

Chapter 2

Basics of Efficient Secure Function Evaluation

2.1 Common Notation and Definitions

In this section we introduce common notation (Sect. 2.1.1), cryptographic primitives (Sect. 2.1.2), function representations (Sect. 2.1.3), the adversary model (Sect. 2.1.4), and the Random Oracle (RO) model (Sect. 2.1.5) used in this book.

2.1.1 Notation

We use the following standard notations.

2.1.1.1 Basics

Bitstrings. $\{0, 1\}^\ell$ denotes the space of binary strings of length ℓ . $a||b$ denotes the concatenation of strings a and b . $\langle a, b \rangle$ is a vector with two components a and b , and its representation as a bit string is $a||b$. For strings $s, t \in \{0, 1\}^\ell$, $s \oplus t$ denotes their *bitwise exclusive-or* (XOR).

Random Choice. Uniform random choice is denoted by the \in_R operator, e.g., $r \in_R D$ reads “draw r uniformly at random from D ”.

Protocol Participants. We call the two Secure Function Evaluation (SFE) participants *client* \mathcal{C} (Alice) and *server* \mathcal{S} (Bob). This naming choice is influenced by the asymmetry in the SFE protocols, which fits into the client–server model. We want to point out that we do not limit ourselves to this setting even though this client–server relationship in fact exists in most real-life two-party SFE scenarios.

2.1.1.2 Security and Correctness Parameters

Our security and correctness parameters are named as shown in Table 2.1.

Table 2.2 contains current recommendations by ECRYPT II [74] for the size of the symmetric security parameter t and the asymmetric security parameter T .

Table 2.1 Security and correctness parameters

Symbol	Name
t	Symmetric security parameter (bit length of symmetric keys)
T	Asymmetric security parameter (bit length of RSA moduli)
σ	Statistical security parameter
κ	Correctness parameter

Table 2.2 Security parameters: recommended sizes [74]

Security level	Recommended use until	t (bit)	T (bit)
Ultra-short	2012	80	1,248
Short	2020	96	1,776
Medium	2030	112	2,432
Long	2040	128	3,248

An overview and comparison of different recommendations is available at [96].

In implementations, the statistical security parameter σ and the correctness parameter κ can be chosen as $\sigma = \kappa = 40$.

2.1.2 Cryptographic Primitives

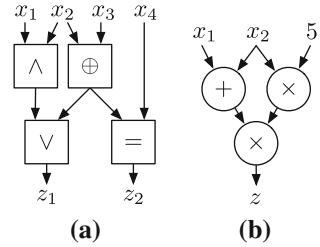
Pseudo-Random Function (PRF) keyed with k and evaluated on x is denoted by $\text{PRF}_k(x)$. PRF can be instantiated with a block cipher, e.g., AES, or a cryptographic hash function, e.g., SHA-256. AES is preferable if PRF is run repeatedly with the same key k as in this case the key schedule of AES needs to be run only once and hence amortizes.

Message Authentication Code (MAC) keyed with k and evaluated on message m is denoted by $\text{MAC}_k(m)$. In our token-based protocols in Chap. 4 we use a MAC algorithm that does not need to store the entire message, but can operate “online” on small blocks, e.g., AES-CMAC [204] or HMAC [146].

2.1.3 Function Representations

We use several standard representations for functions which are particularly useful for SFE protocols as shown in Fig. 2.1: boolean circuits (Sect. 2.1.3.1) and arithmetic circuits (Sect. 2.1.3.2).

Fig. 2.1 Function representations. **a** Boolean circuit. **b** Arithmetic circuit



2.1.3.1 Boolean Circuits

Boolean circuits are a classical representation of functions in engineering and computer science.

Definition 1 (*Boolean Circuit*) A *boolean circuit* with u inputs, v outputs and n gates is a Directed Acyclic Graph (DAG) with $|V| = u + v + n$ vertices (nodes) and $|E|$ edges. Each node corresponds to either a *gate*, an *input* or an *output*. The edges are called *wires*.

For simplicity, the input and output nodes are often omitted in the graphical representation of a boolean circuit as shown in Fig. 2.1a. For a more detailed definition see [225].

Definition 2 (*Gate*) A d -input gate G_d is a boolean function which maps $d \geq 0$ input bits to one output bit, i.e., $G_d : (\text{in}_1, \dots, \text{in}_d) \in \{0, 1\}^d \rightarrow \{0, 1\}$.

Typical gates are XOR (\oplus), XNOR ($=$), AND (\wedge), OR (\vee).

Topologic Order. Gates of a boolean circuit can be evaluated in any order, as long as all of the current gate's inputs are known. This property is ensured by sorting (and evaluating) the gates topologically, which can be done efficiently in $O(|V| + |E|)$ [64, Topological sort, pp. 549–552]. The topologic order of a boolean circuit indexes the gates with labels G_1, \dots, G_n and ensures that the i th gate G_i has no inputs that are outputs of a successive gate $G_{j>i}$. In complexity theory, a circuit with such a topologic order is called a *straight-line program* [6]. Given the values of the inputs, the output of the boolean circuit can be evaluated by evaluating the gates one-by-one in topologic order. A valid topologic order for the example boolean circuit in Fig. 2.1a would be $\wedge, \oplus, \vee, =$. The topologic order is not necessarily unique, e.g., $\oplus, \wedge, =, \vee$ would be possible as well.

Throughout this book we assume that boolean circuits are ordered topologically.

2.1.3.2 Arithmetic Circuits

Arithmetic circuits are a more compact function representation than boolean circuits.

An *arithmetic circuit* over a ring R and the set of variables x_1, \dots, x_n is a DAG. Figure 2.1a illustrates an example. Each node with in-degree zero is called an input

gate labeled by either a variable x_i or an element in R . Every other node is called a gate and labeled by either $+$ or \times denoting addition or multiplication in R .

Any boolean circuit can be expressed as an arithmetic circuit over $R = \mathbb{Z}_2$. However, if we use $R = \mathbb{Z}_m$ for sufficiently large modulus m , the arithmetic circuit can be much smaller than its corresponding boolean circuit, as integer addition and multiplication can be expressed as single operations in \mathbb{Z}_m .

Number Representation. We note that arithmetic circuits can simulate computations on both positive and negative integers x by mapping them into elements of $\mathbb{Z}_m : x \mapsto x \bmod m$. As with all fixed precision arithmetics, over- and underflows must be avoided.

2.1.4 Adversary Model

The standard approach for formalizing and proving security of cryptographic protocols is to consider adversaries with different capabilities. In the following we give intuition for the capabilities of *malicious*, *covert*, and *semi-honest* adversaries. We refer to [99] for formal definitions and to [145, 152] for more detailed discussions.

Malicious adversaries, also called *active adversaries*, are the strongest type of adversaries and are allowed to arbitrarily deviate from the protocol, aiming to learn private inputs of the other parties and/or to influence the outcome of the computation. Not surprisingly, protection against such attacks is relatively expensive, as discussed in Sect. 2.3.1.2.

Covert adversaries are similar to malicious adversaries, but with the restriction that they must avoid being caught cheating. That is, a protocol in which an active attacker may gain advantage may still be considered secure if attacks are discovered with certain fixed probability (e.g., $1/2$). It is reasonable to assume that in many social, political and business scenarios the consequences of being caught overweight the gain from cheating. At the same time, protocols secure against covert adversaries can be substantially more efficient than those secure against malicious players, as shown in Sect. 2.3.1.2.

Semi-honest adversaries, also called *passive adversaries*, do not deviate from the protocol but try to infer additional information from the transcript of messages seen in the protocol. Far from trivial, this model covers many typical practical settings such as protection against insider attacks. Further, designing and evaluating the performance of protocols in the semi-honest model is a first step towards protocols with stronger security guarantees (cf. Sect. 2.3.1.2). Indeed, most protocols and implementations of protocols for practical privacy-preserving applications focus on the semi-honest model [19, 76, 164, 173, 189].

2.1.5 Random Oracle Model

Some of our constructions in this book make use of ROs [24], a relatively strong assumption. In fact, it has been shown in [57] that some (contrived) uses of RO cannot be securely instantiated with *any* hash function. Therefore, proofs in the RO model cannot be seen as proofs in the strictest mathematical sense. However, we believe that modeling cryptographic hash functions, such as SHA-256, as RO is well-justified in many practical settings because of the following reasons:

Firstly, to date, no attacks exploiting the RO assumption are known on practical systems. This holds true even in the academic context: Important attacks on SHA-1 [226] that exploit the structure of the functions were far from being practical, and simply accelerated migration to stronger primitives, which are believed secure today. While some attacks, such as the attack on MD5 [209], are in fact practical, the use of MD5 had long been considered unsafe, and [209] broke poorly managed systems. Thus we do not consider [209] an attack on properly implemented protocols. In fact, [209] and the preliminary work that led to it only support the historic fact that users of hash functions do receive weakness warnings years ahead of possible real breaks.

Further, even in well-understood and deployed real-life systems, the crypto core (which includes the employed hash functions) is almost never targeted for attacks, in favor of *much* easier to exploit implementation flaws.

In sum, we believe that making the RO assumption on the employed hash function is practically justified in our and many other settings.

2.2 Cryptographic Primitives for Secure Two-Party Computation

In this section we describe essential building blocks used in SFE protocols: Homomorphic Encryption (HE) in Sect. 2.2.1, Garbled Circuits (GCs) in Sect. 2.2.2, and Oblivious Transfer (OT) in Sect. 2.2.3.

2.2.1 Homomorphic Encryption

HE schemes are semantically secure encryption schemes which allow computation of specific operations on ciphertexts and hence can be used for secure evaluation of arithmetic circuits as described next.

Let $(\text{Gen}, \text{Enc}, \text{Dec})$ be a semantically secure encryption scheme with plaintext space P , ciphertext space C , and algorithms for key generation Gen , encryption Enc and decryption Dec . We write $\llbracket m \rrbracket$ for $\text{Enc}(m, r)$, where r is random.

Table 2.3 Additively homomorphic encryption schemes

Scheme	P	Ciphertext size	$\text{Enc}(m, r)$
Paillier [176]	\mathbb{Z}_N	$2T$	$g^m r^N \bmod N^2$
Damgård–Jurik [65]	\mathbb{Z}_{N^s}	$(s + 1)T$	$g^m r^{N^s} \bmod N^{s+1}$
Damgård–Geisler–Krøigaard [66]	\mathbb{Z}_u	T	$g^m h^{r'} \bmod N$
Lifted EC-ElGamal [75]	\mathbb{Z}_p	$4t + 2$	$(g^r, g^m h^r)$

N RSA modulus; $s \geq 1$; u small prime; p $2t$ -bit prime; g, h generators

2.2.1.1 Additively Homomorphic Encryption

An *additively* HE scheme allows addition under encryption as follows. It defines an operation $+$ on plaintexts and a corresponding operation \boxplus on ciphertexts, satisfying $\forall x, y \in P : \llbracket x \rrbracket \boxplus \llbracket y \rrbracket = \llbracket x + y \rrbracket$. This naturally allows for multiplication with a plaintext constant a using repeated doubling and adding: $\forall a \in \mathbb{N}, x \in P : a \llbracket x \rrbracket = \llbracket ax \rrbracket$.

Popular instantiations for additively HE schemes are summarized in Table 2.3: The Paillier cryptosystem [176] provides a T -bit plaintext space and $2T$ -bit ciphertexts, where T is the size of the RSA modulus N , and is sufficient for most applications (details see below). The Damgård–Jurik cryptosystem [65] is a generalization of the Paillier cryptosystem which provides a larger plaintext space of size sT -bit for $(s + 1)T$ -bit ciphertexts for arbitrary $s \geq 1$. The Damgård–Geisler–Krøigaard cryptosystem [66–68] has smaller ciphertexts of size T -bit, but can only be used with a small plaintext space \mathbb{Z}_u , where u is a small prime, as decryption requires computation of a discrete logarithm. Finally, the lifted ElGamal [75] cryptosystem has additively homomorphic properties and very small ciphertexts. However, decryption is only possible if the plaintext is known to be in a small subset of the plaintext space, as the discrete logarithm of a generator with large order has to be brute-forced. Lifted ElGamal implemented over an EC group (Lifted EC-ElGamal) provides a $2t$ -bit plaintext space and very small ciphertexts of size $2(2t + 1)$ bits $\sim 4t$ bits when using point compression.

The Paillier Cryptosystem. The most widely used additively HE system is that of Paillier [176] where the public key is an RSA modulus N and the secret key is the factorization of N . The extension of [65, Sect. 6] allows one to pre-compute expensive modular exponentiations of the form $r^N \bmod N^2$ in a setup phase, s.t. only two modular multiplications per encryption are needed in the time-critical online phase. The party which knows the factorization of N (i.e., the secret key), can use Chinese remaindering to efficiently pre-compute these exponentiations and to decrypt.

2.2.1.2 Fully Homomorphic Encryption

Some HE systems allow both addition and multiplication under encryption. For this, a separate operation \times for multiplication of plaintexts and a corresponding operation \boxtimes on ciphertexts is defined satisfying $\forall x, y \in P : \llbracket x \rrbracket \boxtimes \llbracket y \rrbracket = \llbracket x \times y \rrbracket$. Cryptosystems with such a property are called *fully* homomorphic.

Until recently, it was widely believed that such cryptosystems do not exist. Several works provided partial solutions: [34] and [95] allow for polynomially many additions and one multiplication, and ciphertexts of [193] grow exponentially in the number of multiplications. Recent schemes [92, 93, 201, 223] are fully homomorphic. Even if the size of the ciphertexts of these fully HE schemes is independent of the number of multiplications, the size and computational cost of fully HE schemes are *substantially* larger than those of additively HE schemes. First implementation results of [201] show that even for almost fully HE schemes with conservatively chosen security parameters that allow for multiplicative depth $d = 2.5$ of the evaluated circuit, i.e., at most two multiplications, encrypting a single bit takes 3.7 s on a 2.4 GHz Intel Core2 (6600) CPU. Most recent implementation results of [148] indicate that the performance of somewhat homomorphic encryption schemes might be sufficient for outsourcing certain types of computations, whereas fully HE is still very inefficient as shown in [94] whose implementation requires in the order of Gigabytes of communication and minutes of computation on high-end IBM System \times 3500 servers.

Although significant effort is underway in the theoretical community to improve its performance, it seems unlikely that fully HE will reach the efficiency of current public-key encryption schemes. Intuitively, this is because a fully HE cryptosystem must provide both the same strong security guarantees (semantic security) and extra algebraic structure to allow for homomorphic operations. The extra structure weakens security, and countermeasures (costing performance) are necessary.

2.2.1.3 Computing on Encrypted Data

Homomorphic encryption naturally allows for secure evaluation of arithmetic circuits over P , called computing on encrypted data, as follows. The client \mathcal{C} generates a key pair for an HE cryptosystem and sends her inputs encrypted under the public key to the server \mathcal{S} together with the public key. With a fully HE scheme, \mathcal{S} can simply evaluate the arithmetic circuit by computing on the encrypted data and send back the (encrypted) result to \mathcal{C} , who then decrypts it to obtain the output.¹ If the HE scheme only supports addition, one round of interaction between \mathcal{C} and \mathcal{S} is needed to evaluate each multiplication gate (or layer of multiplication gates) as described below. Today,

¹ If \mathcal{S} is malicious, it must additionally be ensured that he indeed computed the intended functionality by means of verifiable computing (cf. Sect. 4.3.3.2).

using additively HE and interaction for multiplication results in much faster SFE protocols than using fully HE schemes.

Packing. Often the plaintext space P of the HE scheme is substantially larger than the size of the encrypted numbers. This allows one to pack multiple numbers into one ciphertext (before or after additive blinding) and send back only a single ciphertext from \mathcal{S} to \mathcal{C} . This optimization substantially decreases the size of the messages sent from \mathcal{S} to \mathcal{C} as well as the number of decryptions performed by \mathcal{C} . The computational overhead for \mathcal{S} is small as packing the ciphertexts $\llbracket x_1 \rrbracket, \dots, \llbracket x_n \rrbracket$ into one ciphertext $\llbracket X \rrbracket = \llbracket x_n \rrbracket \parallel \dots \parallel \llbracket x_1 \rrbracket$ costs less than one full-range modular exponentiation by using Horner's scheme: $\llbracket X \rrbracket = \llbracket x_n \rrbracket$; for $i = n - 1$ downto 1 : $\llbracket X \rrbracket = 2^{\lfloor x_{i+1} \rfloor} \llbracket X \rrbracket \boxplus \llbracket x_i \rrbracket$.

Interactive Multiplication with Additively Homomorphic Encryption. To multiply two ℓ -bit values encrypted under additively HE and held by \mathcal{S} , $\llbracket x \rrbracket$ and $\llbracket y \rrbracket$, the following standard protocol requires one single round of interaction between \mathcal{S} and \mathcal{C} : \mathcal{S} randomly chooses $r_x, r_y \in_R \{0, 1\}^{\ell+\sigma}$, where σ is the statistical security parameter, computes the blinded values $\llbracket \bar{x} \rrbracket = \llbracket x + r_x \rrbracket$, $\llbracket \bar{y} \rrbracket = \llbracket y + r_y \rrbracket$ and sends these to \mathcal{C} . \mathcal{C} decrypts, multiplies and sends back $\llbracket z \rrbracket = \llbracket \bar{x} \bar{y} \rrbracket$. \mathcal{S} obtains $\llbracket xy \rrbracket$ by computing $\llbracket xy \rrbracket = \llbracket z \rrbracket \boxplus (-r_x) \llbracket y \rrbracket \boxplus (-r_y) \llbracket x \rrbracket \boxplus \llbracket -r_x r_y \rrbracket$.

Efficiency of single or multiple multiplications in parallel can be improved by *packing* the blinded ciphertexts together instead of sending them to \mathcal{C} separately.

Security. We note that in SFE protocols based on HE, the security of the party who knows the secret key (\mathcal{C} in our setting) is *computational* as it is computationally hard for the other party to break the semantic security of the HE scheme. The security of the party who computes under HE (\mathcal{S} in our setting) is *statistical* as this party always statistically blinds all intermediate values before sending them back.

Efficiency. As SFE based on additively HE requires interaction for multiplying two ciphertexts, the *round complexity* of such protocols is determined by the multiplicative depth of the evaluated function, i.e., the number of successive multiplications.

When the public key is known to both parties, encryptions and re-randomization values can be pre-computed in a *setup phase*.

Still, the *online phase* requires computationally expensive public key operations such as modular exponentiations for multiplying with constants, or decryptions.

2.2.2 Garbled Circuit Constructions

Another efficient method for computing under encryption is based on Garbled Circuits (GCs). The fundamental idea of GCs going back to Yao [231], is to represent the function f to be evaluated as a boolean circuit C and encrypt (*garble*) each wire with a symmetric encryption scheme. In contrast to HE (cf. Sect. 2.2.1), the encryptions here cannot be operated on directly, but require helper information which is generated and exchanged in a setup phase in the form of a *garbled table* for each gate.

In this section we summarize existing schemes for constructing and evaluating GCs and give applications in Sect. 2.3.1. We give an algorithmic description of GCs and refer to the original papers on GCs constructions for details and proofs of security.

2.2.2.1 Components of GC Constructions

We start by briefly introducing the main components of GC constructions: garbled values and garbled tables to compute thereon.

Garbled Values. Computations in a GC are not performed on plain values 0 or 1, but on random-looking secrets, called *garbled values*. During construction of the GC, two random-looking garbled values $\tilde{w}_i^0, \tilde{w}_i^1$ are assigned to each wire w_i of C . The garbled value \tilde{w}_i^j corresponds to the plain value j , but, as it looks random, does not reveal its corresponding plain value j .

In efficient GC constructions, each garbled value is composed of a symmetric t -bit key and a random-looking *permutation bit* (see Point-and-Permute below):

$$\tilde{w}_i^0 = \langle k_i^0, \pi_i^0 \rangle, \tilde{w}_i^1 = \langle k_i^1, \pi_i^1 \rangle \quad \text{with } k_i^0, k_i^1 \in \{0, 1\}^t, \pi_i^0 \in \{0, 1\} \quad (2.1)$$

and

$$\pi_i^1 = 1 - \pi_i^0. \quad (2.2)$$

The exact method for choosing the values k_i^0, k_i^1, π_i^0 is determined by the specific GC construction (cf. Sect. 2.2.2.3).

Garbled Tables. To allow computations on garbled values, for each gate G_i ($i = 1, \dots, n$) of the circuit C , a garbled table \tilde{G}_i is constructed. Given the garbled values corresponding to G_i 's input wires, \tilde{G}_i allows one to decrypt *only* the corresponding garbled value of G_i 's output wire. Formally, let $\text{in}_1, \dots, \text{in}_d$ be the input wires of gate G and out be its output wire. Then, for *any* input combination $b_j \in \{0, 1\}$ ($j = 1, \dots, d$), given the corresponding garbled inputs $\tilde{\text{in}}_1^{b_1}, \dots, \tilde{\text{in}}_d^{b_d}$, the garbled table \tilde{G} allows one to decrypt *only* $\widetilde{\text{out}}^{G(b_1, \dots, b_d)}$.

In particular, no information about the other garbled output value, the plain input bits b_j , or the plain output bit $G(b_1, \dots, b_d)$ is revealed.

The general idea for constructing garbled tables is to use for all possible input combinations b_j the garbled input keys $\tilde{\text{in}}_j^{b_j}$ to symmetrically encrypt the corresponding output key $\widetilde{\text{out}}^{G(b_1, \dots, b_d)}$. The entries of the garbled table are the ciphertexts for all possible input combinations. The position of the entries in the garbled table must be such that it does not reveal any information about the corresponding plain input values b_j .

To achieve this, the original GC construction proposed by Yao [231] randomly permutes the entries in the garbled table. In order to find the right entry to decrypt, the symmetric encryption function requires an *efficiently verifiable range* to determine which entry was decrypted successfully, as described in [151]. However, this method

has a large overhead as multiple trial-decryption need to be performed and ciphertext size increases.

In the following we briefly discuss the state-of-the-art for efficiently instantiating the used symmetric encryption function.

Point-and-Permute. The *point-and-permute* technique described in [164] allows one to immediately find the right entry for decryption in the garbled table as follows: The entries of the garbled table are permuted such that the permutation bits of the garbled input wires π_1, \dots, π_d are used to point directly to the entry of the garbled table which needs to be decrypted. As the permutation bits look random, the position of the entries in the garbled table appears random as well and hence reveals no information about the input bits b_1, \dots, b_d .

By applying the point-and-permute technique, the employed symmetric encryption scheme no longer needs to have an efficiently verifiable range.

Encryption Function. The encryption function for encrypting garbled table entries is $E_{k_1, \dots, k_d}^s(m)$ with inputs d keys of length t , a message m , and some additional information s . The additional information s must be unique per invocation of the encryption function, i.e., it is used only once for any choice of keys. Indeed, it is crucial that in the GC constructions s contains a unique and independent gate identifier (cf. [213]).

As proposed in [154, 180], E can be instantiated efficiently with a *Key Derivation Function* $\text{KDF}^\ell(k_1, \dots, k_d, s)$ whose ℓ bits of output are independent of the input keys k_1, \dots, k_d in isolation, and which depends on the value of s :

$$E_{k_1, \dots, k_d}^s(m) = m \oplus \text{KDF}^{|m|}(k_1, \dots, k_d, s). \quad (2.3)$$

KDF can be instantiated with a cryptographic hash function H :

- The most efficient implementation of KDF is a single invocation of H ,

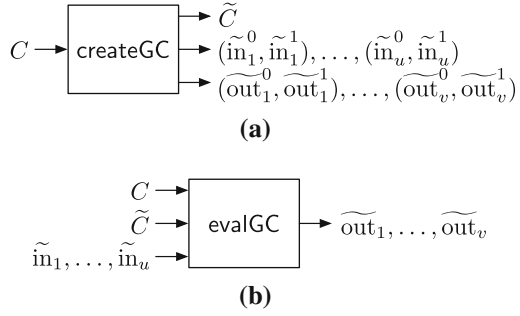
$$\text{KDF}^\ell(k_1, \dots, k_d, s) = H(k_1 || \dots || k_d || s)_{1 \dots \ell}. \quad (2.4)$$

- Alternatively, KDF could also be implemented by d separate calls to H ,

$$\text{KDF}^\ell(k_1, \dots, k_d, s) = H(k_1 || s)_{1 \dots \ell} \oplus \dots \oplus H(k_d || s)_{1 \dots \ell}. \quad (2.5)$$

In practical implementations, H can be chosen for example from the SHA-family. For provable security of the GC construction, H is modeled as RO, circular correlation robust, or PRF, depending on the specific GC construction used as described later in Sect. 2.2.2.5.

Fig. 2.2 Interface of GC constructions. **a** createGC. **b** evalGC



2.2.2.2 Interfaces and Structure of GC Constructions

GC constructions can be seen as algorithms with clean interfaces and a common general structure as described next.

Interface of GC Constructions. Each GC construction consists of two randomized algorithms: **createGC** generates a GC and **evalGC** evaluates it, as shown in Fig. 2.2:

- **createGC** takes a boolean circuit C as input and outputs the corresponding GC \tilde{C} (consisting of a garbled table for each of its gates), and pairs of garbled values for each of C 's input and output wires.
- **evalGC** gets as inputs C , \tilde{C} and one garbled value for each of C 's inputs, $\tilde{\text{in}}_1, \dots, \tilde{\text{in}}_u$ and returns the corresponding garbled output values $\tilde{\text{out}}_1, \dots, \tilde{\text{out}}_v$.

We note that the inputs and outputs of both algorithms can be streams of data, i.e., given piece-by-piece without ever storing the entire objects.

Completeness and Correctness. Each GC construction must be complete and correct. Completeness requires that for all boolean circuits C , **createGC** creates a GC \tilde{C} , pairs of garbled inputs and garbled outputs. Correctness requires that afterwards for all possible input bits $x_i \in \{0, 1\}$, $i = 1, \dots, u$, given the corresponding garbled values $\tilde{\text{in}}_i = \tilde{\text{in}}_i^{x_i}$ as inputs, **evalGC** outputs the garbled values $\tilde{\text{out}}_j = \tilde{\text{out}}_j^{z_j}$, $j = 1, \dots, v$ which correspond to C evaluated on the input values: $(z_1, \dots, z_v) = C(x_1, \dots, x_u)$.

One-time use. We stress that for security reasons, \tilde{C} cannot be evaluated more than once (otherwise, multiple runs of **evalGC** can leak information about input or output values). **evalGC** must always be run on freshly generated outputs of **createGC**.

General Structure of GC Constructions. The efficient GC constructions presented next have the following general structure:

- **createGC** starts by assigning random-looking garbled values $(\tilde{\text{in}}_i^0, \tilde{\text{in}}_i^1)$ to all input wires of C and outputs these. Afterwards, for each gate G_i of C in topologic order (cf. assumption in Sect. 2.1.3.1), two random-looking garbled values are assigned to the gate's output wire and afterwards its garbled table \tilde{G}_i is created and output

as part of \tilde{C} . Finally, the garbled outputs $(\widetilde{\text{out}}_j^0, \widetilde{\text{out}}_j^1)$ for each of C 's output wires are output.

- **evalGC** evaluates the GC \tilde{C} on the garbled inputs $\widetilde{\text{in}}_i$ by evaluating each garbled gate \tilde{G}_i of \tilde{C} in the topologic order determined by C . Finally, the garbled values of C 's output wires $\widetilde{\text{out}}_j$ are output.

2.2.2.3 Efficient GC Constructions

In the following we describe efficient GC constructions which are suited well for efficient implementation [180].

All GC constructions presented next start with choosing the garbled input values. The garbled zero values $\widetilde{\text{in}}_i^0$ are chosen randomly, i.e., $k_i^0 \in_R \{0, 1\}^t$ and $\pi_i^0 \in_R \{0, 1\}$ in Eq. (2.1). The corresponding garbled values for one $\widetilde{\text{in}}_i^1$ are chosen randomly as $k_i^1 \in_R \{0, 1\}^t$, or according to Eq. (2.8) in the case of “free XOR” as described below.

The following GC techniques successively fix the garbled output values of each gate in order to decrease the size of the garbled tables.

Point-and-Permute. The point-and-permute GC construction was first described in [164], implemented in Fairplay [157], and also used in [142, 154].² This technique chooses both garbled output values of a d -input gate G_i at random and results in a garbled table with 2^d table entries. For each of the 2^d possible input combinations b_1, \dots, b_d , the garbled table entry at position π_1, \dots, π_d is constructed by using the keys of G_i 's garbled inputs to encrypt the corresponding garbled output:

$$\pi_1, \dots, \pi_d : E_{k_1^{b_1}, \dots, k_d^{b_d}}^{i||\pi_1||\dots||\pi_d} \left(\widetilde{\text{out}}^{G(b_1, \dots, b_d)} \right). \quad (2.6)$$

Garbled Row Reduction. The GC construction of [164], called *Garbled Row Reduction*, extends the point-and-permute GC construction by fixing one of the garbled output values resulting in a garbled table of $2^d - 1$ table entries. The first entry of each garbled table is forced to be zero and hence does not need to be transferred. By substituting into Eq. (2.6), this fixes one of the two garbled output values to be pseudo-randomly derived from the garbled input values. The other garbled output value is chosen at random satisfying Eq. (2.2). For details we refer to the description in [180].

Secret-Sharing. The GC construction of Pinkas et al. [180] uses Shamir's secret-sharing [197] to fix both garbled output values resulting in a garbled table with $2^d - 2$ entries. In the following we summarize the general idea of this construction and refer to [180, Sect. 5] for details.

The construction exploits the fact that both keys of a gate's garbled output values can be chosen independently and pseudo-randomly. The basic idea is to

² We note that the GC construction of Yu et al. [233, Sect. 3.3] is less efficient as garbled tables are larger and require slightly more computation.

pseudo-randomly derive keys $K_r \in \{0, 1\}^t$ and bit masks $M_r \in \{0, 1\}$ for all combinations of garbled inputs as

$$K_r || M_r = \text{KDF}^{t+1} \left(k_1^{b_1} || \dots || k_d^{b_d} || s \right). \quad (2.7)$$

The keys K_r are interpreted as elements in \mathbb{F}_{2^t} and used as supporting points of two polynomials $P(X), Q(X)$ of the same degree: $P(X)$ is defined by those keys which should map to the garbled output value $\widetilde{\text{out}}^1 := P(0)$. Similarly, $Q(X)$ maps to the garbled output value $\widetilde{\text{out}}^0 := Q(0)$. Overall, $2^d - 2$ points are stored as part of the garbled table, where some points are on both and some on only one of the polynomials. The bits M_r are used to encrypt the permutation bits of the garbled outputs as in the point-and-permute GC construction resulting in an additional 2^d encrypted bits in the garbled table.

During evaluation of the garbled gate, the garbled inputs are used to derive K_r, M_r according to Eq. (2.7). Then, M_r is used to decrypt the output permutation bit which defines through which of the supporting points in the garbled table to interpolate the polynomial. Finally, the garbled output key is determined by evaluating the polynomial at $X = 0$.

Generalization to arbitrary d . We note that the Secret-Sharing GC construction can be generalized from $d = 2$ (as described in [180, Sect. 5]) to arbitrary d -input gates as follows: Assume that n_1 of the 2^d entries in the gate's function table equal one and the remaining $n_0 := 2^d - n_1$ entries equal zero. In the following we assume that $n_1 \geq n_0$ (otherwise we invert the role of zero and one). The polynomial P , interpolated through those keys K_r that should map to the garbled output value for one, has degree n_1 . We store $n_1 - 1$ extra points $P(2^d + 1), \dots, P(2^d + n_1 - 1)$ in the garbled table. Afterwards, we interpolate polynomial Q of degree n_1 through the n_0 keys K_r that should map to the garbled output value for zero and the common $n_1 - n_0$ extra points $P(2^d + 1), \dots, P(2^d + n_1 - n_0)$. Now, we create $n_0 - 1$ extra points $Q(2^d + n_1 - n_0 + 1), \dots, Q(2^d + n_1 - 1)$. The order of the extra points on P and Q in the garbled table is such that the output permutation bit can be used to obviously index which extra points to use for interpolation. The garbled table consists of $n_1 - n_0$ common extra points and $n_0 - 1$ extra points on P resp. Q , in total $n_1 - n_0 + 2(n_0 - 1) = n_1 + n_0 - 2 = 2^d - 2$ keys. The overall size of the garbled table hence is $(2^d - 2)t + 2^d$ bits.

Free XOR. As observed in [142], a fixed distance between corresponding garbled values allows “free” evaluation of XOR gates, i.e., garbled XOR gates require no garbled table and allow very efficient creation and evaluation (XOR of the garbled values). The main idea is to choose a fixed relation between the two garbled values for each garbled wire:

$$k_i^1 := k_i^0 \oplus \Delta, \quad (2.8)$$

where $\Delta \in_R \{0, 1\}^t$ is the randomly chosen *global key distance*. During creation of a garbled XOR gate, the garbled output value is set to $\widetilde{\text{out}}^0 = \widetilde{\text{in}}_1^0 \oplus \widetilde{\text{in}}_2^0$. Similarly,

Table 2.4 Efficient GC constructions for d -input gates

GC technique	Size of garbled table (bits)	Free XOR [142]
Point-and-permute [164]	$2^d t + 2^d$	✓
Garbled row reduction [164]	$(2^d - 1)t + (2^d - 1)$	✓
Secret-sharing [180]	$(2^d - 2)t + 2^d$	✗

t : symmetric security parameter

evaluation of a garbled XOR gate is done by computing $\widetilde{\text{out}} = \widetilde{\text{in}}_1^0 \oplus \widetilde{\text{in}}_2^0$. Garbled non-XOR gates can be constructed with any GC construction which fixes at most one of the garbled outputs of a gate, i.e., from the GC techniques described above Point-and-Permute and Garbled Row Reduction allow combination with “free XORs”, but not the Secret-Sharing technique (cf. Table 2.4).

2.2.2.4 Complexity of Efficient GC Constructions

The complexity of the GC constructions presented in Sect. 2.2.2.3 is summarized in Table 2.4. When using free XORs, XOR gates require no communication and only negligible computation (XOR of bitstrings). We compare the complexity for other gates next.

Computation Complexity. Interestingly, all GC constructions have almost the same computation complexity, which is dominated by invocations of a cryptographic hash function H : for each d -input gate, `createGC` requires 2^d invocations of KDF and `evalGC` requires one invocation. As described in Sect. 2.2.2.1, each invocation of KDF needs one or d invocations of H depending on whether H is modeled as RO or not.

The Secret-Sharing GC construction requires slightly more computations as it also requires interpolation of two polynomials of degree at most $2^d - 1$ over \mathbb{F}_{2^t} .

On the other hand, the computation complexity to randomly choose the garbled output values of the gates decreases as follows: Point-and-Permute chooses both garbled values (one with free XOR), Garbled Row Reduction one (none with free XOR), and Secret-Sharing none.

Communication Complexity. As shown in Table 2.4, the size of each garbled table decreases by approximately t bits per gate from Point-and-Permute to Garbled Row Reduction and from there to Secret-Sharing. Especially for gates with low degree d these savings can be quite significant, i.e., up to -25% for Garbled Row Reduction and -50% for Secret-Sharing for the common case of $d = 2$.

However, the Secret-Sharing construction, which cannot be combined with Free XOR, results only in better communication complexity than Garbled Row Reduction if the evaluated circuits do not have many XOR gates. Indeed, we show in Chap. 3 that

most commonly used circuit building blocks can be transformed such that most of the gates are XOR gates and hence Garbled Row Reduction is more efficient than Secret-Sharing w.r.t. both computation and communication.

2.2.2.5 Security of Efficient GC Constructions

The first full proof of security of the original version of Yao's GC protocol [231] was given in [151]. This proof was later adapted to show the security of various efficient GC constructions that differ in how the underlying KDF is composed from calls to H (cf. Eqs. 2.4 vs. 2.5) and how H needs to be modeled.

For practical applications, modeling H as a RO and instantiating it with a call to a cryptographic hash function, e.g., chosen from the SHA family, should provide reasonable security guarantees for all efficient GC constructions presented above. In more detail, the current situation is as follows:

The GC construction that uses Point-and-Permute together with free XORs and instantiates KDF with a single invocation of H (cf. Eq. 2.4) was proven secure when H is modeled as RO [142]. As proven in [60], this assumption can be relaxed to circular correlation robustness, but not to correlation robustness alone.

According to [154], for Point-and-Permute without free XORs, H can be modeled as RO for one invocation of H (cf. Eq. 2.4), and as PRF for several invocations of H (cf. Eq. 2.5).

As sketched in [180], for Garbled Row Reduction and Secret-Sharing, that use several invocations of H (cf. Eq. 2.5), H can be modeled to be some variant of correlation robust or as PRF, depending on whether free XORs are used or not.

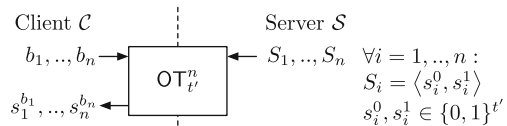
2.2.3 Oblivious Transfer

Parallel 1-out-of-2 OT of n t' -bit strings, denoted as $\text{OT}_{t'}^n$, is a two-party protocol run between a chooser (client \mathcal{C}) and a sender (server \mathcal{S}) as shown in Fig. 2.3: For $i = 1, \dots, n$, \mathcal{S} inputs n pairs of t' -bit strings $s_i^0, s_i^1 \in \{0, 1\}^{t'}$ and \mathcal{C} inputs n choice bits $b_i \in \{0, 1\}$. At the end of the protocol, \mathcal{C} learns the chosen strings $s_i^{b_i}$, but nothing about the other strings $s_i^{1-b_i}$, while \mathcal{S} learns nothing about \mathcal{C} 's choices b_i .

In the following, we assume that OT is used in the context of SFE protocols (as described later in Sect. 2.3.1), i.e., the transferred strings are garbled values with length $t' = t + 1 \sim t$ where t is the symmetric security parameter (cf. Sect. 2.1.1.2).

We describe techniques to efficiently implement OT next.

Fig. 2.3 Parallel 1-out-of-2 OT of n t' -bit strings ($\text{OT}_{t'}^n$)



2.2.3.1 Efficient OT Protocols

$\text{OT}_{t'}^n$ can be instantiated efficiently with different protocols, e.g., [3, 163].

For example the protocol of Naor and Pinkas [163] implemented over a suitably chosen EC consists of three messages ($\mathcal{S} \rightarrow \mathcal{C} \rightarrow \mathcal{S} \rightarrow \mathcal{C}$) in which $2n + 1$ EC points and $2nt'$ encrypted bits are sent. Using point compression, each point can be represented with $2t + 1$ bits and hence the overall communication complexity of this protocol is $(2n + 1) \cdot (2t + 1) + 2nt'$ bits $\approx 6nt$ bits. As a computation, \mathcal{S} performs $2n + 1$ point multiplications and $2n$ invocations of a cryptographic hash function H , modeled as RO, and \mathcal{C} performs $2n$ point multiplications and n invocations of H . This protocol is provably secure against malicious \mathcal{C} and semi-honest \mathcal{S} in the RO model.

Similarly, the protocol of Aiello et al. [3] implemented over a suitably chosen EC using point compression has communication complexity $n(6(2t + 1)) + (2t + 1)$ bits $\sim 12nt$ bits and is secure against malicious \mathcal{C} and semi-honest \mathcal{S} in the standard model as described in [144].

2.2.3.2 Extending OT Efficiently

The extensions of Ishai et al. [121] can be used to reduce the number of computationally expensive public-key operations of $\text{OT}_{t'}^n$ to be independent of n .³ The transformation for semi-honest \mathcal{C} reduces $\text{OT}_{t'}^n$ to OT_t^t (with roles of \mathcal{C} and \mathcal{S} swapped) and a small additional overhead: one additional message, $2n(t' + t)$ bits of additional communication, and $\mathcal{O}(n)$ invocations of a correlation robust hash function H ($2n$ for \mathcal{S} and n for \mathcal{C}) which is substantially cheaper than $\mathcal{O}(n)$ public-key operations. A slightly less efficient OT extension for malicious \mathcal{C} is given in [121] and improved in [166].

2.2.3.3 Pre-Computing OT

All computationally expensive operations for OT can be shifted into a setup phase by pre-computing OT as described in Beaver [23]: In the setup phase, the parallel OT protocol is run on randomly chosen values $r_i \in_R \{0, 1\}$ by \mathcal{C} and $m_i^j \in \{0, 1\}^{t'}$ by \mathcal{S} . In the online phase, \mathcal{C} uses her random bits r_i to mask her private inputs b_i ,

³ This is the reason for our choice of notation $\text{OT}_{t'}^n$ instead of $n \times \text{OT}^{t'}$.

Table 2.5 Complexity of OT_t^n in the RO model

Complexity			Setup phase	Online phase
For $n \leq t$: Beaver [23] + Naor and Pinkas [163]				
Communication		Moves	3	2
		Data [bits]	$6nt$	$2nt$
Computation	Client \mathcal{C}	H	n	
		EC mult	$2n$	
	Server \mathcal{S}	H	$2n$	
		EC mult	$2n + 1$	
For $n > t$: Beaver [23] + Ishai et al. [121] + Naor and Pinkas [163]				
Communication		Moves	4	2
		Data [bits]	$4nt + 6t^2$	$2nt$
Computation	Client \mathcal{C}	H	$n + 2t$	
		EC mult	$2t + 1$	
	Server \mathcal{S}	H	$2n + t$	
		EC mult	$2t$	

and sends the masked bits to \mathcal{S} . \mathcal{S} replies with encryptions of his private inputs s_i^j using his random masks m_i^j from the setup phase. Which input of \mathcal{S} is masked with which random value is determined by \mathcal{C} 's message. Finally, \mathcal{C} applies the masks m_i she received from the OT protocol in the setup phase to decrypt the correct output values $s_i^{b_i}$.

More precisely, the *setup phase* works as follows: For $i = 1, \dots, n$, \mathcal{C} chooses random bits $r_i \in_R \{0, 1\}$ and \mathcal{S} chooses random masks $m_i^0, m_i^1 \in_R \{0, 1\}^{t'}$. Both parties run an $\text{OT}_{t'}^n$ protocol on these randomly chosen values, where \mathcal{S} inputs the pairs $\langle m_i^0, m_i^1 \rangle$ and \mathcal{C} inputs r_i and obtains the masks $m_i = m_i^{r_i}$ as output. In the *online phase*, for each $i = 1, \dots, n$, \mathcal{C} masks its input bits b_i with r_i as $\tilde{b}_i = b_i \oplus r_i$ and sends these masked bits to \mathcal{S} . \mathcal{S} responds with the masked pair of t' -bit strings $\langle \tilde{s}_i^0, \tilde{s}_i^1 \rangle = \langle m_i^0 \oplus s_i^0, m_i^1 \oplus s_i^1 \rangle$ if $\tilde{b}_i = 0$ or $\langle \tilde{s}_i^0, \tilde{s}_i^1 \rangle = \langle m_i^0 \oplus s_i^1, m_i^1 \oplus s_i^0 \rangle$ otherwise. \mathcal{C} obtains $\langle \tilde{s}_i^0, \tilde{s}_i^1 \rangle$ and decrypts $s_i^{b_i} = \tilde{s}_i^{r_i} \oplus m_i$. Overall, the online phase consists of two messages of size n bits and $2nt'$ bits and negligible computation (XOR of bitstrings).

2.2.3.4 OT Complexity

Combining the previously described improvements for pre-computing and extending OT with the efficient OT protocol of Naor and Pinkas [163] yields a highly efficient implementation of OT_t^n in the RO model as summarized in Table 2.5. Similarly, an efficient implementation in the standard model using correlation robust hashing can be obtained by combining with the protocol of Aiello et al. [3] instead.

2.3 Garbled Circuit Protocols

In this section we show how GCs are used in several protocols for secure computation in the two-party (Sect. 2.3.1) and multi-party (Sect. 2.3.2) settings. Further applications of GC such as OTP (Sect. 4.2) or verifiable computing (Sect. 4.3) are described later in this book.

2.3.1 Two-Party Secure Function Evaluation

SFE allows two parties to implement a joint computation without using a TTP. One classical example is the Millionaires Problem [231] where two millionaires want to know who is richer, without either of them revealing their net worth to the other or a TTP.

More formally, SFE is a cryptographic protocol that allows two players, client \mathcal{C} with private input $\text{in}_{\mathcal{C}}$ and server \mathcal{S} with private input $\text{in}_{\mathcal{S}}$, to evaluate a function f on their private inputs:

$$(\text{out}_{\mathcal{C}}, \text{out}_{\mathcal{S}}) = f(\text{in}_{\mathcal{C}}, \text{in}_{\mathcal{S}}). \quad (2.9)$$

The SFE protocol ensures that both parties learn only their respective output, i.e., \mathcal{C} learns $\text{out}_{\mathcal{C}}$ and \mathcal{S} learns $\text{out}_{\mathcal{S}}$, but nothing else about the other party's private input. In SFE, the function f is known to both parties.⁴

Intuitively, according to the real/ideal world paradigm (e.g., [55]), an SFE protocol executed in the real world is secure if and only if an adversary with defined capabilities can do no more harm to the protocol executed in the real world than in an ideal world where each party submits its input to a TTP which computes the results according to Eq. (2.9) and returns them to the respective party.

In Sect. 2.3.1.1 we start with the description of the classical SFE protocol of Yao [231] which is secure against semi-honest adversaries and summarize how this protocol can be secured against more powerful covert and malicious adversaries in Sect. 2.3.1.2. Afterwards, we show how the evaluated function itself can be hidden in Sect. 2.3.1.3.

2.3.1.1 SFE with Semi-Honest Adversaries (Yao's Protocol)

Yao's protocol [145, 151, 231] for SFE of a function f represented as a boolean circuit (cf. Sect. 2.1.3.1) works as follows:

⁴ If needed, SFE can be extended s.t. the function is known to only one of the parties and hidden from the other as described in Sect. 2.3.1.3.

1. *Create GC*: In the *setup phase*, the *constructor* (server \mathcal{S}) generates a GC \tilde{f} using algorithm `createGC` as described in Sect. 2.2.2 and sends \tilde{f} to the *evaluator* (client \mathcal{C}).
2. *Encrypt Inputs*: Afterwards, in the *online phase*, the inputs of the two parties $\text{in}_{\mathcal{C}}$, $\text{in}_{\mathcal{S}}$ are converted into the corresponding garbled input $\tilde{\text{in}} = \{\tilde{\text{in}}_{\mathcal{C}}, \tilde{\text{in}}_{\mathcal{S}}\}$ provided to \mathcal{C} : For \mathcal{S} 's inputs $\text{in}_{\mathcal{S}}$, \mathcal{S} simply sends the garbled values corresponding to his inputs to \mathcal{C} , i.e., $\tilde{\text{in}}_{\mathcal{S},i} = \tilde{\text{in}}_{\mathcal{S},i}^{\text{in}_{\mathcal{S},i}}$. Similarly, \mathcal{C} must obtain the garbled values $\tilde{\text{in}}_{\mathcal{C},i}$ corresponding to her inputs $\text{in}_{\mathcal{C},i}$, but without \mathcal{S} learning $\text{in}_{\mathcal{C},i}$. This can be achieved by running (in parallel for each bit $\text{in}_{\mathcal{C},i}$ of $\text{in}_{\mathcal{C}}$) a 1-out-of-2 OT protocol as described in Sect. 2.2.3.
3. *Evaluate Function Under Encryption*: Now, \mathcal{C} can evaluate the GC \tilde{f} on the garbled inputs $\tilde{\text{in}}$ using algorithm `evalGC` as described in Sect. 2.2.2 and obtains the garbled outputs $\text{out} = \{\text{out}_{\mathcal{C}}, \text{out}_{\mathcal{S}}\}$.
4. *Decrypt Outputs*: Finally, the garbled outputs are converted into plain outputs for the respective party: For \mathcal{C} 's outputs $\text{out}_{\mathcal{C}}$, \mathcal{S} reveals their permutation bits to \mathcal{C} (this can be done already in the setup phase). For \mathcal{S} 's outputs $\text{out}_{\mathcal{S}}$, \mathcal{C} sends the obtained permutation bits to \mathcal{S} .

Security. As proven in detail in [152], Yao's protocol is secure against semi-honest adversaries.

We observe that in Yao's protocol the security of GC constructor \mathcal{S} is *computational* as GC evaluator \mathcal{C} can break the GC by guessing garbled input values, verify if they decrypt correctly and match them with the garbled inputs provided by \mathcal{S} . When instantiating OT with a protocol which provides statistical security for receiver \mathcal{C} (e.g., using the OT protocol of Naor and Pinkas [163]), the security of GC evaluator \mathcal{C} is *statistical*.

Efficiency. The efficiency of Yao's protocol is dominated by the efficiency of the GC construction and OT for each input bit of \mathcal{C} .

As described in Sect. 2.2.3, OT requires only a constant number of public-key operations and allows one to shift most communication and computation into the setup phase. The resulting setup phase requires one to pre-compute $|\text{in}_{\mathcal{C}}|$ OTs (cf. Sect. 2.2.3.4), create the GC \tilde{f} (cf. Sect. 2.2.2.4), and transfer \tilde{f} to \mathcal{C} (cf. Table 2.4).

The online phase is highly efficient as it requires only symmetric-key operations for evaluating \tilde{f} (cf. Sect. 2.2.2.4), and three moves (two for the online phase of pre-computed OT and one for sending the output to \mathcal{S}) with about $t(2|\text{in}_{\mathcal{C}}| + |\text{in}_{\mathcal{S}}|) + |\text{out}_{\mathcal{S}}|$ bits of communication in total.

2.3.1.2 SFE with Stronger Adversaries

GC-based SFE protocols can easily be protected against a covert or malicious client \mathcal{C} by using an OT protocol with corresponding security properties.

Efficient SFE protocols based on GC which additionally protect against a covert [12, 103] or malicious [150] server \mathcal{S} rely on the following cut-and-choose technique: \mathcal{S} creates multiple GCs, deterministically derived from random seeds s_i , and commits

to each, e.g., by sending \tilde{f}_i or $\text{Hash}(\tilde{f}_i)$ to \mathcal{C} . In the covert case, \mathcal{C} asks \mathcal{S} to open all but one GC \tilde{f}_I by revealing the corresponding seeds $s_i \neq I$. For all opened functions, \mathcal{C} computes \tilde{f}_i and checks that they match the commitments. The malicious case is similar, but \mathcal{C} asks \mathcal{S} to open half of the functions, evaluates the remaining ones and chooses the majority of their results. Additionally, it must be guaranteed that \mathcal{S} 's input into OT is consistent with the GCs as pointed out in [138], e.g., using committed or committing OT. The most recent construction of [153] improves over previous protocols (smaller number of GCs, completely removing the commitments, and also removing the need to increase the size of the inputs) by using a new primitive called cut-and-choose OT, an extension of parallel 1-out-of-2 OT with a cut-and-choose functionality.

The practical performance of cut-and-choose-based GC protocols has been investigated experimentally in [154, 180]: Secure evaluation of the AES functionality (a boolean circuit with 33,880 gates) between two Intel Core 2 Duos running at 3.0 GHz, with 4 GB of RAM connected by a Gigabit ethernet takes approximately 0.5 MB data transfer and 7s for semi-honest, 8.7 MB/1 min for covert, and 400 MB/19 min for malicious adversaries [180]. This shows that protecting GC protocols against stronger adversaries comes at a relatively high prize.

For completeness, note that cut-and-choose may be avoided with SFE schemes such as [125] which prove in zero-knowledge that the GC was computed correctly and the inputs are consistent with committed inputs [88]. However, their elementary steps involve public-key operations. As estimated in [180], such protocols which apply public-key operations per gate [125, 168] often require substantially more computation than cut-and-choose-based protocols.

We further note that there are yet other approaches to malicious security such as the approach of [123] which achieves malicious security by simulating a SMPC protocol inside a secure two-party computation protocol with semi-honest security. Their precise performance comparison is a desirable but complicated undertaking, since there are several performance measures, and some schemes may work well only for certain classes of functions.

2.3.1.3 SFE with Private Functions

In some application scenarios of SFE, the evaluated function itself needs to be hidden, e.g., as it represents intellectual property of a service provider. This can be achieved by securely evaluating a Universal Circuit (UCi) which can be programmed to simulate any circuit C and hence entirely hides C (besides an upper bound on the number of inputs, number of gates and number of outputs). Efficient UCi constructions to simulate circuits consisting of up to k two-input gates are given in [143, 221]. Generalized UCis of [184] can simulate circuits consisting of d -input gates. Which UCi construction is favorable depends on the size of the simulated functionality: Small circuits can be simulated with the UCi construction of [184, 194] with overhead $\mathcal{O}(k^2)$ gates, medium-size circuits benefit from the construction of [143] with overhead $\mathcal{O}(k \log^2 k)$ gates and for very large circuits the

construction of [221] with overhead $\mathcal{O}(k \log k)$ gates is most efficient. Explicit sizes and a detailed analysis of the break-even points between these constructions are given in [184]. The alternative approach of [136] for evaluating private functions without using UCis has complexity linear in k , but requires $\mathcal{O}(k)$ public-key operations.

While UCis entirely hide the structure of the evaluated functionality f , it is sometimes sufficient to hide f only within a class of topologically equivalent functionalities \mathcal{F} , called secure evaluation of a *semi-private* function $f \in \mathcal{F}$ [177]. The circuits for many standard functionalities are topologically equivalent and differ only in the specific function tables, e.g., comparison ($<$, $>$, $=$, \dots) or addition/subtraction, as described later in Sect. 3.3. When no cut-and-choose is used for GCs, it is possible to directly evaluate the circuit and avoid the overhead of a UCi for semi-private functions, as GC constructions of [157] and [164] (cf. Sect. 2.2.2.3) completely hide the type of the gates from the GC evaluator. These techniques were used for example in [83–86, 177].

2.3.2 Garbled Circuit Protocols with Multiple Parties

GCs can also be used for SMPC, i.e., secure computation with more than two parties. In the following we describe applications of GCs to SMPC in Sect. 2.3.2.1 and secure mobile agents in Sect. 2.3.2.2.

In the multi-party setting, one party, the GC *creator*, which is assumed to behave correctly, creates the GC (cf. algorithm `createGC` in Sect. 2.2.2.2); another party, the GC *evaluator*, obviously obtains the corresponding garbled inputs and evaluates the GC (cf. algorithm `evalGC` in Sect. 2.2.2.2). The other parties provide inputs to or obtain outputs from the protocol.

We will show later in Chap. 4 that the GC creator can be implemented with constant-size memory, e.g., within a tamper-proof HW token.

Verifiability of GC. As discussed in detail in Chap. 4, the GC evaluator, who evaluates the GC on the garbled inputs, need not be trusted at all. Indeed, GC evaluation can be performed by one or more *untrusted* parties as the garbled outputs allow verification that the GC evaluation was done correctly [164]: For each garbled output \tilde{z}_i , the GC creator provides the *output decryption information* $\langle 0, G(\tilde{z}_i^0) \rangle, \langle 1, G(\tilde{z}_i^1) \rangle$, where G is a one-way function (e.g., a cryptographic hash function). This allows one to check whether \tilde{z}_i is correct, i.e., either $\tilde{z}_i = \tilde{z}_i^0$ or $\tilde{z}_i = \tilde{z}_i^1$, and which is the corresponding plain value without revealing the values \tilde{z}_i^0 and \tilde{z}_i^1 . As the GC evaluator is unable to guess a correct \tilde{z}_i (except with negligible probability), she must have obtained it by honestly evaluating the GC.

2.3.2.1 SMPC with Two Servers

As proposed in [164], Yao’s GC protocol (cf. Sect. 2.3.1.1) can be turned into a SMPC protocol with multiple input players, multiple output players, and two

non-colluding computation players who perform the secure computation: the GC creator is trusted by the output players to behave semi-honestly and the GC evaluator can even be malicious.

For multiple input players, the parallel 1-out-of-2 OT protocol (cf. Sect. 2.2.3) is replaced with a parallel 1-out-of-2 proxy OT protocol. The proxy OT protocol splits the role of the chooser in the OT protocol into two parties: the *chooser* (input player) provides the secret input bit b , and the *proxy* (the GC evaluator) learns the chosen output string s^b , but neither b nor s^{1-b} . As described in [164, Appendix A], efficient OT protocols (e.g., the protocols of Aiello et al. [3], Naor and Pinkas [163] described in Sect. 2.2.3) can be naturally converted into a proxy OT protocol as follows: The chooser sends the two public keys, of which she knows the trapdoor to exactly one, to the sender. The sender applies an error-correcting code to each of the two strings s^0, s^1 and sends their encryptions under the respective public key to the proxy. The proxy uses the trapdoor obtained by the chooser to decrypt both ciphertexts obtained from the sender and uses the error correcting code to compute s^b .

For multiple output players, the GC evaluator forwards the garbled outputs to the respective output player who can decrypt and verify the correctness of the output using the output decryption information obtained from the GC creator.

2.3.2.2 Secure Mobile Agents

In the *mobile agents* scenario, the *originator* creates SW agents that can perform tasks on behalf of the originator. After creating the agents for some specific purpose, the originator sends them out to visit various remote hosts, where the agents perform computations on behalf of the originator. When the agents return home, the originator retrieves the results of these computations from the agents. The utility of this paradigm is based on the ability of the originator to go offline after sending the agents out, and, ideally, no further interaction between the agent and the originator or the host should be required. A possible application would be an agent which travels through the web to select, depending on a policy of the originator, an offer for the most suitable product at the lowest price.

Secure mobile agents extend the mobile agents scenario with security features. Here, the visited hosts are not trusted by the originator and vice versa. When an agent visits a host, it carries along some state from previous computations and uses this together with input from the host to compute the new agent state possibly along with an output provided to the host. The agent state (both old and new) is “owned” by the agent, and should be protected from potentially malicious hosts, whereas the host input and output are “owned” by the host and should likewise be protected from potentially malicious agents. The code evaluated by the agent (policy) can be hidden as well by evaluating a UCi (cf. Sect. 2.3.1.3).

The concept of secure mobile agents was introduced in [192] who give partial solutions based on HE (Sect. 2.2.1). More practical constructions for secure mobile agents proposed afterwards are based on GCs: An agent can securely migrate from one host to the next by running a (slightly modified) GC-based SFE protocol

(cf. Sect. 2.3.1) between the two hosts as described in [53]. To protect against malicious hosts, a TTP can be used to generate the GCs, similarly to the GC constructor in the construction of Naor et al. [164] (cf. Sect. 2.3.2.1), as proposed in [5]. The assumption of the TTP was later removed in [212] and a construction which achieves universal composability is given in [229]. Finally, non-interactive OT based on trusted HW reduces the communication overhead to the essential minimum where the agent is sent from one host to the next in a single message [106].

<http://www.springer.com/978-3-642-30041-7>

Engineering Secure Two-Party Computation Protocols
Design, Optimization, and Applications of Efficient
Secure Function Evaluation

Schneider, Th.

2012, XVI, 138 p. 35 illus., Softcover

ISBN: 978-3-642-30041-7