fig. 5. A circuit computing $(x \wedge y_1) \wedge y_2$. Wire α carries the signal for x, while wires β_1 and β_2 carry the signals for signals y_1 and y_2, respectively. All gates are AND-gates.

For other types of gates, we can use similar constructions.

Since any gate can have at most as many inputs as there are gates in the whole circuit, the size of the total circuit increases by a polynomial factor in the size of the circuit.

Note that this idea does not only work for AND gates, but also for other gates. In particular, it is working for splitter gates (as introduced in section 4.1).

A.6 Why the use of Pseudorandom Generators?

Let us for a moment consider equation (3) and propose the following question: The output of G_0 resp. G_1 on some k-bit string is a binary string of length $nk+1$. So, one might argue that it would be simpler to work directly on the signals $\sigma = \sigma_1 \dots \sigma_n a$ and $\tau = \tau_1 \dots \tau_n b$ instead of first splitting up the signals into its components and then considering $G_b(\sigma_i)$ and $G_a(\tau_j)$, respectively.

While this simplification would be useful if we were just interested in correctness, we would lose privacy, if this was done. That is, a player possibly can deduce the semantics of some signals – which is something we strictly want to prevent.

To see this, consider a simple circuit computing the function $(x \wedge y_1) \wedge y_2$. This circuit is depicted in figure 5. Assume that this circuit does not use pseudorandom generators, but instead operates directly on the signals. We will now inspect how the player supplying x can deduce the plain-text of y_2, even if he supplies $x = 0$.

Assume that the wire for x is called α, the wires for y_1 and y_2 are β_1 and β_2, respectively. First, we take the gate M_1 (input wires α and β_1) into account. The respective output wire (holding the garbled signal for $x \wedge y_1$) is denoted by γ.

Since we obtain the garbled circuit and the garbled input signals, we already know the (garbled) signals along α and β_1. We assume them without loss of generality to be of parity 0.

So, we know the signal s_0^α and that this signal represents plain-text $\mathbf{0}$ (i.e. we know s_0^α and $s_0^\alpha \leftrightarrow \mathbf{0}$). Moreover, we know $s_0^{\beta_1}$, but we do not know what $s_0^{\beta_1}$

stands for (we do not know the plain-text). We then – of course – can compute the outgoing signal along wire γ (using the gate labels). Without loss of generality, we assume that the output signal has parity 0 (that is, we have signal s_0^γ along wire γ). We know that $s_0^\gamma \leftrightarrow \mathbf{0}$ since we supplied the signal for $x = 0$ and, thus, $(x \wedge y_1) = (0 \wedge y_1) = 0$.

Using only these things, and since we know *all* gate labels, we can now compute some other values: We know that if we supply the signal $s_1^{\beta_1}$, the outcome along wire γ must still represent $\mathbf{0}$, since the gate represents an AND and $s_0^\alpha \leftrightarrow \mathbf{0}$. Thus, we can say that $s_0^\gamma = A_{01} \oplus s_0^\alpha \oplus s_1^{\beta_1}$. We can solve this equation for $s_1^{\beta_1}$ (we know $A_{01}, s_0^\alpha, s_0^\gamma$). That is, by now we have both signals for wire β_1.

We now introduce the following notation: σ_{xy}^ω denotes the signal that would be obtained along wire ω if we used signals of parities x and y as inputs to the corresponding gate.

Next, we consider two equations:

$$\sigma_{10}^\gamma = A_{10} \oplus s_1^\alpha \oplus s_0^{\beta_1} \tag{14}$$

$$\sigma_{11}^\gamma = A_{11} \oplus s_1^\alpha \oplus s_1^{\beta_1} \tag{15}$$

As said above, σ_{10}^γ denotes the signal we obtain if we supply s_1^α and s_0^β along the input wires to the gate. So these two equations basically tell us what the output would be if we supplied particular input signals. Note that *exactly one* of the following holds:

$$(s_0^\gamma = \sigma_{01}^\gamma \wedge s_1^\gamma = \sigma_{11}^\gamma) \quad \text{or} \quad (s_1^\gamma = \sigma_{01}^\gamma \wedge s_0^\gamma = \sigma_{11}^\gamma).$$

We solve equations (14) and (15) for s_1^α, and deduce that

$$\sigma_{10}^\gamma \oplus \sigma_{11}^\gamma = A_{10} \oplus A_{11} \oplus s_0^{\beta_1} \oplus s_1^{\beta_1}.$$

Thus, the left-hand-side $\sigma_{10}^\gamma \oplus \sigma_{11}^\gamma$ denotes exactly the "bit-difference" of s_0^γ and s_1^γ. This means, we can deduce the value s_1^γ (simply by computing $s_1^\gamma = s_0^\gamma \oplus (A_{10} \oplus A_{11} \oplus s_0^{\beta_1} \oplus s_1^{\beta_1})$) and we know that $s_1^\gamma \leftrightarrow \mathbf{1}$.

Now we go onto the second gate M_2 and recognize that we know *all* signals for wire γ with the associated plain-text values and the signal for wire β_2 representing the plain-text y_2. Without loss of generality, we assume that the parity of the signal for y_2 is 0 (but we do not know what this parity-0-signal stands for).

We know that if we supply all our original garbled signals (in particular the signal for $x = 0$), the output of M_2 must represent plain-text 0, since $(0 \wedge y_1) \wedge y_2 = 0$. We assume without loss of generality that $s_0^\delta \leftrightarrow \mathbf{0}$ (i.e. that the original garbled output signal representing plain-text $\mathbf{0}$ has parity 0).

Analogously to the gate M_1, we deduce $s_1^{\beta_2}$ from the given values. That is, we now know for gate M_2 the following:

- Left input signals and associated semantics (i.e. we know s_0^γ and s_1^γ and $s_0^\gamma \leftrightarrow \mathbf{0}, s_1^\gamma \leftrightarrow \mathbf{1}$)
- Right input signals, but not yet associated semantics (i.e. we know $s_0^{\beta_2}, s_1^{\beta_2}$)
- Output signal for plain-text 0 (i.e. we know s_0^δ and $s_0^\delta \leftrightarrow \mathbf{0}$)

Using all these, we can simply "test" gate M_2 by supplying several signals and looking at the output signal: We supply all *original* garbled signals and know that the output must represent 0. Then, we supply s_1^γ (representing plain-text 1) and check which signal for wire β_2 changes the output.

To be more precise, we compute σ_{10}^δ and σ_{11}^δ as follows:

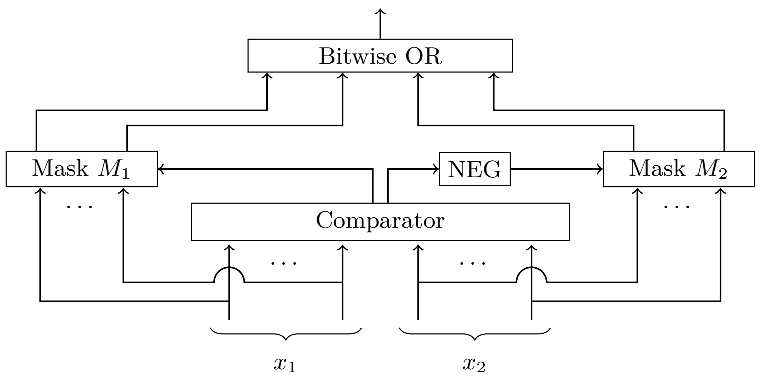


Fig. 6. A circuit representing the function $\max(x_1, x_2)$. This circuit has not the property that each wire is an used as an input wire at most once. Section A.7 explains how this can be exploited by tackling the subcircuit M_1.

$$\sigma^{\delta}_{10} = A^{M_2}_{10} \oplus s^{\gamma}_{1} \oplus s^{\beta_2}_{0}$$
$$\sigma^{\delta}_{11} = A^{M_2}_{11} \oplus s^{\gamma}_{1} \oplus s^{\beta_2}_{1}$$

If $\sigma^{\delta}_{10} = s^{\delta}_{0}$, then $s^{\beta_2}_{0}$ represents plain-text **0**. If $\sigma^{\delta}_{10} = s^{\delta}_{1}$, then $s^{\beta_2}_{0}$ represents plain-text **1**. Executing the above steps, we deduced the plain-text of y_2.

How can pseudorandom generators help? As we saw, if we pass pseudorandom generaturs up, this enables players to deduce the proper signals for wires. "Testing" a gate with the obtained signals then allows them to deduce the semantics of the signals to obtain the inputs of other parties. In this case, the gate labels are not obscured enough so that they leak some precious information.

On the other hand, if we use pseudorandom generators, we cannot deduce all the other signals, since the gate labels A^{g}_{ab} are scrambled. This prevents us from simply reconstructing the signals (and, thus, the semantics).

A.7 Why do we need splitters?

We illustrate the need for splitters by an example along [15]. Let us imagine the situation, where two parties hold two numbers x_1 and x_2 (each consisting of l bits). The parties want to know the higher of the numbers, i.e. $\max(x_1, x_2)$.

Therefore, they could use a circuit as depicted in figure 6. It takes two l-bit inputs x_1 and x_2 and gives them to a comparator that outputs one bit [15]. This output is 1 if $x_1 \geq x_2$, and 0 otherwise.

Moreover, it uses two *masks*. A mask takes an l-bit input vector (in figure 6, depicted as the incoming wires from the bottom) and a single *masking* bit (shown as a single wire from the side). Its output is a l-bit-vector. This vector holds zeroes only, if the masking bit is 0. Otherwise, the output vector is the same as the input vector.

Let us assume that the players already computed the outcome and saw that x_2 was the higher of the two numbers (i.e. player 2 had the higher number). We now consider what player 2 might do retrieve the value x_1 (which he was not supposed to learn).

[15] This output is shown as two separate wires, since it is used in several gates as input.

Remember: We consider now what would happen if we would *not* have used splitters, but instead just relied on the "pure" protocol described in [1].

The crucial part of this splitter-less garbled circuit is M_1, which is – as a close-up – depicted in figure 7. This is a sub-circuit receiving input wires $\alpha_1, \ldots, \alpha_l$ and one additional input wire β. Then each α_i is ANDed with β and the results are "returned" as values $\gamma_i = \alpha_i \oplus \beta$. Note that in this case β is participating in several different gates, since we did *not* introduce splitters.

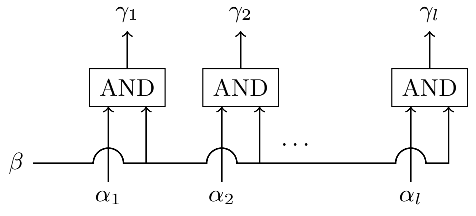


Fig. 7. The mask M_1 from the circuit in figure 6. This sub-circuit can be exploited because one single wire – namely wire β – participates in several gates.

Player 2 already knows that his original number x_2 was at least as big as x_1. Now, he could try what would happen if x_1 represented zero[16].

Since the circuit – evaluated on x_1 and x_2 – revealed some signals $\sigma^{\gamma_1}, \ldots, \sigma^{\gamma_l}$ to player 2, and we know that $x_2 \geq x_1$, we know that the signals σ^{γ_i} must represent semantics **0**.

If all bits represented by signals σ^{α_i} represented **0**, this would result in signals σ^{γ_i} representing **0** even if the signal σ^{β} represented **1** (since all the gates under consideration are AND-gates).

We now make a guess, that is likely wrong: We simply assume that *all* signals σ^{α_i} represent plain-text **0**. What follows, is essentially described by this: For the correctly-guessed bits (i.e. for those bits that are actually 0), we will get "useful" results, while for the wrongly-guessed bits, we will only obtain random stuff. We will now see what this means in detail.

So, let us now consider the i-th AND-gate within M_1, and look what we can deduce:

We assume now that b is the parity of the signal along wire β, and a_i is the parity of signal on input wire α_i. Then, for the AND-gate g_i, we could compute σ^{γ_i} as follows:

$$\sigma^{\gamma_i} = A^i_{ab} \oplus \underbrace{G^*_b(\sigma^{\alpha_i}_{a_i})}_{G_b(\sigma^{\alpha_i}_{a_i,1}) \oplus G_b(\sigma^{\alpha_i}_{a_i,2})} \oplus \underbrace{G^*_{a_i}(\sigma^{\beta}_{b})}_{G_{a_i}(\sigma^{\beta}_{b,1}) \oplus G_{a_i}(\sigma^{\beta}_{b,2})} \tag{16}$$

Note that the term $G^*_{a_i}(\sigma^{\beta}_{b})$ participates in *all* AND-gates. Moreover, note that each a_i can either be 0 or 1. That means, *each AND-gate* within M_1 has either $G^*_0(\sigma^{\beta}_{b})$ or $G^*_1(\sigma^{\beta}_{b})$ as a contribution to the signal σ^{γ_i}.

That is, one of these terms is present in all gates within M_1.

The fact we will exploit is that $G^*_0(\sigma^{\beta}_{b})$ and $G^*_1(\sigma^{\beta}_{b})$ are *the same in several gates.* We can easily "solve" equation (16) for $G^*_{a_i}(\sigma^{\beta}_{b})$ and rename the result to μ_i:

$$\mu_i := \sigma^{\gamma_i} \oplus G_b(s^{\alpha_i}_{a,1}) \oplus G_b(s^{\alpha_i}_{a,2}) \oplus A^{g_i}_{a_i b}. \tag{17}$$

[16] That is, essentially, player 2 asks himself: "If x_1 really represented **0**, what could I deduce from that?"

We consider the parts of the right-hand side:

- The values σ^{γ_i} is known since it must represent semantics 0. Since we already evaluated the garbled circuit with $x_2 \geq x_1$, we know these signals.
- The outputs of our pseudorandom generator, $G_b(s_{a,1}^{\alpha_i})$ and $G_b(s_{a,2}^{\alpha_i})$, are known because we know the signals along wires α_i and can just compute the output of the pseudorandom generators G_0 and G_1 on the respective inputs.
- The gate labels $A_{a_i b}^{g_i}$ are trivially known since they are part of the garbled circuit.

As you can see, we can compute the vales μ_i without any further complications.

Now comes a key insight. Note that μ_i contains the following information: If our guess for x_i was right (i.e. if the signal α_i indeed represented **0**), then it follows immediately that $\mu_i = G_a(s_b^{\beta})$.

If our guess was incorrect, μ_i will be some random binary string.

This means: If we guessed the i-th bit of x_1 correct, then μ_i will be either $G_0(s_{b1}^{\beta}) \oplus G_0(s_{b2}^{\beta})$ or $G_1(s_{b1}^{\beta}) \oplus G_1(s_{b2}^{\beta})$. Incorrect guessed bits yield random strings.

Now, we can do the following: Note that – in a collection of random, sufficiently long bit-strings – the probability that a string occurs twice is rather low. But in our collection of values μ_i, there might well be duplicate vales – the ones for whom our guess of the i-th bit was correct. So, we collect all the values μ_i and sort them into certain "groups" the following way: We collect values that occur multiple times and group them. The remaining strings form the last group.

That is, in the first and second group, all the values μ_i are collected that are probably generated by $G_0(s_{b1}^{\beta}) \oplus G_0(s_{b2}^{\beta})$ or $G_1(s_{b1}^{\beta}) \oplus G_1(s_{b2}^{\beta})$, respectively. The third group contains all the other, random strings. All indices in the third group correspond to the bits that were wrongly guessed. Thus, these bits are probably 1 (since we guessed 0). The indices in the other groups are the correctly-guessed ones.

It can be shown that the probability that we obtain the correct bits of x_1 using this technique, is rather high – too high to be acceptable in a cryptographic setting.

The weakness exploited here lay in the fact that one single value – the signal along wire β (resp. the output of a pseudorandom generator applied to it) – took part in several other gates.

If we now use splitters, this ensures that not one single signal participates in several gates, but this single signal is duplicated into many other *random* signals, so one cannot exploit the shown inter-gate dependencies.

Of course, this is just a very superficial treatment of this topic. For an in-depth proof of all the statements, we refer to [15].

References

1. Rogaway, P.: The Round Complexity Of Secure Protocols. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA (1991)
2. Beaver, D., Micali, S., Rogaway, P.: The round complexity of secure protocols. In: Proceedings of the twenty-second annual ACM symposium on Theory of computing, New York, NY, USA, ACM (1990) 503–513
3. Goldwasser, S., Bellare, M.: Lecture notes on cryptography. Lecture Notes (July 2008) Accessed March 2012 (`http://cseweb.ucsd.edu/users/mihir/papers/gb.pdf`).
4. Bellare, M., Hoang, V.T., Rogaway, P.: Garbling schemes. Cryptology ePrint Archive, Report 2012/265 (2012) `http://eprint.iacr.org/`.
5. Goldreich, O.: Foundations of Cryptography: Volume 2, Basic Applications. Cambridge University Press, New York, NY, USA (2004)
6. Shamir, A.: How to share a secret. Commun. ACM **22**(11) (November 1979) 612–613

7. Chor, B., Goldwasser, S., Micali, S., Awerbuch, B.: Verifiable secret sharing and achieving simultaneity in the presence of faults. In: Proceedings of the 26th Annual Symposium on Foundations of Computer Science. SFCS '85, Washington, DC, USA, IEEE Computer Society (1985) 383–395
8. Rabin, T., Ben-Or, M.: Verifiable secret sharing and multiparty protocols with honest majority. In: Proceedings of the twenty-first annual ACM symposium on Theory of computing. STOC '89, New York, NY, USA, ACM (1989) 73–85
9. Beaver, D.: Multiparty protocols tolerating half faulty processors. In: Proceedings on Advances in cryptology. CRYPTO '89, New York, NY, USA, Springer-Verlag New York, Inc. (1989) 560–572
10. Goldreich, O., Micali, S., Wigderson, A.: How to play any mental game. In: Proceedings of the nineteenth annual ACM symposium on Theory of computing. STOC '87, New York, NY, USA, ACM (1987) 218–229
11. Ben-Or, M., Goldwasser, S., Wigderson, A.: Completeness theorems for non-cryptographic fault-tolerant distributed computation. In: Proceedings of the twentieth annual ACM symposium on Theory of computing. STOC '88, New York, NY, USA, ACM (1988) 1–10
12. Blum, M., Micali, S.: How to generate cryptographically strong sequences of pseudo-random bits. SIAM J. Comput. **13**(4) (November 1984) 850–864
13. Yao, A.C.: Theory and application of trapdoor functions. In: Proceedings of the 23rd Annual Symposium on Foundations of Computer Science. SFCS '82, Washington, DC, USA, IEEE Computer Society (1982) 80–91
14. Tate, S.R., Xu, K.: On Garbled Circuits and Constant Round Secure Function Evaluation. Technical report, UNIVERSITY OF NORTH TEXAS (2003)
15. Xu, K.: Mobile agent security through multi-agent cryptographic protocols. PhD thesis, University of North Texas, Denton, TX, USA (2004) AAI3126591.