Applications of Secure Multiparty Computation
P. Laud and L. Kamm (Eds.)

doi:10.3233/978-1-61499-532-6-1

Chapter 1
Basic Constructions of Secure Multiparty Computation

Peeter LAUD [a], Alisa PANKOVA [a], Liina KAMM [a], and Meilof VEENINGEN [b]
[a] *Cybernetica AS, Estonia*
[b] *Eindhoven University of Technology, Netherlands*

Abstract. In this chapter, we formally define multiparty computation tasks and the security of protocols realizing them. We give a broad presentation of the existing constructions of secure multiparty computation (SMC) protocols and explain why they are correct and secure. We discuss the different environmental aspects of SMC protocols and explain the requirements that are necessary and sufficient for their existence.

Introduction

There are several cryptography textbooks that rigorously cover the basic definitions and constructions of secure multiparty computation, e.g. [1,2,3]. In this introductory chapter, we do not attempt to repeat this rigorous treatment. Instead, we will give the basic security definitions of SMC and present the major ideas behind different SMC protocols, sufficient for understanding the algorithms and protocols in the rest of this book. We refer to other sources for thorough proofs and discussions on these constructions.

1. Definitions

An SMC protocol for a functionality f allows a number of parties to evaluate f on the inputs they have provided, and learn its outputs without learning anything beyond their own inputs and outputs. Almost all SMC techniques expect f to be expressed as a Boolean or an arithmetic circuit, and process it gate by gate. We can specify a multiparty computation task using the following definitions.

Definition 1 *An* arithmetic circuit *over a ring R is a tuple $C = (G, V_{\text{in}}, V_{\text{out}}, \lambda)$, where*

- *$G = (V, E)$ is a directed acyclic graph, where the incoming edges of each vertex have been linearly ordered;*
- *$V_{\text{in}} \subseteq \{v \in V \mid \overrightarrow{\deg}(v) = 0\}$ and $V_{\text{out}} \subseteq V$ denote the input and output vertices of the circuit (here $\overrightarrow{\deg}(v)$ denotes the number of incoming edges of the vertex V);*
- *λ assigns to each $v \in V$ an operation $\lambda(v) : R^{\overrightarrow{\deg}(v)} \to R$.*

A Boolean circuit can be seen as a special case of Def. 1, where $R = \mathbb{Z}_2$. The semantics of a circuit $C = (G, V_{\text{in}}, V_{\text{out}}, \lambda)$ extend mapping $V_{\text{in}} \to R$ to mapping $V \to R$, thereby assigning values to all vertices in V_{out}.

Definition 2 *A* multiparty computation task *for a set of parties* $\mathbf{P} = \{P_1, \ldots, P_n\}$ *is a tuple* $f = (C, T_{\text{in}}, T_{\text{out}})$*, where* $C = (G, V_{\text{in}}, V_{\text{out}}, \lambda)$ *is an arithmetic circuit,* $T_{\text{in}} : V_{\text{in}} \to \mathbf{P}$ *determines which party provides each input, and* $T_{\text{out}} \subseteq V_{\text{out}} \times \mathbf{P}$ *states which outputs are learned by which parties.*

To solve the multiparty computation task, the parties execute some protocol Π, with the party P_i having an interactive (Turing) machine M_i that implements the steps of P_i in this protocol. At first, the machine M_i receives the inputs x_i from P_i, and in the end, returns the outputs to P_i. Here, x_i is a mapping from the set $T_{\text{in}}^{-1}(P_i)$ to R. We let $\mathbf{x} : V_{\text{in}} \to R$ denote the concatenation of all parties' inputs. Let $\mathbf{x}[P_i]$ denote x_i. For a subset of parties $\mathbf{P}' \subseteq \mathbf{P}$, we let $\mathbf{x}[\mathbf{P}']$ denote the tuple of all $\mathbf{x}[P]$ with $P \in \mathbf{P}'$.

In the threat model of SMC, some parties may be corrupted, but the honest parties do not know which ones. The secrecy of honest parties' inputs has to be protected against the coalition of corrupt parties. Also, the honest parties should still obtain correct outputs despite the actions of corrupt parties. We can formalize both of these requirements through the real/ideal-model paradigm. In this paradigm, we specify our desired properties through a protocol that contains an ideal component that makes these properties "obviously hold". In case of SMC, this ideal component $\mathcal{F}_{\mathsf{SMC}}^{f}$ collects the inputs of all parties, computes the functionality f, and hands the outputs back to the parties. Correctness and secrecy are obvious, because $\mathcal{F}_{\mathsf{SMC}}^{f}$ indeed computes f, and each party only gets back its own outputs. The execution of this ideal protocol produces certain outputs for the honest parties, as well as for the adversary (modeled as a Turing machine) controlling all the corrupted parties. The output by the adversary may reflect its guesses about the inputs and outputs of honest parties. In the real protocol, the ideal component is not available and the messages exchanged are different, but all honest parties and the adversary again produce some outputs. The protocol is secure if any outputs produced by the real protocol could also have been produced by the ideal protocol — for any real adversary, there is an ideal adversary, so that their outputs look the same in some sense, even when taking into account the outputs by honest parties. We formalize these notions below.

Definition 3 *The* ideal component *for securely computing the multiparty functionality* f *by n parties is an interactive Turing machine* $\mathcal{F}_{\mathsf{SMC}}^{f}$ *that works as follows:*

- *On input* x_i *from the i-th party, where* x_i *has the correct type (i.e. it maps* $T_{\text{in}}^{-1}(P_i)$ *to the ring R), it stores* x_i *and ignores further inputs from the i-th party.*
- *After receiving* $x_1, \ldots, x_n$*, it computes* $\mathbf{z} = f(\mathbf{x})$*, where* $\mathbf{z}$ *is a mapping from* V_{out} *to R. Let* z_i *be the restriction of* $\mathbf{z}$ *to* $T_{\text{out}}^{-1}(P_i)$*.*
- *For all i, the machine* $\mathcal{F}_{\mathsf{SMC}}^{f}$ *sends* z_i *to the i-th party.*

Definition 4 *Let* $\mathbf{P}_c \subset \mathbf{P}$ *be the set of corrupted parties, S the adversarial Turing machine controlling them,* $f = (C, T_{\text{in}}, T_{\text{out}})$ *a multiparty computation task for the set of parties* $\mathbf{P}$*, and* $\mathbf{x}$ *a possible input for this task. The* ideal-model outcome *of computing* f *on* $\mathbf{x}$ *with corrupted parties* $\mathbf{P}_c$ *and adversary S is a probability distribution* $\textsc{Ideal}_{f,S}^{\mathbf{P}_c}(\mathbf{x})$ *sampled as follows:*

1. *Send* $\mathbf{x}[\mathbf{P}_c]$ *to S. The adversary S returns* $\mathbf{y}$ *— a mapping from* $T_{\text{in}}^{-1}(\mathbf{P}_c)$ *to R, specifying the actual inputs of corrupted parties to* $\mathcal{F}_{\mathsf{SMC}}^f$ *computing* f.
2. *Each honest party* P_i *sends* x_i *to* $\mathcal{F}_{\mathsf{SMC}}^f$. *Each corrupted party* P_i *sends* y_i *to* $\mathcal{F}_{\mathsf{SMC}}^f$. *Each party* P_i *receives* z_i *from* $\mathcal{F}_{\mathsf{SMC}}^f$. *Let* $\mathbf{z}_c$ *be the concatenation of the values* z_i *for* $P_i \in \mathbf{P}_c$.
3. *Send* $\mathbf{z}_c$ *to the adversary S. It returns* $(\mathbf{P}_{h'}, r_A)$, *where* $\mathbf{P}_{h'} \subseteq \mathbf{P} \backslash \mathbf{P}_c$ *and* r_A *is a tuple of elements of R.*
4. *For each honest party* $P_i \in \mathbf{P}_{h'}$, *let* r_i *be the tuple of values* $\mathbf{z}(v)$, *where* $(v, P_i) \in T_{\text{out}}$. *For each honest party* $P_i \notin \mathbf{P}_{h'}$, *let* $r_i = \perp$.
5. *Output* r_A *and* r_i *for all honest parties* P_i.

We see that in sampling $\text{IDEAL}_{f,S}^{\mathbf{P}_c}(\mathbf{x})$, the adversary indeed learns only the inputs of corrupted parties. But as the adversary controls these parties, it can somewhat affect the outcome of computing f by choosing corrupted parties' inputs itself. This is a power that we obviously have to tolerate because the real-model adversary can do the same (as specified in Def. 5). The adversary receives the outcome of f for corrupted parties and is able to influence which of the honest parties actually receive their outputs. Again, this corresponds to the capability of the real-model adversary to prematurely stop the execution of the protocol, as defined next.

Definition 5 *Let* $\mathbf{P}_c \subset \mathbf{P} = \{P_1, \ldots, P_n\}$ *be the set of corrupted parties, A the adversarial Turing machine controlling them,* $\Pi = (M_1, \ldots, M_n)$ *a multiparty computation protocol for task* f *(with* M_i *being the machine executed by* P_i*), and* $\mathbf{x}$ *a possible input to the protocol. The* real-model outcome *of executing* Π *on* $\mathbf{x}$ *with corrupted parties* $\mathbf{P}_c$ *and adversary A is a probability distribution* $\text{REAL}_{\Pi,A}^{\mathbf{P}_c}(\mathbf{x})$ *sampled as follows:*

1. *Execute in parallel the machines* $M_i(\mathbf{x}[P_i])$, *computing the messages for parties* $P_i \in \mathbf{P} \backslash \mathbf{P}_c$, *and* $A(\mathbf{x}[P_c])$, *computing the messages for all parties in* $\mathbf{P}_c$. *During the execution, all messages sent to parties in* $\mathbf{P}_c$ *are routed to A, and A is allowed to send messages on behalf of any* $P \in \mathbf{P}_c$. *Let* r_A *be the output of A and* r_i *the output of* M_i. *Messages sent between honest parties are not learned by A.*
2. *Output* r_A *and* r_i *for all honest parties* P_i.

Having defined the outcomes of ideal and real executions, the security of an SMC protocol is straightforward to define. Typically, security is not provided against any set $\mathbf{P}_c$ of corrupted parties, but only against certain coalitions. It is possible to precisely keep track of tolerated coalitions [4], but in this book, we simplify the presentation and only consider *threshold adversaries* that are allowed to corrupt up to t parties for some $t < n$.

Definition 6 *An n-party protocol* Π *for a functionality* f *is a* secure multiparty computation protocol tolerating at most t malicious parties *if for all* $\mathbf{P}_c \subseteq \mathbf{P}$ *with* $|\mathbf{P}_c| \leq t$ *and all adversaries A, there is an adversary S, so that for all possible inputs* $\mathbf{x}$ *to* f,

$$\text{REAL}_{\Pi,A}^{\mathbf{P}_c}(\mathbf{x}) \stackrel{d}{=} \text{IDEAL}_{f,S}^{\mathbf{P}_c}(\mathbf{x}) \quad . \tag{1}$$

The sign $\stackrel{d}{=}$ in Eq. (1) denotes that the two distributions have to be very close to each other. This closeness can be interpreted in different ways.

- One may require the two distributions to be equal.
- In practice, an equally acceptable requirement is the statistical ε-closeness of the two distributions, where ε is an acceptable failure probability (typically around 2^{-80} or less).
- Alternatively, one may require the two distributions to be merely computationally indistinguishable [5], meaning that no efficient (i.e. probabilistic polynomial-time) algorithm can tell them apart with success probability that is non-negligibly better than $1/2$. In this case, we actually have two families of distributions, indexed by the security parameter η determining the length of cryptographic keys etc. in Π. The running time of Π, A and S (as functions of η) must also be polynomial in η. The success probability of the distinguishing algorithm must be at most $1/2+\alpha(\eta)$, where α is a *negligible* function (i.e. $\lim_{\eta\to\infty}\eta^c\cdot\alpha(\eta)=0$ for all c). If cryptographic constructions are part of Π, then this is the natural level of closeness in Eq. (1).

In Def. 6, the adversary is given full control over the parties it controls. In practice, the adversary may be unable to change the execution of these parties, but still be able to observe their internal state and the messages they exchange with other parties. We thus also define security against *semi-honest* parties. We obtain this definition by making the following changes to Def. 4, Def. 5 and Def. 6:

- In step. 1 of Def. 4, the output $\mathbf{y}$ from the adversary S must equal $\mathbf{x}[\mathbf{P}_c]$.
- In step. 3 of Def. 4, the set $\mathbf{P}_{h'}$ must be equal to $\mathbf{P}\backslash\mathbf{P}_c$.
- While participating in the protocol in the first step of Def. 5, the adversary A must use the actual machines M_i (for the parties $P_i\in\mathbf{P}_c$) to compute the messages sent by P_i.
- In Def. 6, modified definitions of ideal- and real-model outcomes must be used.

In literature, malicious parties are also called "active" and semi-honest parties are called "passive". Correspondingly, one speaks about active vs. passive security, and about actively vs. passively secure protocols. These synonyms will also be used interchangably throughout this book.

2. Oblivious Transfer

Oblivious transfer (OT) is a two-party computational task. The inputs from the first party, called the *sender*, are two bit-strings m_0, m_1 of the same length. The input of the second party, called the *receiver* is a bit b. The output of the receiver is m_b, while the sender gets no outputs. Oblivious transfer is used as a sub-protocol in several SMC protocols.

2.1. Basic Construction

The following construction first appeared in [6]. It requires a cyclic group $\mathbb{G}$ where the Diffie-Hellman problem is hard, e.g. the group $\mathbb{Z}_p^*$. Let g be a fixed generator of $\mathbb{G}$. The *computational Diffie-Hellman problem (CDH)* is to construct g^{xy} from g, g^x and g^y for random integers $x,y\in\{0,\ldots,|\mathbb{G}|-1\}$. The possibly easier *decisional Diffie-Hellman problem (DDH)* is to distinguish tuples (g,g^x,g^y,g^{xy}) from tuples (g,g^x,g^y,g^z), again for random integers x,y,z. A problem is hard if no efficient algorithm can solve it with a non-

negligible success probability (in case of decisional problems, with a success probability that is non-negligibly better than $1/2$). Hence, in the definitions of CDH and DDH, group $\mathbb{G}$ actually depends on the security parameter η.

Let H be a cryptographic hash function mapping elements of $\mathbb{G}$ to bit-strings of the length $|m_0| = |m_1|$. For $b \in \{0,1\}$ let $\overline{b} = 1 - b$. The following protocol securely realizes oblivious transfer.

1. The sender generates a random $C \leftarrow \mathbb{G}$ and sends it to the receiver.
2. The receiver picks $a \leftarrow \{0,\ldots,|\mathbb{G}|-1\}$, sets $h_b = g^a$ and $h_{\overline{b}} = C \cdot h_b^{-1}$. It sends h_0, h_1 to the sender.
3. The sender checks that $h_0 h_1 = C$. If not, it aborts the protocol. Otherwise, it generates $r_0, r_1 \stackrel{\$}{\leftarrow} G$ and sends g^{r_0}, g^{r_1}, $c_0 = H(h_0^{r_0}) \oplus m_0$ and $c_1 = H(h_1^{r_1}) \oplus m_1$ to the receiver.
4. The receiver computes $m_b = c_b \oplus H((g^{r_b})^a)$.

We see that in this protocol, the sender basically treats h_0 and h_1 as public keys for ElGamal encryption [7]. It encrypts m_0 with h_0 and m_1 with h_1. The receiver is able to decrypt under one of the keys, but not under the other one. The protocol is information-theoretically secure against the sender (even if it is malicious), because (h_0, h_1) is uniformly distributed among the pairs of elements of $\mathbb{G}$ whose product is C. Security against a semi-honest receiver follows from the hardness of the DDH problem. In general, security against a malicious receiver is difficult to prove. However, it will follow from CDH if the hash function H is assumed to be a random oracle [8], i.e. $H(x)$ is a random bit-string independent of any other $H(x')$ (or several of them).

We see that that the computational complexity of this OT construction is similar to public-key operations. Indeed, as key exchange can be built on OT [9], it is unlikely that it could be implemented with cheaper symmetric-key primitives only [10].

2.2. Random Oblivious Transfer

Random oblivious transfer (ROT) is a variation of OT that we present here for the benefit of Sec. 2.3 on increasing the practicality of OT. It is a randomized two-party task, where neither the sender nor the receiver input anything, the sender obtains two uniformly, independently sampled random messages r_0, r_1 of predetermined length, and the receiver obtains a random bit b and the message r_b.

Clearly, with the help of OT we can build ROT — the sender and the receiver will just run OT with random inputs. We can also use ROT to build OT as follows. Let the sender have two messages m_0 and m_1, and the receiver have the bit b.

1. The sender and the receiver run ROT, with the sender receiving r_0, r_1 and the receiver receiving b' and $r_{b'}$.
2. The receiver sends $c = b \oplus b'$ to the sender.
3. The sender sends $m'_0 = m_0 \oplus r_c$ and $m'_1 = m_1 \oplus r_{\overline{c}}$ to the receiver.
4. The receiver computes $m_b = m'_b \oplus r_{b'}$.

Indeed, $m'_b \oplus r_{b'} = m_b \oplus r_{b \oplus c} \oplus r_{b'} = m_b$. The protocol is secure for the receiver, because c is independent of b. It is also secure for the sender, because $m_{\overline{b}}$ is masked by $r_{\overline{b'}}$, which the receiver does not have. If the ROT protocol is secure against malicious adversaries, then the resulting OT protocol also has the same security level.

2.3. Extending Oblivious Transfers

Even though public-key encryption is expensive, it is widely used through the hybrid mechanism — to encrypt a long message m, generate a symmetric key k, encrypt m under k, and k (which is much shorter) under the public-key encryption primitive. As we show next, similar constructions exist for OT — a small number of OT instances can be converted into a large number of OT instances with only the help of symmetric-key cryptography.

First, an OT instance for transferring a short message from the sender to the receiver can be converted into an OT instance for large messages by considering these short messages as keys that encrypt real messages. If the sender has two long messages m_0 and m_1, and the receiver has a bit b, then the sender may generate two keys k_0, k_1, send $\mathcal{E}nc(k_0, m_0)$ and $\mathcal{E}nc(k_1, m_1)$ to the receiver, and use OT to transfer k_b to the receiver. Second, m OT instances for messages of the length n, with $m \ll n$, can be converted into n ROT instances for messages of the length m, as we show next [11]. Such an OT extension construction is the main tool to make OT-s practicable in various protocols. The construction where $s[i]$ denotes the i-th bit of the bit-string s, is the following:

1. The receiver randomly generates messages $r_0^1, \ldots, r_0^m, c$ of the length n. It defines $r_1^i = r_0^i \oplus c$ for all $i \in \{1, \ldots, m\}$.
2. The sender generates a bit-string b of the length m.
3. The receiver and the sender use m instances of OT (with roles reversed) to transfer $q^i = r^i_{b[i]}$ from the receiver to the sender, where $i \in \{1, \ldots, m\}$.
4. For each $j \in \{1, \ldots, n\}$, the sender defines the m-bit string s_0^j as consisting of the bits of $q^1[j], \ldots, q^m[j]$. It also defines $s_1^j = s_0^j \oplus b$.
5. For each $j \in \{1, \ldots, n\}$, the receiver defines the m-bit string s^j as consisting of bits $r_0^1[j], \ldots, r_0^m[j]$.
6. In the j-th instance of ROT, the output to the sender is $H(j, s_0^j), H(j, s_1^j)$, and the output to the receiver is $c[j], H(j, s^j)$.

Here, H is a cryptographic hash function from pairs of integers and m-bit strings to m-bit strings. Indeed, one may not simply return s_0^j, s_1^j to the sender and s^j to the receiver, because they satisfy $s_0^j \oplus s_1^j = s_0^{j'} \oplus s_1^{j'}$ for all j, j'. The hash function H is used to break this correlation between different pairs (s_0^j, s_1^j).

The functionality of the construction is easy to verify and its security can be proved if H is modeled as a random oracle. However, the full power of the random oracle is not needed to break the correlations. The authors of the construction introduce the notion of *correlation-robust hash functions* [11] and show that this is sufficient for security.

If the underlying OT protocol is secure against malicious adversaries, then the presented ROT construction is also secure against a malicious sender. But it is only secure against a semi-honest receiver, because of the need to maintain the relationship $r_0^i \oplus r_1^i = c$ for all i. If $r_0^i \oplus r_1^i$ can take different values for different i-s, then the receiver may be able to learn both of the sender's messages. Security against a malicious receiver can be obtained through the following *cut-and-choose* technique [12]. Here, σ is a statistical security parameter, which affects the complexity of the construction and the sucess probability of a cheating receiver.

1. Run σ copies of the previous construction.

2. Let the sender randomly choose $\sigma/2$ of these copies. In these, the receiver reveals to the sender all the messages it has generated. If they are not consistent, the sender aborts. Otherwise, the messages in these $\sigma/2$ copies are discarded and the runs of the other $\sigma/2$ copies are combined as described in the steps below.
3. The receiver randomly picks a bit-string c' of the length n. The bits of c' are the choice bits of the receiver in n instances of ROT.
4. For each $j \in \{1,\ldots,n\}$ and for each of the $\sigma/2$ runs still in use, the receiver tells the sender whether the bits $c'[j]$ and $c[j]$ (in this copy) were the same. If they were not, then the sender swaps s_0^j and s_1^j in this copy.
5. In each instance of ROT, the two messages output to the sender and the message output to the receiver are exclusive ORs of the same messages in each of the $\sigma/2$ copies still in use.

We can again verify that the construction is functional. Its security is based on the use of exclusive OR in combining the non-discarded runs: if in at least one of them, the receiver cannot know both messages to the sender, then it cannot know them in the combined execution either. The probability that the check in step 2 is passed, but no honestly generated runs remain afterwards, is at most $2^{-\sigma/2}$.

3. The GMW Protocol

Goldreich et al. [13] proposed one of the first SMC protocols. Let the multiparty computation task f for n parties be given as a Boolean circuit, where the possible operations are exclusive OR, conjunction, and passing constants. The protocol evaluates the circuit gate by gate, representing the value computed at each gate in a privacy-preserving manner and invoking subprotocols to construct the representation of the result of a gate from the representations of its inputs. In this protocol, the representation $[\![b]\!]$ of a Boolean value $b \in \mathbb{Z}_2$ consists of n Boolean values $[\![b]\!]_1,\ldots,[\![b]\!]_n$ satisfying $[\![b]\!]_1 \oplus \cdots \oplus [\![b]\!]_n = b$. The component $[\![b]\!]_i$ is known to party P_i. The protocol works as follows:

Inputs If party P_i provides an input x for the input vertex v, it will randomly generate $b_1,\ldots,b_{n-1} \stackrel{\$}{\leftarrow} \mathbb{Z}_2$ and define $b_n = b_1 \oplus \cdots \oplus b_{n-1} \oplus x$. It sends b_j to party P_j, which will use that value as $[\![x]\!]_j$.

Constants A constant c computed by a nullary gate is represented as $[\![c]\!] = (c,0,0,\ldots,0)$.

Addition If the result x of gate v is computed as $x = y_1 \oplus y_2$ for some y_1 and y_2 computed in gates v_1 and v_2, and the representations $[\![y_1]\!]$ and $[\![y_2]\!]$ have already been computed, then each party P_i defines $[\![x]\!]_i = [\![y_1]\!]_i \oplus [\![y_2]\!]_i$.

Multiplication If the result x of some gate is computed as $x = y_1 \wedge y_2$, and $[\![y_1]\!]$ and $[\![y_2]\!]$ are already available, then the representation $[\![x]\!]$ is computed as follows. We have $x = \bigoplus_{i=1}^n \bigoplus_{j=1}^n [\![y_1]\!]_i \wedge [\![y_2]\!]_j$. For each $i,j,k \in \{1,\ldots,n\}$, party P_k will learn a value $c_{ijk} \in \mathbb{Z}_2$, such that $\bigoplus_{k=1}^n c_{ijk} = [\![y_1]\!]_i \wedge [\![y_2]\!]_j$. These values are computed as follows:

- If $i = j$, then $c_{iii} = [\![y_1]\!]_i \wedge [\![y_2]\!]_i$ and $c_{iik} = 0$ for $k \neq i$.
- If $i \neq j$ then $c_{iji} \in \mathbb{Z}_2$ is chosen randomly by P_i. Party P_i defines the bits $d_0 = c_{iji}$ and $d_1 = [\![y_1]\!]_i \oplus c_{iji}$. Parties P_i and P_j use oblivious transfer to send $d_{[\![y_2]\!]_j}$ to P_j; this value is taken to be c_{ijj}. If $k \notin \{i,j\}$, then $c_{ijk} = 0$.

Afterwards, each party P_k defines $[\![x]\!]_k = \bigoplus_{i=1}^{n} \bigoplus_{j=1}^{n} c_{ijk}$.

Outputs If party P_i is expected to learn the value x computed in some gate v, and $[\![x]\!]$ has already been computed, then each party P_j sends $[\![x]\!]_j$ to P_i. Party P_i will output $x = [\![x]\!]_1 \oplus \cdots \oplus [\![x]\!]_n$.

It is not difficult to verify that the protocol correctly computes f. The protocol is secure against a passive adversary that controls an arbitrary number (i.e. up to $n-1$) of parties, if the protocol used for oblivious transfer is secure against passive adversaries. Indeed, as long as the adversary does not know all the components of the representation $[\![x]\!]$, and if each component of this representation is distributed uniformly, then the adversary has no idea about the actual value of x. Apart from the sharing of inputs, the only place in the protocol where an adversarially controlled P_j may receive a message from an honest P_i is during oblivious transfer, where P_j learns c_{ijj}. This value is masked by a freshly generated c_{iji}. Hence the view of the adversary can be simulated by generating random bits for all messages that adversarially controlled parties receive.

The protocol is actually more general than presented above. In addition to exclusive OR and conjunction, all other binary Boolean operations can be handled in a manner similar to the multiplication protocol. Indeed, in this protocol, party P_i defines the bits $d_b = c_{iji} \oplus [\![y_1]\!]_i \wedge b$ for $b \in \{0,1\}$. Party P_j receives the bit that corresponds to $b = [\![y_2]\!]_j$. Instead of conjunction, any other operation can be used to compute d_b.

The protocol is not secure against malicious adversaries, because there are no checks to ensure that the parties are behaving according to the protocol. A generic way to achieve security is to use zero-knowledge proofs [5] to show that the protocol is being followed [1, Chapter 7.4]. Due to the high cost of these proofs, they are not used in practice.

4. Secure Multiparty Computation Based on Garbled Circuits

Garbled circuits [14] present a different approach to two-party SMC. Let $f = (C, T_{\text{in}}, T_{\text{out}})$ be a two-party computation task where $C = (G, V_{\text{in}}, V_{\text{out}}, \lambda)$ is a Boolean circuit and $G = (V, E)$. Without a loss of generality, assume that the vertices in V_{out} have no successors. Also assume that only party P_2 gets outputs (i.e. $T_{\text{out}} = V_{\text{out}} \times \{P_2\}$). We discuss later, how a (private) output to P_1 can be provided. In its most basic form, a *garbling* of C is the result of the following steps:

1. For each $v \in V$, generate two keys k_v^0 and k_v^1 (for a chosen symmetric-key encryption scheme).
2. Let $w \in V \setminus V_{\text{in}}$. Let $u, v \in V$ be the two predecessors of w (in this order). Let $\otimes$ be the operation $\lambda(w)$ of w. Let g_w denote the following *garbling* of w: a random permutation of the four ciphertexts $\mathcal{E}nc(k_u^0, \mathcal{E}nc(k_v^0, k_w^{0\otimes 0}))$, $\mathcal{E}nc(k_u^0, \mathcal{E}nc(k_v^1, k_w^{0\otimes 1}))$, $\mathcal{E}nc(k_u^1, \mathcal{E}nc(k_v^0, k_w^{1\otimes 0}))$, and $\mathcal{E}nc(k_u^1, \mathcal{E}nc(k_v^1, k_w^{1\otimes 1}))$.
3. Let $v \in V_{\text{out}}$. The *output garbling* g_v^{out} of v is a random permutation of the two ciphertexts $\mathcal{E}nc(k_v^0, 0)$ and $\mathcal{E}nc(k_v^1, 1)$.
4. Output keys k_v^0 and k_v^1 for $v \in V_{\text{in}}$, and all garblings and output garblings constructed in the previous steps.

To compute f in a privacy-preserving manner, party P_1 garbles circuit C and sends all garblings and output garblings of vertices to P_2. For each $v \in V_{\text{in}}$, where $T_{\text{in}}(v) = P_1$,

party P_1 also sends the key k_v^b to P_2, corresponding to the input b of P_1 to vertex v. For each $v \in V_{\text{in}}$, where $T_{\text{in}} = P_2$, parties P_1 and P_2 use oblivious transfer to transmit k_v^b to P_2, corresponding to the input b of P_2 to vertex v. Party P_2 will then *evaluate* the garbled circuit — going through the vertices of C in topological order, it attempts to decrypt the ciphertexts in the garbling g_v of each vertex v, using the keys it learned while processing the ancestors of v. Assuming that P_2 recognizes when decryption fails, it finds that it cannot decrypt two out of four ciphertexts, can remove one layer of encryption for one ciphertext, and can remove both layers of encryption for one ciphertext. Hence P_2 learns one key while processing g_v. This key is equal to k_v^b for the bit b that would have been computed in vertex v if f had been executed in the clear, but P_2 does not know whether this key is k_v^0 or k_v^1. Similarly, for output garblings g_v^{out}, party P_2 attempts to decrypt the two ciphertexts using the key it learned at v. One of the decryptions is successful and results in a bit that P_2 takes as the result from gate v.

This protocol provides security against semi-honest adversaries (that have corrupted one party). It is quite clearly secure for P_2 if the used OT protocol is secure, as P_2 only interacts with P_1 through that protocol. Security for P_1 follows from the security properties of the encryption scheme, from the inability of P_2 to obtain both keys of any gate, and from its inability to find out whether the key it has corresponds to bit 0 or 1 [15].

The functionality f can be modified and the protocol slightly extended to provide output to P_1 as well. If $f^{\#}$ is the functionality we want to compute, giving $f_1^{\#}(x_1,x_2)$ to P_1 and $f_2^{\#}(x_1,x_2)$ to P_2 (where x_i is the input from P_i), then we let f provide the output $(f_1^{\#}(x_1,x_2) \oplus r, f_2^{\#}(x_1,x_2))$ to P_2 on inputs (x_1,r) from P_1 and x_2 from P_2. When invoking the secure two-party protocol for f, party P_1 lets r be a random bit-string. After executing the protocol, P_2 sends $f_1^{\#}(x_1,x_2) \oplus r$ back to P_1 who unmasks it.

There are a number of optimizations that reduce the cryptographic load of the garbled circuit construction. Instead of using an encryption scheme secure against chosen-plaintext attacks (as our security arguments in previous paragraphs tacitly assumed), the construction of garbling can be modified so that a block cipher, or even just a pseudorandom generator is used [16]. In case of a block cipher, it is possible to do all encryptions with a single key [17], meaning that if any standard cipher (e.g. AES) is used, the key expansion must be done only once. In addition, it is possible to construct the garblings so that the evaluator knows which ciphertext out of the four possible ones it has to decrypt.

More substantial optimizations can significantly reduce the effort of garbling and evaluating the circuit. When using the *free-XOR* technique [18], the garbler first selects a random and private bit-string R of the same length as the keys. It will then select the keys k_v^0 and k_v^1 for each non-XOR gate v so that $k_v^0 \oplus k_v^1 = R$. For each XOR-gate w with inputs u and v, it defines $k_w^0 = k_u^0 \oplus k_v^0$ and $k_w^1 = k_w^0 \oplus R$. In this way, $k_u^b \oplus k_v^c = k_w^{b \oplus c}$ for all $b,c \in \{0,1\}$. The non-XOR gates are garbled as usual. No effort has to be made to garble the XOR-gates. Similarly, the evaluator only has to compute a single exclusive OR of bit-strings in order to evaluate a garbled XOR-gate. When using the free-XOR technique, one attempts to minimize not the size of the entire circuit, but the number of the non-XOR gates in it.

If we use block ciphers in garbling the circuit, then we can choose the keys k_w^0, k_w^1 for some gate w so that one of the ciphertexts in the garbling of w is a constant ciphertext, e.g. $\mathbf{0}$. If u and v are the input gates of w and $\otimes$ is the operation of w, then we can randomly choose $b_u, b_v \in \{0,1\}$ and define $k_w^{b_u \otimes b_v} = \mathcal{D}ec(k_v^{b_v}, \mathcal{D}ec(k_u^{b_u}, \mathbf{0}))$. The ciphertext $\mathbf{0}$ does not have to be sent from the garbler to the evaluator, reducing the

communication complexity by 25% [19]. This technique is compatible with the free-XOR technique described above. A different technique allows eliminating two out of four elements of the garbling of an AND-gate, still keeping the compatibility with free XORs [20].

It is possible to make garbled circuits secure against malicious adversaries. Again, cut-and-choose techniques can be used. To make sure that P_1 actually garbles the circuit that both parties have agreed to evaluate, it is going to garble the same circuit not just once, but σ times for a statistical security parameter σ. Party P_2 selects $\sigma/2$ out of them, and P_1 hands over all randomness that was used to produce these garblings of the circuit for f. After checking the validity of the opened garbled circuits, party P_2 executes the other $\sigma/2$ of the garbled circuits and takes the majority of their results as the final result. It is important to combine the results from different circuits using majority, instead of failing if P_2 receives several different results from the $\sigma/2$ circuits it is executing, as the fact whether or not P_2 has failed can give one bit of information about the inputs of P_2 to a malicious P_1.

Using the cut-and-choose technique introduces further complications relating to the preparation of inputs. Namely, P_2 has to make sure that the keys it receives are valid (and that complaints about invalidity do not leak information to P_1), correspond to its inputs, and to the same inputs of P_1 in all garbled circuits. We refer to [21] for details.

If P_2 is malicious, then the output of P_1 obviously cannot be simply masked with a random r as P_2 could modify it afterwards. Instead, the original functionality $f^{\#}$ is modified to compute an authenticated encryption [22] of the output of P_1, using a key that is part of the input of P_1 to the circuit. This encryption is learned by P_2 and sent back to P_1 who verifies its integrity and decrypts it.

5. Secure Multiparty Computation Based on Shamir's Secret Sharing

A *secret sharing* scheme allows a value to be shared among n parties so that certain coalitions of them can recover it from their shares, and certain other, smaller coalitions obtain no information about that value from their shares. Most frequently, there is a threshold t, so that all the coalitions with a size of at least t can find the value, and no coalition smaller than t gets any information. We will explore this case. Secret sharing is relevant for SMC because a number of operations can be performed on secret-shared values without leaking any further information about the values themselves to small coalitions.

Shamir's secret sharing scheme [23] is based on polynomial interpolation. Let $\mathbb{F}$ be a field with at least $n+1$ elements. Let $c_1,\ldots,c_n$ be mutually different, non-zero elements of $\mathbb{F}$. If a *dealer* wishes to share a value v among the parties $P_1,\ldots,P_n$, it will randomly generate a polynomial f of a degree of $t-1$ at most, satisfying $f(0)=v$, and send $s_i=f(c_i)$ to party P_i. The polynomial is generated by randomly generating $a_1,\ldots,a_{t-1} \stackrel{\$}{\leftarrow} \mathbb{F}$ and defining $f(x)=v+a_1x+a_2x^2+\cdots+a_{t-1}x^{t-1}$.

Polynomials over fields can be interpolated: for any t points (i.e. argument-value pairs), there is exactly one polynomial of a degree of $t-1$ at most that passes through these points. For a fixed set of arguments, the coefficients of this polynomial can be computed as linear combinations of the values of the polynomial. In particular, using the notation of the previous paragraph, for each $\mathbf{I}=\{i_1,\ldots,i_t\}$ of the size t, there are coefficients $\lambda^{\mathbf{I}}_{i_1},\ldots,\lambda^{\mathbf{I}}_{i_t}$, so that $v=\sum_{j\in\mathbf{I}}\lambda^{\mathbf{I}}_j s_j$. Using this equality, any t parties can recover

the secret. On the other hand, fewer than t parties obtain no information at all about the secret. If $\mathbf{I}' \subseteq \{1,\ldots,n\}$ and $|\mathbf{I}'| < t$, then for any v' there is a polynomial of a degree of $t-1$ at most that passes through the points $\{(i,s_i)\}_{i\in\mathbf{I}'}$ and $(0,v')$. Moreover, for each $v' \in \mathbb{F}$, the number of such polynomials is the same.

It is possible to perform computations with shared values. If $s_1^1,\ldots,s_n^1$ are shares for v_1 (using polynomial f_1), and $s_1^2,\ldots,s_n^2$ are shares for v_2 (using polynomial f_2), then $s_1^1+s_1^2,\ldots,s_n^1+s_n^2$ are shares for value v_1+v_2 (using polynomial f_1+f_2). Indeed, adding points corresponds to adding polynomials. If the degrees of f_1 and f_2 are at most $t-1$, then the degree of f_1+f_2 is also at most $t-1$. Similarly, the value cv for a public $c \in \mathbb{F}$ is represented by the shares $cs_1,\ldots,cs_n$. Hence, linear combinations of shared values can be computed by computing the same linear combinations of shares.

The multiplication of shared values is also possible through a protocol between $P_1,\ldots,P_n$. The following protocol [24] requires $2t-1 \le n$. Using the same notation as in the previous paragraph, let $f = f_1 \cdot f_2$. The degree of f is at most $2t-2$. Party P_i computes $f(c_i) = s_i^1 \cdot s_i^2$ and shares it among all n parties using a polynomial of a degree of $t-1$ at most. If all parties do it, they have available the shared values of $f(c_1),\ldots,f(c_n)$. As $n \ge 2t-1$, the value $vv' = f(0)$ is a linear combination of these shared values. Each party computes this linear combination on its shares, resulting in a sharing of vv'.

The described protocols are constituents of an SMC protocol among n parties, secure against a passive adversary corrupting at most $t-1$ parties, where $2t-1 \le n$. In other words, the number of corrupted parties must be smaller than $n/2$. We can securely evaluate arithmetic circuits (in which the operations are constants, additions, multiplications with a constant, and multiplications) over a finite field $\mathbb{F}$. Similarly to Sec. 3, the circuit is evaluated gate-by-gate and the result $v \in \mathbb{F}$ of a gate is represented as $[\![v]\!] = ([\![v]\!]_1,\ldots,[\![v]\!]_n)$, so that $[\![v]\!]_1,\ldots,[\![v]\!]_n$ are a Shamir's secret sharing of v secure against $t-1$ parties. The protocol works as follows:

Inputs A party P_i providing an input v to some vertex will secret-share v among all n parties (including itself).

Constants The constant $c \in \mathbb{F}$ is represented by shares $(c,c,\ldots,c)$. Indeed, these are the values of the constant polynomial $f(x) = c$ (of degree 0, i.e. at most $t-1$) at the points $c_1,\ldots,c_n$.

Arithmetic operations The representation $[\![v]\!]$ of a value v computed in a gate for addition, multiplication with a constant, or multiplication is computed from the representations of the values in the predecessors of that gate, using the protocols described in this section.

Outputs If the party P_j is expected to learn a value v computed in some gate, then each party P_i sends $[\![v]\!]_i$ to P_j. The party P_j uses interpolation to compute v.

The protocol is secure against passive adversaries. A simulator is easy to construct after noting that a coalition of parties of the size of $t-1$ or less only ever sees random values of $\mathbb{F}$ as incoming messages. But a malicious party P_i can cheat in the following places:

- When sharing a value, it can pick a polynomial of degree t or more.
- In the multiplication protocol, it can incorrectly compute $s_i^1 \cdot s_i^2$, where s_i^1 and s_i^2 are its shares of the values that are multiplied together.

- When sending its share to another party (which is supposed to learn the value in some gate), it can send a wrong share.

The first and last issue are solved by using *verifiable secret sharing (VSS)* [25,26,27]. If the VSS scheme has the necessary homomorphic properties, then it can be used to solve the second issue as well. Next, we will describe Pedersen's VSS scheme [25], which is conceptually simple, although not the most efficient. It can be used to make the described SMC protocol secure against malicious adversaries.

Let a *broadcast channel* be available. All parties receive the same values over this channel, even if they were broadcast by a malicious party. A broadcast channel can be built with the help of digital signatures. Let $\mathbb{F} = \mathbb{Z}_p$ for some prime number p and let $\mathbb{G}$ be a group where the discrete logarithm problem is hard, and $|\mathbb{G}| = p$. Let $g, h \in \mathbb{G}$, so that $C = \log_g h$ would be unknown to all parties. Such g and h are easy to generate: each party P_i broadcasts a random $g_i \in \mathbb{G}$ and $h_i \in \mathbb{G}$; and g and h are defined as the products of those.

When using Pedersen's VSS, a dealer computes the sharing $[\![v]\!]$ of a value $v \in \mathbb{Z}_p$ among the parties $P_1, \ldots, P_n$ in the following manner:

- Randomly generate $a_1, \ldots, a_{t-1}, a'_0, \ldots, a'_{t-1} \stackrel{\$}{\leftarrow} \mathbb{Z}_p$. Let $a_0 = v$. Define $f(x) = v + a_1 x + \cdots + a_{t-1} x^{t-1}$ and $f'(x) = a'_0 + a'_1 x + \cdots + a'_{t-1} x^{t-1}$.
- Broadcast $y_j = g^{a_j} h^{a'_j}$ for all $j \in \{0, \ldots, t-1\}$.
- Send $(f(c_i), f'(c_i))$ to party P_i.

The party P_i, having received (s_i, s'_i) can verify its share with respect to the published values. The following equation must hold:

$$g^{s_i} h^{s'_i} = \prod_{j=0}^{t-1} y_j^{c_i^j} \quad . \tag{2}$$

Indeed, if the share (s_i, s'_i) corresponds to the broadcast values, then taking the logarithm (to the base of g) of the sides of this equation gives us $s_i + C s'_i = f(c_i) + C f'(c_i)$. If the dealer were able to produce a different set of broadcast values that would still pass verification, it could use them to recover C. On the other hand, the broadcast values do not reveal anything further about the value v. In fact, each y_j is a random element of $\mathbb{G}$ as it has been masked with $h^{a'_j}$.

We now show how the issues defined above can be solved with Pedersen's VSS.

- *Ensure that the polynomial degree is at most $t-1$.* By construction, the logarithm of $\prod_{j=0}^{t-1} y_j^{c_i^j}$ is equal to some polynomial g of degree $t-1$ evaluated on c_i. However, as we have shown above, the same logarithm equals $f(c_i) + C f'(c_i)$. We need to show that the degree of f is at most $t-1$ in this case. If $\deg(f(c_i) + C f'(c_i)) = d_1 \leq t-1$, but $\deg(f(c_i)) = d_2 > t-1$, then the last $d_2 - d_1$ coefficients of the polynomials f and Cf' should be negations of each other. The coefficients of f and f' are known to the sender and, hence, it may compute $C = -a_i/a'_i$ for some $i \in \{d_1, \ldots, d_2\}$.
- *Multiplication protocol.* In the particular multiplication protocol presented above, each party computes $s = s^1 \cdot s^2$ and then shares it amongst all the n parties. The party that shares the product $s = s^1 \cdot s^2$ can commit the shares s_i of s exactly in

the same way as the input is committed, publishing $(g^{s_i}, h^{s'_i})$. Assuming that the shares of s^1 and s^2 have already been committed in the same way, the party has to prove that $s = s^1 \cdot s^2$.

A linear combination can be computed locally by any party. The commitment for $\alpha_1 s_1 + \ldots + \alpha_m s_m$ is $(g^{\alpha_1 s_1 + \ldots + \alpha_m s_m}, h^{\alpha_1 s'_1 + \ldots + \alpha_m s'_m}) = ((g^{s_1})^{\alpha_1} \cdot \ldots \cdot (g^{s_m})^{\alpha_m}, (h^{s'_1})^{\alpha_1} \cdot \ldots \cdot (h^{s'_m})^{\alpha_m})$, where $(g^{s_i}, h^{s'_i})$ have already been committed. Since s is a linear combination of s_i, the parties are able to compute $(g^s, h^{s'})$. Similarly, they compute $(g^{s^1}, h^{s'^1})$ and $(g^{s^2}, h^{s'^2})$. Now there are different ways of verifying $s = s^1 \cdot s^2$.

- Verifying a multiplication is easy if the homomorphic scheme allows verifying products (there is an operation $\odot$ so that $E(x) \odot E(y) = E(x \cdot y)$ for a commitment function E). Such schemes exist, but they are not very efficent in practice (see Sec. 6 for details).
- If the group $\mathbb{G}$ supports a *bilinear pairing* (a function $e : \mathbb{G} \times \mathbb{G} \mapsto \mathbb{G}$ so that $e(g^a, g^b) = e(g,g)^{ab}$ for all $g \in \mathbb{G}$, $a, b \in \mathbb{Z}$), then the product $s = s^1 \cdot s^2$ can be verified as $e(g^{s^1}, g^{s^2}) = e(g,g)^{s^1 s^2}$.
- *Multiplication triples* can be used to reduce all the multiplications to linear combinations. This is described in more detail in Sec. 7.
- The most general method for verifying multiplications is to use any zero-knowledge proofs of knowing $\bar{s}^2$, so that $g^{\bar{s}^2} = g^{s^2}$ and $(g^{s^1})^{\bar{s}^2} = g^{s^1 s^2}$. Here, $g^{\bar{s}^2} = g^{s^2}$ proves that $\bar{s}^2 = s^2$ as a discrete logarithm is unique in the group $\mathbb{G}$.

- *Verifying a share sent by another party.* Each secret input s is committed as $(g^s, h^{s'})$, where s' is the leading coefficient of the polynomial f'. Each intermediate value is either a linear combination or a product of the previous values. These operations can be verified as shown in the previous clause.

6. Secure Multiparty Computation Based on Threshold Homomorphic Encryption

Threshold homomorphic encryption is another method that allows revealing a secret value to any coalition of at least t parties, while giving no information about that value to any coalition smaller than t.

In a (t,n)-threshold homomorphic cryptosystem, anybody can perform the following operations using the public key:

- Encrypt a plaintext.
- Add (denoted $\oplus$) two ciphertexts to obtain a (uniquely determined) encryption of the sum of the corresponding plaintexts.
- Multiply (denoted $\otimes$) a ciphertext by a constant to obtain a (uniquely determined) encryption of the product of the plaintext with the constant.

Decryption is only possible if at least t out of the n decryption keys are known. An example of such a cryptosystem is the threshold variant of the Paillier cryptosystem from [28]; see [29] for details.

Computation on homomorphic encryptions can be performed by the n parties holding the decryption keys for the cryptosystem. Additions and multiplications by a constant can be performed locally using the homomorphic operators $\oplus$ and $\otimes$. Multiplications of

encryptions X of x and Y of y can be performed using an interactive protocol between the n parties due to [30]. In this protocol, each party i chooses a random value d_i, and broadcasts encryptions D_i of d_i and $E_i = Y \otimes (-d_i)$ of $y \cdot (-d_i)$. The parties then compute $X \oplus D_1 \oplus \cdots \oplus D_n$, and threshold decrypt it to learn $s = x + d_1 + \ldots + d_n$. This allows them to compute the encryption $Z = Y \otimes (x + d_1 + \cdots + d_n)$ of $y \cdot (x + d_1 + \cdots + d_n)$ and, hence, also an encryption of $x \cdot y$ as $Z \oplus \bigoplus_{i=1}^{n} E_i$.

While computations on homomorphic encryptions are much slower than computations on secret shares, the advantage is that it is easy to make them secure against an active adversary. Namely, in the above multiplication protocol, the parties can use zero-knowledge proofs to prove that each E_i indeed contains the product $y \cdot (-d_i)$ and that their share of the threshold decryption of $X \oplus D_1 \cdots \oplus D_n$ was correct. To perform these proofs, [29] uses a multiparty variant of Σ-protocols. Recall that a Σ-protocol for a binary relation R is a three-move protocol in which a potentially malicious prover convinces an honest verifier that he knows a witness for a certain statement. First, the prover sends an announcement to the verifier. The verifier responds with a uniformly random challenge. Finally, the prover sends its response, which the verifier verifies. In order to let all n parties prove statements to each other simultaneously, they jointly generate a single challenge to which they all respond. Namely, each party broadcasts a commitment to its announcement, the parties jointly generate a challenge, and, finally, the parties broadcast their response to this challenge, along with an opening of their commitment.

Combining these techniques, we get an SMC protocol among n parties, which is secure against an active adversary corrupting at most $t-1$ parties. The protocol securely evaluates arithmetic circuits over the plaintext ring $\mathbb{Z}_N$ of the threshold homomorphic cryptosystem, e.g. N is an RSA modulus in the case of Paillier encryption. The protocol also achieves robustness in the sense that, if at least t parties are honest, then all parties are guaranteed to learn the computation result. In particular, if $t = 1$, then the protocol guarantees privacy and correctness for all honest parties (but no robustness). If $t = \lceil (n+1)/2 \rceil$, then the protocol guarantees privacy, correctness, and robustness if fewer than t parties are corrupted. The full protocol works in the manner described below.

Inputs Party P_i providing an input v broadcasts a homomorphic encryption of v. Each party proves knowledge of the corresponding plaintext. This prevents parties from adaptively choosing their inputs based on the inputs of others.

Constants The constant $c \in \mathbb{Z}_N$ is represented by encryption $C = \mathsf{Enc}(c)$ of c with fixed randomness.

Addition If the result x of a gate is computed as $x = y_1 + y_2$ and encryptions Y_1 and Y_2 have already been computed, then each party defines $X = Y_1 \oplus Y_2$.

Multiplication Multiplication is performed using the interactive n-party protocol described above, i.e. the parties exchange encryptions D_i, E_i, perform an interactive multiplication proof, exchange threshold decryptions, and perform an interactive decryption proof. Finally, they compute the product encryption from the decrypted value and the encryptions E_i.

Outputs If party P_j is expected to learn a value v computed in some gate as encryption V, then it broadcasts an encryption D of a random value d and proves knowledge of the plaintext of D. Then, all parties provide threshold decryptions of $Z = V \oplus D$ and prove correctness of the decryption z. Finally, party P_j computes $v = z - d$.

The homomorphic encryption scheme defined above can be applied *locally* (i.e. without any interaction) to various tasks for which computing the sum and the scalar product of plaintexts is sufficient. In general, it is not applicable to an arbitrary computation. *Fully homomorphic encryption* [31] includes an operation denoted $\odot$ on ciphertexts (in addition to $\oplus$ and $\otimes$ mentioned before) that allows anybody to multiply two ciphertexts to obtain a (uniquely determined) encryption of the product of the corresponding plaintexts. Supporting addition and multiplication is sufficient to perform an arbitrary computation on plaintexts. However, fully homomorphic encryption schemes are much less efficient, and hence, a simpler homomorphic scheme is preferable for a more constrained task that does not require the support of arbitrary computation (such as computing linear combinations). Alternatively, some special multiplication protocols like the one presented in this section can be used.

7. Secure Multiparty Computation Based on Somewhat Homomorphic Encryption

In this section we describe the protocol SPDZ that was initially proposed in [32]. The main idea of this protocol is to introduce an expensive preprocessing phase that allows making the online phase cheap.

The SPDZ protocol makes use of the following techniques:

- *Message authentication codes (MAC)* prevent the shares from being affected by a malicious adversary.
- *Beaver triples* reduce multiplication to linear combinations.
- *Somewhat homomorphic encryption* is used in the preprocessing phase to compute MACs and Beaver triple shares.

Somewhat homomorphic encryption [33] satisfies the properties of homomorphic encryption (defined in Sec. 6). The difference is that it supports only a finite number of sequential multiplications. A group that supports a bilinear pairing (see Sec. 6) is an example of somewhat homomorphic encryption with at most one sequential multiplication. SPDZ uses somewhat homomorphic encryption in the preprocessing phase which can be seen as many smaller arithmetic circuits that are evaluated in parallel, and where the number of sequential multiplications is not large.

Message authentication codes make it difficult for the adversary to modify the messages by introducing special correctness checks that are generated for the inputs and propagated by the operations. SPDZ uses two kinds of additive sharings that allow message checking. The generation of these shares is described in more detail in [32].

1. The first sharing is

$$\langle a\rangle = (\delta, (a_1,\ldots,a_n), (\gamma(a)_1,\ldots,\gamma(a)_n)) \ ,$$

where $a = a_1 + \cdots + a_n$ and $\gamma(a)_1 + \cdots + \gamma(a)_n = \alpha(a+\delta)$. The party P_i holds the pair $(a_i, \gamma(a)_i)$, and δ is public. The interpretation is that $\gamma(a) = \gamma(a)_1 + \cdots + \gamma(a)_n$ is the MAC authenticating the message a under the global key α.
We have $\langle a\rangle + \langle b\rangle = \langle a+b\rangle$ for secret values a and b where $+$ is pointwise addition: $\langle a+b\rangle = (\delta_a + \delta_b, (a_1+b_1,\ldots,a_n+b_n), (\gamma(a)_1 + \gamma(b)_1,\ldots,\gamma(a)_n +$

$\gamma(b)_n)$). If c is a constant, $c+\langle a\rangle = (\delta - c, (a_1+c, a_2, \ldots, a_n), (\gamma(a)_1, \ldots, \gamma(a)_n))$. Hence, any linear combination can be computed locally, directly on the shares.

2. The second sharing is

$$[\![\alpha]\!] = ((\alpha_1, \ldots, \alpha_n), (\beta_1, \ldots, \beta_n), (\gamma(\alpha)^i_1, \ldots, \gamma(\alpha)^i_n)_{i\in[n]}) \ ,$$

where $\alpha = \alpha_1 + \cdots + \alpha_n$ and $\gamma(\alpha)^i_1 + \cdots + \gamma(\alpha)^i_n = \alpha\beta_i$. The party P_i holds the values $\alpha_i, \beta_i, \gamma(\alpha)^1_i, \ldots, \gamma(\alpha)^n_i$. The idea is that $\gamma(\alpha)^i_1 + \cdots + \gamma(\alpha)^i_n$ is the MAC authenticating α under the private key β_i of P_i. To open $[\![\alpha]\!]$, each P_j sends to each P_i its share α_j of α and its share $\gamma(\alpha)^i_j$ of the MAC on α made with the private key of P_i. P_i checks that $\sum_{j\in[n]} \gamma(\alpha)^i_j = \alpha\beta_i$. To open the value to only one party P_i (so-called *partial opening*), the other parties simply send their shares only to P_i, who does the checking. Only shares of α and $\alpha\beta_i$ are needed for that.

Beaver triples [34] are triples of values (a,b,c) over the corresponding finite field $\mathbb{F}$ where the computation takes place, generated by randomly picking $a, b \xleftarrow{\$} \mathbb{F}$ and computing $c = a \cdot b$. Precomputing such triples can be used to linearize multiplications. This may improve the efficiency of the online protocol phase significantly, as computing a linear combination can be done locally. In order to compute $\langle x\rangle \cdot \langle y\rangle$, if a triple $(\langle a\rangle, \langle b\rangle, \langle c\rangle)$ has been precomputed and shared already, we may first compute and publish $x' := \langle x\rangle - \langle a\rangle$ and $y' := \langle y\rangle - \langle b\rangle$, and then compute a linear combination $\langle x*y\rangle = (x'+\langle a\rangle)(y'+\langle b\rangle) = x'y' + y'\langle a\rangle + x'\langle b\rangle + \langle a\rangle\langle b\rangle = x'y' + y'\langle a\rangle + x'\langle b\rangle + \langle c\rangle$. Here, the publication of x' and y' leaks no information about x and y. In this way, interactive multiplication is substituted with an opening of two values. In SPDZ, a sufficient number of such triples is generated in the preprocessing phase. The triples are random and depend neither on the inputs nor on the function that is being computed. The checks performed during preprocessing ensure that all triples used during the online phase are valid.

In the specification of the protocol, it is assumed for simplicity that a broadcast channel is available, that each party has only one input, and only one public output value has to be computed. The number of input and output values can be generalized to an arbitrary number without affecting the overall complexity (as shown in [32]). The protocol works as follows:

Initialization The parties invoke the preprocessing to get:

- The shared secret key $[\![\alpha]\!]$.
- A sufficient number of Beaver triples $(\langle a\rangle, \langle b\rangle, \langle c\rangle)$.
- A sufficient number of pairs of random values $\langle r\rangle$, $[\![r]\!]$.
- A sufficient number of single random values $[\![t]\!]$, $[\![e]\!]$.

The generation of all these values is presented in more detail in [32]. It is based on a somewhat homomorphic encryption scheme.

Inputs If the party P_i provides an input x_i, a pre-shared pair $\langle r\rangle$, $[\![r]\!]$ is taken and the following happens:

1. $[\![r]\!]$ is opened to P_i.
2. P_i broadcasts $x'_i = x_i - r$.
3. The parties compute $\langle x_i\rangle = \langle r\rangle + x'_i$.

Addition In order to add $\langle x\rangle$ and $\langle y\rangle$, locally compute $\langle x+y\rangle = \langle x\rangle + \langle y\rangle$.
Multiplication To multiply $\langle x\rangle$ and $\langle y\rangle$ the parties do the following:

1. Take the two triples $(\langle a\rangle, \langle b\rangle, \langle c\rangle)$, $(\langle f\rangle, \langle g\rangle, \langle h\rangle)$ from the set of the available ones and check that indeed, $a \cdot b = c$. This can be done as follows:

   - Open a random value $[\![t]\!]$.
   - Partially open $a' = t\langle a\rangle - \langle f\rangle$ and $b' = \langle b\rangle - \langle g\rangle$.
   - Evaluate $t\langle c\rangle - \langle h\rangle - b'\langle f\rangle - a'\langle g\rangle - a'b'$ and partially open the result.
   - If the result is not zero the protocol aborts, otherwise go on with $(\langle a\rangle, \langle b\rangle, \langle c\rangle)$.

   The idea is that as t is random, it is difficult for the adversary to generate malicious shares so that the result would be 0. This check can be done as part of the preprocessing for all triples in parallel, and hence, only one random value t is sufficient.
2. Partially open $x' = \langle x\rangle - \langle a\rangle$ and $y' = \langle y\rangle - \langle b\rangle$. Compute $\langle z\rangle = x'y' + x'\langle b\rangle + y'\langle a\rangle + \langle c\rangle$.

Outputs The output stage starts when the parties already have $\langle y\rangle$ for the output value y, but this value has not been opened yet. Before the opening, it should be checked that all parties have behaved honestly.

- Let $a_1, \ldots, a_T$ be all values publicly opened so far, where

$$\langle a_j\rangle = (\delta_j, (a_{j1}, \ldots, a_{jn}), (\gamma(a_j)_1, \ldots, \gamma(a_j)_n)) \ .$$

  A new random value $[\![e]\!]$ is opened, and parties set $e_i = e^i$ for $i \in [T]$. All parties compute $a = \sum_{j\in[T]} e_j a_j$.
- Each P_i commits to $\gamma_i = \sum_{j\in[T]} e_j \gamma(a_j)_i$. For the output value $\langle y\rangle$, P_i also commits to the shares $(y_i, \gamma(y)_i)$ in the corresponding MAC.
- $[\![\alpha]\!]$ is opened.
- Each P_i opens the commitment γ_i, and all parties check that $\alpha(a + \sum_{j\in[T]} e_j\delta_j) = \sum_{i\in[n]} \gamma_i$. If the check is not passed, the protocol aborts. Otherwise, the parties conclude that all the messages a_j are correct.
- To get the output value y, the commitments to $(y_i, \gamma(y)_i)$ are opened. Now y is defined as $y := \sum_{i\in[n]} y_i$, and each player checks that $\alpha(y+\delta) = \sum_{i\in[n]} \gamma(y)_i$. If the check is passed, then y is the output.

This verifies that all the intermediate values a_j, and also y, have indeed all been computed correctly.

8. Universally Composable Secure Multiparty Computation

The security definitions in Sec. 1 consider the case where only a single instance of the SMC protocol is running. This is rarely the case in reality where we want to run several instances, possibly concurrently, and possibly together with other protocols. *Universal composability (UC)* [35] has emerged as the way to define the security of protocols that is preserved under concurrent compositions. In UC definitions of the security of protocols, the ideal and real executions contain one more entity, which is called the *environment* that models the other parts of the system in addition to the protocol.

The ideal-model and real-model executions in Def. 4 and Def. 5 are modified to incorporate the environment $\mathcal{Z}$ that depends on the functionality offered by the protocol. The modifications are the following:

- The parties $P_1, \ldots, P_n$ receive their inputs $x_1, \ldots, x_n$ from $\mathcal{Z}$. They also hand back their outputs to $\mathcal{Z}$.
- During the execution, $\mathcal{Z}$ can freely communicate with the adversary (S or A). This models the adversary's influence on other parts of the system.
- At some moment, the environment $\mathcal{Z}$ stops and outputs a bit. This bit is the result of the execution.

We thus obtain the probability distributions $\textsc{Ideal}^{\mathbf{P}_c}_{f,S}(\mathcal{Z})$ and $\textsc{Real}^{\mathbf{P}_c}_{\Pi,A}(\mathcal{Z})$, both over the set $\{0,1\}$. Intuitively, the bit output by $\mathcal{Z}$ represents its guess whether it participated in an ideal-model or a real-model execution. We can now define the UC security of SMC protocols, following the general pattern of UC definitions where a real system Π is at least as secure as an ideal system $\mathcal{F}$ if anything that can happen to the environment when interacting with the real system can also happen when interacting with the ideal system.

Definition 7 *An n-party protocol Π for a functionality f is* at least as secure as $\mathcal{F}^f_{\mathsf{SMC}}$ *against attackers corrupting at most t parties, if for all $\mathbf{P}_c \subseteq \mathbf{P}$ with $|\mathbf{P}_c| \leq t$ and all adversaries A, there exists an adversary S, so that for all possible environments $\mathcal{Z}$,*

$$\textsc{Real}^{\mathbf{P}_c}_{\Pi,A}(\mathcal{Z}) \stackrel{d}{=} \textsc{Ideal}^{\mathbf{P}_c}_{f,S}(\mathcal{Z}) \ . \tag{3}$$

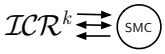
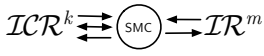
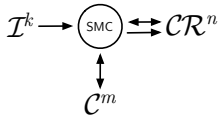
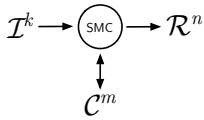
It is known that if several secure protocols (or several instances of the same secure protocol) are run concurrently, the adversary may be able to attack the resulting system as the parties may confuse messages from different sessions [36, Chapter 2]. In practice, this confusion is eliminated by using suitable session identifiers on messages. This in turn requires the parties to somehow agree on these identifiers.

9. Party Roles and Deployment Models of Secure Multiparty Computation

The descriptions of SMC protocols given in previous sections assume that all parties of the SMC protocol provide some inputs to it (possibly trivial ones), participate actively in the computation, and receive some outputs (possibly trivial ones). In practice, the number of involved parties may be large. This happens in particular when there are many parties that provide inputs to the computation. In this case, some of the protocols described before become inefficient, and other cannot be used at all.

To implement large-scale SMC applications with many parties, we break the symmetry among them. We consider three different party roles that define which parties can see what and who is in charge of certain operations. These three roles include *input parties*, who secret-share or encrypt the data they provide to the computation. Each of the protocols described in this chapter has, or can be amended with an input phase that can be executed by anyone without knowledge of any secrets set up to execute the SMC protocol. The input parties send the secret-shared or encrypted values to the *computing parties*, who carry out the SMC protocols on the hidden values. The number of computing parties is kept small in order to efficiently execute the SMC protocol. The computing

Table 1. SMC deployment models and examples of applications

Basic deployment model	Examples of applications
	**The classic millionaires' problem [14]** *Parties:* Two, Alice and Bob (both $\mathcal{ICR}$) *Overview:* The millionaires Alice and Bob use SMC to determine who is richer **Joint genome studies [38]** *Parties:* Any number of biobanks (all $\mathcal{ICR}$) *Overview:* The biobanks use SMC to create a joint genome database and study a larger population
	**Studies on linked databases [37]** *Parties:* The Ministry of Education, the Tax Board, the Population Register (all $\mathcal{IC}$) and the Statistical Office ($\mathcal{R}$) *Overview:* Databases from several government agencies are linked to perform statistical analyses and tests
	**Outsourcing computation to the cloud [31]** *Parties:* Cloud customer ($\mathcal{IR}$) and cloud service providers (all $\mathcal{C}$) *Overview:* The customer deploys SMC on one or more cloud servers to process his/her data
	**Collaborative network anomaly detection [39]** *Parties:* Network administrators (all $\mathcal{IR}$), a subset of whom are running computing servers (all $\mathcal{ICR}$) *Overview:* A group of network administrators use SMC to find anomalies in their traffic
	**The sugar beet auction [40]** *Parties:* Sugar beet growers (all $\mathcal{I}$), Danisco and DKS (both $\mathcal{CR}$) and the SIMAP project ($\mathcal{C}$) *Overview:* Sugar beet growers and their main customer use SMC to agree on a price for purchase contracts
	**The Taulbee survey [41]** *Parties:* Universities in CRA (all $\mathcal{I}$), universities with computing servers (all $\mathcal{IC}$) and the CRA ($\mathcal{R}$) *Overview:* The CRA uses SMC to compute a report of faculty salaries among CRA members **Financial reporting in a consortium [42]** *Parties:* Members of the ITL (all $\mathcal{I}$), Cybernetica, Microlink and Zone Media (all $\mathcal{IC}$) and the ITL board ($\mathcal{R}$) *Overview:* The ITL consortium uses SMC to compute a financial health report on its members

parties send the encrypted or secret-shared results to the *result parties*, who combine the received values in order to see the results. A party can have one or several of these roles. Table 1 from [37] describes several practical prototype applications within the described party role paradigm.

10. Classes of Properties of Secure Multiparty Computation Protocols

In previous sections, we have given examples of different sharing schemes and SMC protocols. We have seen that the protocols can be classified into those secure against a passive adversary and those secure against an active adversary. There are even more protocol classifications, some of which we present in this section. Even more protocol properties can be found in [43].

Trusted setup. Some protocols require pre-sharing of certain information before the start of an execution. This information is independent from the actual protocol inputs on which the functionality is computed. Here we consider pre-sharing that is done in a trusted setup.

A common reference string (CRS) [44] is a polynomial-length string that comes from a certain pre-specified distribution. All the involved parties must have access to the same string. Introducing a CRS makes it possible to remove some interaction from the protocol. For example, the random values that must be generated by one party and sent to another can be pre-shared before the execution starts.

A protocol may use a public key infrastructure (PKI), where a public and a secret key are issued to each party. The PKI can be used for various purposes such as signatures, commitments, and ensuring that only the intended receiver gets the message. Its advantage compared to a CRS is that it can be reused (unless it is used for certain tasks such as bit commitments, where the secret key is revealed), while in general, a CRS cannot be reused and a new instance has to be generated for each protocol run.

If there is no trusted setup, it is still possible to achieve the same properties that the trusted setup gives (for example, include a key exchange subprotocol), at the expense of an online protocol execution phase.

Existence of a broadcast channel. A broadcast channel allows a party to send the same message to all other parties in such a way that each receiver knows that each other (honest) party has received exactly the same message.

If there is no explicit broadcast channel, it can still be modeled in some settings. For example, if at least $2n/3+1$ of the n parties are honest, then a broadcast can be implemented as follows. If P_i wants to broadcast m, it sends (init, i, m) to all other parties. If a party P_j receives (inti, i, m) from P_i, it sends (echo, i, m) to all parties (including itself). If a party P_j receives (echo, i, m) from at least $n/3+1$ different parties, then it sends (echo, i, m) to all parties too. If a party P_j receives (echo, i, m) from at least $2n/3+1$ different parties, then it accepts that P_i has broadcast m. It can be shown that if at least one party accepts m, then all the other honest parties do as well.

Assumption level. The security of protocols can be based on the intractability of certain computational tasks. Some protocols use quite specific assumptions such as factoring or finding the minimal distance of vectors generated by a matrix over a finite field. In some cases, the intractability has not even been formally reduced to well-known open problems. Even if no efficient algorithm for solving these tasks is known right now, it may still be solved in the future. Some complex tasks (e.g. factoring) can be solved in polynomial time using quantum computation.

Instead of assuming the hardness of a particular computational task, the security may be based on a more general assumption such as the existence of trapdoor functions. For a trapdoor function f, given an input x, it is easy to compute $f(x)$, but it is difficult to

compute x from $f(x)$ unless a special *trapdoor* is known, which may depend on f itself, but not on x. A weaker assumption is the existence of one-way functions that do not require the existence of a trapdoor. When implementing a protocol, a specific one-way function f can be chosen. If it turns out that this particular f is not one-way, the protocol will not be immediately broken, as some other f can be used instead. In this case, the particular implementation becomes insecure, but not the whole protocol.

It is not known if one-way functions exist. There are no computational problems whose hardness can be steadily proven, so in the best-case scenario no computational assumptions are used. The next level is *statistical* security, where the data may leak only with negligible probability. If the leakage probability is 0, then we have *perfect* security.

Maliciousness. In previous sections, we considered protocols secure against passive and active adversaries. We describe two intermediate levels between passive and active adversaries.

A *fail-stop* adversary [45] follows the protocol similarly to the passive adversary, except for the possibility of aborting. This means that the adversary has the power to interrupt the protocol execution, but nothing more compared to the passive one.

A *covert* adversary [46] estimates the probability of being caught. It deviates from the protocol as long as this probability is sufficiently low.

Adversary mobility. A *static* adversary chooses a set of corrupted parties before the protocol starts. After that, the set of corrupted parties stays immutable.

An *adaptive* adversary adds parties to the malicious set during the execution of the protocol, until the threshold is reached. The choice of the next corrupted party depends on the state of the other parties corrupted so far.

A *mobile* adversary can not only add new parties to the malicious set during the execution of the protocol, but also remove them, so that some other party can be corrupted instead.

Corrupted parties. In the simplest case, a single adversary that corrupts a set of parties.

In the case of *mixed adversary security*, different sets of parties can be corrupted by different adversaries. For example, it may happen that one of them is passive, and the other active.

In the case of *hybrid security*, the protocol may tolerate different sets of corrupted parties with different capabilities. For example, one set of malicious parties is computationally bounded, while the other is not.

Fairness. If a protocol has the *agreement* property, then if at least one *honest* party receives its output, then all the other honest parties do as well. If a protocol has the *fairness* property, then if *any* party receives its output, then all the honest parties do as well.

Composability. If a protocol is secure in the *stand-alone* model, then it is secure only if it is executed once, and there are no other protocols running. For example, if one protocol uses PKI for commitments, the secret key is published when the commitment is opened, and the keys cannot be reused. Hence, the protocol can be run only once.

If the protocol is *sequentially* composable, then it is secure regardless of any other instance of the same protocol running before and after it. However, there may still be problems if some other protocol is running in parallel. For example, a party P_1 may instantiate two protocol executions with P_2 and P_3, pretending to be P_3 for P_2. If P_2

requires proof that it indeed communicates with P_3, and sends a corresponding challenge to which only P_3 can respond, then P_1 may deliver this challenge to P_3 in a parallel protocol session in which it is the turn of P_1 to send the challenge.

A protocol that supports *parallel* composition is secure even if several instances are executed in parallel, regardless of the timings that the adversary inserts between the rounds of all the protocol runs. However, it can still be insecure in the presence of some other protocols. For example, a protocol that uses PKI for message transmission can be secure in parallel composition, but executing another protocol that uses the same PKI for commitments will break it.

A *universally* composable [35] protocol is secure, regardless of the environment. More details can be found in Sec. 8.

The presented protocol properties can be seen as dimensional axes that define some space in which the protocols can be compared to each other. These axes are not orthogonal: if we want to improve one property then the worsening of some other property may be unavoidable. Below are some possibility and impossibility results of combining these properties that show how different axes depend on each other. More results can be found in [43].

10.1. Impossibility Results

If there are no cryptographic assumptions (assuming the existence of private channels), the following is required:

- If statistical or perfect security is obtained, then either broadcast or private channels must be assumed [47].
- For unconditional security (negligible failure probability) against t maliciously corrupted parties, $n/3 \leq t < n/2$, a broadcast channel is required [47].
- There can be no unconditionally secure protocol against an adversary controlling the majority of the parties. Two-party computation is impossible without cryptographic assumptions [47].
- No protocol can have perfect security against more than $n/3$ maliciously corrupted adversaries [47].
- Let t_a, t_p, and t_f be the number of actively corrupt, passively corrupt, and fail-corrupt parties, respectively. Perfect security is possible if and only if $3t_a + 2t_p + t_f < n$, regardless of the existence of a broadcast channel (a party that is passively corrupt and fail-corrupt at the same time is counted twice) [48].
- Unconditional security without a broadcast channel is possible if and only if $2t_a + 2t_p + t_f < n$ and $3t_a + t_f < n$ [48].
- Unconditional security, with a broadcast channel, is possible if and only if $2t_a + 2t_p + t_f < n$ [48].

For cryptographic security against $n/3$ or more maliciously corrupt players, either a trusted key setup or a broadcast channel is required [13].

Fail-stop adversaries can be tolerated only if there are fewer than $n/2$ corrupt parties, no matter what other assumptions we use [49].

No protocol with security against malicious adversaries can tolerate more than $n/2$ corrupted parties without losing the complete fairness property [50].

There is no protocol with UC security against more than $n/2$ corrupt parties without setup assumptions [51].

10.2. Possibility Results

Passive adversaries can be handled if fewer than $n/2$ participants are corrupt [47]. Active adversaries can be handled, if fewer than $n/2$ participants are corrupt and we are willing to tolerate an exponentially small chance of failure or leakage [50].

In the case of a mixed adversary, there are protocols that tolerate fewer than $n/3$ actively corrupt parties and further passively corrupt parties, so that the total number of corrupt parties is fewer than $n/2$ [48].

Using cryptographic assumptions, we can tolerate any number of active adversaries (although no protection against failures is guaranteed). As an example, we may take any protocol that uses threshold homomorphic encryption (see Sec. 6), taking an n-out-of-n threshold encryption.

References

[1] Oded Goldreich. *Foundations of Cryptography: Volume 2, Basic Applications*. Cambridge University Press, New York, NY, USA, 2004.

[2] Ronald Cramer, Ivan Damgård, and Jesper Buus Nielsen. Secure multiparty computation and secret sharing: An information theoretic approach (book draft), May 2013. `http://www.cs.au.dk/~jbn/mpc-book.pdf`.

[3] Manoj Prabhakaran and Amit Sahai, editors. *Secure Multiparty Computation*. Number 10 in Cryptology and Information Security Series. IOS Press, January 2013.

[4] Martin Hirt and Ueli M. Maurer. Player simulation and general adversary structures in perfect multiparty computation. *J. Cryptology*, 13(1):31–60, 2000.

[5] Oded Goldreich. *Foundations of Cryptography: Basic Tools*, volume 1. Cambridge University Press, New York, NY, USA, 2000.

[6] Mihir Bellare and Silvio Micali. Non-interactive oblivious transfer and spplications. In Gilles Brassard, editor, *Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings*, volume 435 of *Lecture Notes in Computer Science*, pages 547–557. Springer, 1989.

[7] Taher El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, 1985.

[8] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In Dorothy E. Denning, Raymond Pyle, Ravi Ganesan, Ravi S. Sandhu, and Victoria Ashby, editors, *CCS '93, Proceedings of the 1st ACM Conference on Computer and Communications Security, Fairfax, Virginia, USA, November 3-5, 1993.*, pages 62–73. ACM, 1993.

[9] Joe Kilian. Founding Crytpography on Oblivious Transfer. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, STOC '88, pages 20–31, New York, NY, USA, 1988. ACM.

[10] Russell Impagliazzo and Steven Rudich. Limits on the provable consequences of one-way permutations. In Johnson [52], pages 44–61.

[11] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In Boneh [53], pages 145–161.

[12] David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty Unconditionally Secure Protocols (Extended Abstract). In Simon [54], pages 11–19.

[13] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority. In *STOC*, pages 218–229. ACM, 1987.

[14] Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *FOCS*, pages 160–164. IEEE, 1982.

[15] Yehuda Lindell and Benny Pinkas. A Proof of Security of Yao's Protocol for Two-Party Computation. *J. Cryptology*, 22(2):161–188, 2009.

[16] Yehuda Lindell, Benny Pinkas, and Nigel P. Smart. Implementing two-party computation efficiently with security against malicious adversaries. In Rafail Ostrovsky, Roberto De Prisco, and Ivan Visconti, editors, *Security and Cryptography for Networks, 6th International Conference, SCN 2008, Amalfi, Italy,*

September 10-12, 2008. Proceedings, volume 5229 of *Lecture Notes in Computer Science*, pages 2–20. Springer, 2008.

[17] Mihir Bellare, Viet Tung Hoang, Sriram Keelveedhi, and Phillip Rogaway. Efficient garbling from a fixed-key blockcipher. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 478–492. IEEE Computer Society, 2013.

[18] Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free XOR gates and applications. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part II - Track B: Logic, Semantics, and Theory of Programming & Track C: Security and Cryptography Foundations*, volume 5126 of *Lecture Notes in Computer Science*, pages 486–498. Springer, 2008.

[19] Benny Pinkas, Thomas Schneider, Nigel P. Smart, and Stephen C. Williams. Secure two-party computation is practical. In Mitsuru Matsui, editor, *Advances in Cryptology - ASIACRYPT 2009, 15th International Conference on the Theory and Application of Cryptology and Information Security, Tokyo, Japan, December 6-10, 2009. Proceedings*, volume 5912 of *Lecture Notes in Computer Science*, pages 250–267. Springer, 2009.

[20] Samee Zahur, Mike Rosulek, and David Evans. Two halves make a whole: Reducing data transfer in garbled circuits using half gates. Cryptology ePrint Archive, Report 2014/756, 2014. `http://eprint.iacr.org/`.

[21] Yehuda Lindell and Benny Pinkas. Secure two-party computation via cut-and-choose oblivious transfer. In Yuval Ishai, editor, *Theory of Cryptography - 8th Theory of Cryptography Conference, TCC 2011, Providence, RI, USA, March 28-30, 2011. Proceedings*, volume 6597 of *Lecture Notes in Computer Science*, pages 329–346. Springer, 2011.

[22] Charanjit S. Jutla. Encryption modes with almost free message integrity. *J. Cryptology*, 21(4):547–578, 2008.

[23] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.

[24] Rosario Gennaro, Michael O. Rabin, and Tal Rabin. Simplified vss and fact-track multiparty computations with applications to threshold cryptography. In *PODC*, pages 101–111, 1998.

[25] Torben P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In Feigenbaum [55], pages 129–140.

[26] Ranjit Kumaresan, Arpita Patra, and C. Pandu Rangan. The round complexity of verifiable secret sharing: The statistical case. In Masayuki Abe, editor, *ASIACRYPT*, volume 6477 of *Lecture Notes in Computer Science*, pages 431–447. Springer, 2010.

[27] Michael Backes, Aniket Kate, and Arpita Patra. Computational verifiable secret sharing revisited. In Dong Hoon Lee and Xiaoyun Wang, editors, *ASIACRYPT*, volume 7073 of *Lecture Notes in Computer Science*, pages 590–609. Springer, 2011.

[28] Ivan Damgård and Mads Jurik. A Generalisation, a Simplification and Some Applications of Paillier's Probabilistic Public-Key System. In Kwangjo Kim, editor, *Public Key Cryptography*, volume 1992 of *Lecture Notes in Computer Science*, pages 119–136. Springer, 2001.

[29] Ronald Cramer, Ivan Damgård, and Jesper Buus Nielsen. Multiparty computation from threshold homomorphic encryption. In Birgit Pfitzmann, editor, *EUROCRYPT*, volume 2045 of *Lecture Notes in Computer Science*, pages 280–299. Springer, 2001.

[30] Ivan Damgård and Jesper Buus Nielsen. Universally composable efficient multiparty computation from threshold homomorphic encryption. In Boneh [53], pages 247–264.

[31] Craig Gentry. Fully homomorphic encryption using ideal lattices. In Michael Mitzenmacher, editor, *STOC*, pages 169–178. ACM, 2009.

[32] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO*, volume 7417 of *Lecture Notes in Computer Science*, pages 643–662. Springer, 2012.

[33] Craig Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009. `crypto.stanford.edu/craig`.

[34] Donald Beaver. Efficient multiparty protocols using circuit randomization. In Feigenbaum [55], pages 420–432.

[35] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, pages 136–145. IEEE Computer Society, 2001.

[36] Yehuda Lindell. *Composition of Secure Multi-Party Protocols, A Comprehensive Study*, volume 2815 of

Lecture Notes in Computer Science. Springer, 2003.

[37] Dan Bogdanov, Liina Kamm, Sven Laur, and Ville Sokk. Rmind: a tool for cryptographically secure statistical analysis. Cryptology ePrint Archive, Report 2014/512, 2014.

[38] Liina Kamm, Dan Bogdanov, Sven Laur, and Jaak Vilo. A new way to protect privacy in large-scale genome-wide association studies. *Bioinformatics*, 29(7):886–893, 2013.

[39] Martin Burkhart, Mario Strasser, Dilip Many, and Xenofontas Dimitropoulos. SEPIA: Privacy-preserving aggregation of multi-domain network events and statistics. In *USENIX Security Symposium*, pages 223–239, Washington, DC, USA, 2010.

[40] Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas P. Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, Michael I. Schwartzbach, and Tomas Toft. Secure multiparty computation goes live. In *Financial Cryptography*, pages 325–343, 2009.

[41] Joan Feigenbaum, Benny Pinkas, Raphael Ryger, and Felipe Saint-Jean. Secure computation of surveys. In *EU Workshop on Secure Multiparty Protocols*, 2004.

[42] Dan Bogdanov, Riivo Talviste, and Jan Willemson. Deploying secure multi-party computation for financial data analysis - (short paper). In Angelos D. Keromytis, editor, *Financial Cryptography*, volume 7397 of *Lecture Notes in Computer Science*, pages 57–64. Springer, 2012.

[43] Jason Perry, Debayan Gupta, Joan Feigenbaum, and Rebecca N. Wright. Systematizing secure computation for research and decision support. In Michel Abdalla and Roberto De Prisco, editors, *Security and Cryptography for Networks - 9th International Conference, SCN 2014, Amalfi, Italy, September 3-5, 2014. Proceedings*, volume 8642 of *Lecture Notes in Computer Science*, pages 380–397. Springer, 2014.

[44] Manuel Blum, Paul Feldman, and Silvio Micali. Non-interactive zero-knowledge and its applications. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, STOC '88, pages 103–112, New York, NY, USA, 1988. ACM.

[45] Zvi Galil, Stuart Haber, and Moti Yung. Cryptographic computation: Secure fault-tolerant protocols and the public-key model (extended abstract). In Carl Pomerance, editor, *Advances in Cryptology - CRYPTO '87*, volume 293 of *Lecture Notes in Computer Science*, pages 135–155. Springer Berlin Heidelberg, 1988.

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

[47] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In Simon [54], pages 1–10.

[48] Matthias Fitzi, Martin Hirt, and Ueli Maurer. Trading correctness for privacy in unconditional multi-party computation. In Hugo Krawczyk, editor, *Advances in Cryptology — CRYPTO '98*, volume 1462 of *Lecture Notes in Computer Science*, pages 121–136. Springer-Verlag, August 1998. Corrected proceedings version.

[49] R Cleve. Limits on the security of coin flips when half the processors are faulty. In *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, STOC '86, pages 364–369, New York, NY, USA, 1986. ACM.

[50] Tal Rabin and Michael Ben-Or. Verifiable secret sharing and multiparty protocols with honest majority (extended abstract). In Johnson [52], pages 73–85.

[51] Ran Canetti, Eyal Kushilevitz, and Yehuda Lindell. On the limitations of universally composable two-party computation without set-up assumptions. In Eli Biham, editor, *Advances in Cryptology – EUROCRYPT 2003*, volume 2656 of *Lecture Notes in Computer Science*, pages 68–86. Springer Berlin Heidelberg, 2003.

[52] David S. Johnson, editor. *Proceedings of the 21st Annual ACM Symposium on Theory of Computing, May 14-17, 1989, Seattle, Washigton, USA*. ACM, 1989.

[53] Dan Boneh, editor. *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, volume 2729 of *Lecture Notes in Computer Science*. Springer, 2003.

[54] Janos Simon, editor. *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA*. ACM, 1988.

[55] Joan Feigenbaum, editor. *Advances in Cryptology - CRYPTO '91, 11th Annual International Cryptology Conference, Santa Barbara, California, USA, August 11-15, 1991, Proceedings*, volume 576 of *Lecture Notes in Computer Science*. Springer, 1992.