Practical Secure Function Evaluation

Diplomarbeit im Fach Informatik

vorgelegt von

Thomas Schneider

geb. 1. Juni 1983 in Koblenz

angefertigt am

Institut für Informatik
Lehrstuhl für Informatik 8
Künstliche Intelligenz
Friedrich-Alexander-Universität Erlangen–Nürnberg
Prof. Dr. Volker Strehl

Betreuer: Vladimir Kolesnikov, Ph.D.
(Bell Laboratories, Security Solutions/Cryptographic Systems,
600 Mountain Ave. Murray Hill, NJ 07974,USA)

Beginn der Arbeit: 7. September 2007
Abgabe der Arbeit: 27. Februar 2008

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, den 27. Februar 2008

Thomas Schneider

Abstract

This thesis focuses on practical aspects of general two-party Secure Function Evaluation (SFE). We give a new SFE protocol that allows free evaluation of XOR gates and is provably secure against semi-honest adversaries in the random oracle model.

Furthermore, the extension of SFE to private functions (PF-SFE) using universal circuits (UC) is considered. Based on our new practical UC construction, FairplayPF is implemented as extension of the well-known Fairplay SFE system to demonstrate practicability of UC-based PF-SFE.

Also new protocols for SFE and PF-SFE of functions alternatively represented as Ordered Binary Decision Diagram (OBDD) are given.

Synopsis

Since the first publication of Yao [Yao86], Secure Function Evaluation (SFE) is a well-researched problem. Continuing advances in available computational power and communication have made secure computation of many useful functions affordable. Recent work like Fairplay [MNPS04] demonstrate practicability of general SFE. This thesis focuses on several practical aspects of SFE.

Our new improved SFE protocol allows free evaluation of XOR gates and is provably secure against semi-honest adversaries in the random oracle model - the same assumptions that Fairplay relies on. The protocol merges elements of the information-theoretic SFE protocol GESS [Kol05] with Fairplay. This results in substantial performance improvements of 50% for many important circuit structures like addition or equality test.

SFE is extended to allow the evaluated function to be secret and only known by one party, called SFE of private functions (PF-SFE). These settings occur naturally in applications like no-fly-list-, credit report-, or medical history checking. It is known that PF-SFE can easily be reduced to SFE of universal circuits (UC). We give a practical UC construction [KS08] that is up to 50% smaller than the best UC of Valiant [Val76] when used in today's PF-SFE. FairplayPF [KS] was implemented as extension of Fairplay to demonstrate practicability of PF-SFE based on the new UC construction. Using the improved SFE protocol, UC-based PF-SFE can be improved by another factor of 4.

Besides these circuit-based approaches for SFE and PF-SFE new protocols for SFE and PF-SFE of functions represented as Ordered Binary Decision Diagrams (OBDDs) are given that are based on [KJGB06]. This SFE protocol for OBDDs is extended to the malicious model and shown how to obtain a PF-SFE protocol for OBDDs at the cost of a small overhead only.

The results of this thesis substantially improve general SFE for many practical functions and demonstrate practicability of general PF-SFE for "small" functions.

Kurzfassung

Diese Arbeit konzentriert sich auf praktische Aspekte allgemeiner, sicherer Funktionsauswertung (SFE, engl. Secure Function Evaluation) zwischen zwei Teilnehmern. Ein neues SFE Protokoll wird präsentiert, das eine kostenlose Auswertung von XOR Gattern erlaubt und beweisbar sicher gegen "semi-honest" Angreifer im "Random Oracle" (RO) Modell ist.

Zusätzlich wird die Erweiterung von SFE auf private Funktionen (PF-SFE) unter Verwendung von universellen Schaltkreisen (UC, engl. Universal Circuit) betrachtet. Basierend auf unserer neuen, praktischen UC Konstruktion wurde FairplayPF als Erweiterung des namhaften Fairplay SFE Systems implementiert, um die Praktikabilität von UC-basierter PF-SFE zu demonstrieren.

Außerdem werden für Funktionen, die eine alternative Darstellung als geordnete, binäre Entscheidungsdiagramme (OBDD, engl. Ordered Binary Decision Diagrams) haben, neue SFE- und PF-SFE Protokolle angegeben.

Zusammenfassung

Seit der ersten Veröffentlichung von Yao [Yao86] ist sichere Funktionsauswertung (SFE, engl. Secure Function Evaluation) ein gut erforschtes Gebiet. Anhaltende Erhöhungen der verfügbaren Rechen- und Kommunikationsressourcen haben die sichere Auswertung vieler nützlicher Funktionen erschwinglich gemacht. Neue Veröffentlichungen wie Fairplay [MNPS04] demonstrieren die Praktikabilität von allgemeiner SFE. Diese Arbeit konzentriert sich auf verschiedene praktische Aspekte von SFE.

Unser neues, verbessertes SFE Protokoll erlaubt die kostenlose Auswertung von XOR Gattern und ist beweisbar sicher gegen "semi-honest" Angreifer im "Random Oracle" (RO) Modell - die selben Annahmen, auf denen auch Fairplay basiert. Das Protokoll kombiniert Elemente des informationstheoretischen SFE Protokolls GESS [Kol05] mit Fairplay. Daraus resultiert eine erhebliche Effizienzverbesserung auf 50% für viele bedeutende Schaltkreisstrukturen wie Addition oder Gleichheitstest.

SFE wird erweitert, so dass die ausgewertete Funktion geheim und nur einem Teilnehmer bekannt ist. Dies wird als SFE privater Funktionen (PF-SFE) bezeichnet. Praktische Anwendungen hierfür sind das Prüfen von Flugverbotslisten, der Kreditwürdigkeit oder der Krankheitsgeschichte. Es ist bekannt, dass PF-SFE leicht auf SFE von universellen Schaltkreisen (UC, engl. Universal Circuit) reduziert werden kann. Wir präsentieren eine praktische UC Konstruktion [KS08], die bis zu 50% kleiner als der bisher kleinste UC von Valiant [Val76] ist, wenn sie für heutige PF-SFE verwendet wird. FairplayPF [KS] wurde als Erweiterung von Fairplay implementiert, um die Praktikabilität von PF-SFE basierend auf der neuen UC Konstruktion zu demonstrieren. Bei Verwendung des verbesserten SFE Protokolls kann UC-basierte PF-SFE um einen weiteren Faktor von 4 verbessert werden.

Neben diesen schaltkreisbasierten Ansätzen für SFE und PF-SFE werden neue Protokolle für Funktionen vorgestellt, die als geordnete, binäre Entscheidungsdiagramme (OBDD, engl. Ordered Binary Decision Diagrams) repräsentiert sind. Das SFE Protokoll für OBDDs basiert auf dem Protokoll aus [KJGB06], das auf das "malicious model" ausgeweitet wird. Zusätzlich wird gezeigt, wie man daraus ein PF-SFE Protokoll für OBDDs mit nur geringem Zusatzaufwand erhält.

Die Ergebnisse dieser Diplomarbeit verbessern allgemeine SFE für viele praktische Funktionen und demonstrieren die Praktikabilität von allgemeiner PF-SFE für "kleine" Funktionen.

Acknowledgements

First and foremost I would like to thank Vladimir Kolesnikov for his kind supervision of this thesis and my six months research internship at Bell Labs, Security Solutions/ Cryptographic Systems in Murray-Hill, NJ, USA. His introduction into state of the art research and methodologies in SFE, many fruitful hints, discussions and tons of thrown away "crappy" drafts and ideas substantially contributed to the results of this thesis. He gave me a very positive insight into scientific research and constructive cooperation. The possibility to publish the results we found together on international conferences combined with the two provisional patent applications were an outstanding opportunity. Thanks for the nice time we worked and spent together in and around "the city". I'm looking forward to continuing our joint research in the future.

Special thanks also goes to Prof. Dr. Volker Strehl for kindly supervising this thesis and encouraging me to go abroad during one semester for it. His interesting lectures, motivating exercise classes and kind mentorship guided me throughout my studies and formed my interest in theoretical computer science - especially cryptography. The support he gave opened plenty of opportunities during my studies and for continuing scientific research in the future.

Thanks to my beloved girlfriend Karolin Steidel for her understanding, support and encouragements during the last year and our wonderful time together.

Last but not least I would like to thank all people that contributed to parts of this thesis with helpful comments, Prof. Dr. Rolf Wanka, Debbie Cook and as reviewers, my father Klaus, my brother Matthias, and my university friends Korbinian Riedhammer and Thomas Holleczek.

Thank you.

Contents

1 Introduction

Consider the following situation. Several parties, each of which has a private input, wish to evaluate a function on their inputs. This need arises often indeed – almost any transaction or a communication over a network can be cast as evaluation of a function on the participant's inputs. For example, in an online auction, the bidders and the auctioneer are players who wish to evaluate the auction function, whose value is equal to the ID of the highest bidder. Other natural and important examples include financial transactions, voting, distributed database mining, etc.

A lot of the time, parties' inputs need to be hidden from the rest of the world. For example, in the case of the auction, unsuccessful bidders would want to preserve the privacy of their bids. Depending on the function, some information about the parties' inputs might be easily derived from the output of the function. For example, the winning bid of an auction might necessarily be revealed. The goal of secure computation is to ensure that no other information is leaked during the computation. Clearly, efficient methods of secure evaluation of functions are of great interest. The problem is often referred to as Secure Function Evaluation, or SFE.

As SFE is a very wide area of research this thesis focuses on the following subclass of SFE protocols. *Two-party* SFE is considered, where the number of players is two. These protocols can be run in *one-round* that allows relatively easy protection against malicious players trying to cheat in the protocol (whereas often multi-party protocols run in multiple rounds [GMW87, Can96]). *General* SFE protocols are discussed that can evaluate arbitrary functions (whereas special-purpose SFE protocols might be more efficient but can only be used for a subclass of functions and applications such as number comparison [BK06] for auction functions, etc.). *Computationally secure* protocols are considered where adversaries' resources are bounded polynomially (whereas in information-theoretically secure protocols adversaries may have unlimited power). A wide range of practical applications such as privacy-preserving no-fly-list-/credit history-/medical history checking, mobile code, or privacy-preserving database querying is covered with this subclass of SFE protocols as described in this thesis.

Continuing advances in available computational power and communication have made secure computation of many useful functions affordable. Several recent works approach the problem of general two-party SFE from the practical angle, discuss and fine-tune the implementation details [KJGB06, MNPS04]. By extending and improving this work general two-party SFE is pushed further towards practicability.

1.1 Contents and Contributions

As discussed before, this thesis focuses on practical aspects of general two-party SFE. The currently best known approach for general SFE of Fairplay [MNPS04] is combined with the information-theoretically secure approach of Gate Evaluation Secret Sharing (GESS) [Kol05] to a new, practical method for general SFE, to which we refer to as *improved SFE* in the following. This new method allows free evaluation of XOR gates which results in substantial performance improvements of up to 50% for many important circuit structures such as addition or number comparison. A proof of security in the semi-honest model is given that is based on the random oracle (RO) assumption (same as Fairplay).

In practice, there is often a need to not only protect the inputs, but the function being evaluated as well. One example is checking a passenger against the no-fly list (or, more generally, no-fly function of passenger's data). Here, a compromise of the function weakens the security of the system significantly. Other examples include credit checking or background- and medical history checking functions.

This well-known problem called SFE of *private functions* (PF-SFE) is addressed by a large amount of work like [FAZ05, CCKM00, SYY99, Pin02]. In PF-SFE, the evaluated function is known only by one party and needs to be kept secret (i.e. everything besides the size, the number of inputs and outputs is hidden from the other party). Full or even partial revelation of these functions opens vulnerabilities in the corresponding process, exploitable by dishonest participants (e.g. credit applicants), and should be prevented.

The problem of PF-SFE can be reduced to the "regular" SFE by parties evaluating a *Universal Circuit* (UC) instead of a circuit defining the evaluated function [SYY99, Pin02]. UC can be thought of as a "program execution circuit", capable of simulating any circuit C of certain size, given the description of C as input. Therefore, disclosing the UC does not reveal anything about C, except its size. At the same time, the SFE computes output correctly and C remains private, since the player holding C simply treats description of C as additional (private) input to SFE. This reduction is the most common (and often the most efficient) way of securely evaluating private functions [SYY99, Pin02].

We improve previous PF-SFE constructions by giving a new simple and efficient UC construction [KS08]. Our *practical UC construction* for simulating k gates has size $\sim 1.5k \log^2 k$ and depth $\sim k \log k$ (depth can be improved to $O(k)$ at the cost of a small increase in size). It is up to 50% smaller than the best UC (Valiant's UC [Val76] has size $\sim 19k \log k$) for practical circuit sizes of up to ≈ 5000 gates. This improvement results in corresponding performance improvement of SFE of (small) private functions. Since, due to cost, only small circuits (i.e. < 5000 gates) are practical for PF-SFE, our construction appears to be the best fit for many practical PF-SFE.

General PF-SFE is implemented using this UC construction as extension of the Fairplay SFE system, called *FairplayPF* [KS]. Using our improved SFE protocol, UC-based

PF-SFE can be improved to cost as little as 25% of the previously best known solution (which uses Fairplay as underlying SFE protocol).

Besides these circuit-based approaches for SFE and PF-SFE, new protocols for SFE and PF-SFE of functions represented as Ordered Binary Decision Diagrams (OBDDs) are given that are based on [KJGB06]. This SFE protocol for OBDDs is extended to the malicious model and shown how to obtain a PF-SFE protocol for OBDDs at the cost of a small overhead only.

The results of this thesis substantially improve general SFE for many practical circuits and demonstrate practicability of general PF-SFE for "small" functions.

1.2 Publications and Copyright Notice

Parts of this thesis are published/submitted by Vladimir Kolesnikov and Thomas Schneider in/to two international conferences.

a) Chapter 4.2 and Chapter 5: "A Practical Universal Circuit Construction and Secure Evaluation of Private Functions". In Financial Cryptography and Data Security 2008 (FC08). [KS08]

b) Chapter 3.3.4: "Improved Garbled Circuit: Free XOR Gates and Applications". Submitted to International Colloquium on Automata, Languages and Programming (ICALP08).

In order to emphasize that these parts of the thesis are published, formulated and patented in close cooperation with my supervisor Vladimir Kolesnikov, I intensionally use formulations "we" and "our" in combination with these results.

1.3 Structure

The thesis is structured as follows:

Chapter 2 introduces relevant primitives and notation used throughout this thesis.

Chapter 3 gives practical Secure Function Evaluation (SFE) protocols.

Chapter 4 extends SFE to private functions, called PF-SFE.

Chapter 5 describes universal circuit (UC) constructions suitable for PF-SFE.

Chapter 6 explains FairplayPF, an extension of Fairplay SFE system for PF-SFE.

Chapter 7 concludes the results of the thesis and gives and outlook to future work.

2 Notation, Definitions and Preliminaries

This chapter introduces relevant primitives and notation used throughout this thesis.

2.1 Notation

This thesis uses the following standard notations.

$\{0,1\}^*$ denotes the space of finite binary strings, $\{0,1\}^n$ the space of binary strings of length n. $a||b$ or just ab denotes the concatenation of strings a and b. $\langle a, b\rangle$ is a vector with two components a and b, and its bit string representation is $a||b$.

For strings of the same length $s, t \in \{0,1\}^*$, let $s \oplus t$ denote their *bitwise exclusive-or* (XOR).

Uniformly random sampling is denoted by the $\in_R$ operator. For example, $r \in_R D$ means "choose r uniformly at random from D".

$X \stackrel{c}{\equiv} Y$ denotes that random variables X and Y are *computationally indistinguishable* (see [Gol01, 3.2 Computational Indistinguishability] for an exact definition of computational indistinguishability).

Abbreviations are introduced before they are used. A list of acronyms can be found in the end of this thesis.

Throughout this thesis parties are referred to mainly by their function (Evaluator, Constructor) or by their index (P_i). Sometimes names are used, where Alice is the Evaluator and Bob is the Constructor, depending on the context of the particular problem.

2.2 Boolean Functions

A boolean function $f : \{0,1\}^u \rightarrow \{0,1\}^v$ is a function that maps u binary inputs to v binary outputs. The following representations of boolean functions are especially relevant as they are the basis of SFE protocols described in this thesis.

2.2.1 Boolean Circuits

Boolean circuits are a classical representation of boolean functions used in complexity theory and they also correspond to the circuit diagrams of stateless digital logic in

hardware. A boolean circuit with u inputs, v outputs and k gates is a finite *directed acyclic graph (DAG)* with $|V| = u + v + k$ 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 omitted in the graphical representation of a circuit as seen in Fig. 2.1. For a more detailed definition see [Vol99].

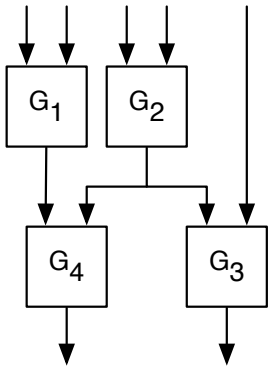


Figure 2.1: *Boolean Circuit* with $u = 5$ inputs, $v = 2$ outputs, $k = 4$ gates $G_1, \ldots, G_4$ in topologic order and fan-out 2.

Any DAG and hence any boolean circuit can be topologically sorted efficiently in $O(|V| + |E|)$ [CLRS01, Topological sort, pp. 549-552]. The topologic order of a boolean circuit labels the gates with $G_1, \ldots, G_k$ and ensures that the i-th gate G_i has no inputs that are outputs of a successive gate G_j, where $j > i$. In complexity theory, a circuit with such a topologic order is called a *straight-line program* [ALR99]. 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. Fig. 2.1 shows a boolean circuit with topologically ordered gates $G_1, \ldots, G_4$. Note, the topologic order is not necessarily unique.

$W_{out} = G(W_{in_1}, \ldots, W_{in_d})$ denotes a gate G which has d input wires $W_1, \ldots, W_d$ and output wire W_{out}. Gate G computes a d-ary boolean function $g : \{0,1\}^d \to \{0,1\}$ like XOR, AND, OR, NAND, NOR or any other boolean function that can be expressed by a function table with 2^d entries.

The *fan-out* of a boolean circuit is the maximum out-degree of its underlying DAG. The circuit in Fig. 2.1 has fan-out 2 as G_2's output is the input of two gates.

2.2.2 Ordered Binary Decision Diagrams (OBDDs)

Another possibility to represent boolean functions are OBDDs (Ordered Binary Decision Diagrams) which were introduced by Bryant [Bry86]. A definition of OBDDs is given in [KJGB06]:

"Given a Boolean function $f(x_1, x_2, \ldots, x_n)$ of n variables $x_1, \ldots, x_n$ and a total ordering on the n variables, the OBDD for f, denoted by $OBDD(f)$, is a rooted, directed acyclic graph (DAG) with two types of vertices: *terminal* and *non-terminal* vertices. $OBDD(f)$ also has the following components:

- Each vertex v has a level, denoted by $level(v)$, between 0 and n. There is a distinguished vertex called *root* whose level is 0.

- Each nonterminal vertex v is labeled by a variable $var(v) \in \{x_1, \ldots, x_n\}$ and has two successors, $low(v)$ and $high(v)$. Each terminal vertex is labeled with either 0 or 1. There are only two terminal vertices in an OBDD. Moreover, the labeling of vertices respects the total ordering $<$ on variables, i.e., if u has nonterminal successor v, then $var(u) < var(v)$.

Given an assignment $\mathcal{A} = \langle x_1 \leftarrow b_1, \ldots, x_n \leftarrow b_n \rangle$ to the variables $x_1, \ldots, x_n$, the value of the Boolean function $f(b_1, \ldots, b_n)$ can be found by starting at the root and following the path where the edges on the path are labeled with $b_1, \ldots, b_n$. OBDDs can also be used to represent functions with finite range and domain. Let g be a function of n Boolean variables with output that can be encoded by k Boolean variables. The function g can be represented as an array of k OBDDs where the i-th OBDD represents the Boolean function corresponding to the i-th output bit of g."

As in this paper, function f is assumed to be a boolean function with one output only in the following. The protocols can be easily extended for the case of functions with a finite range.

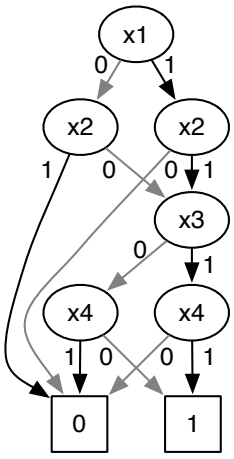


Figure 2.2: OBDD for the function $f(x_1, x_2, x_3, x_4) = (x_1 = x_2) \wedge (x_3 = x_4)$.

Example: "Fig. 2.2[1] shows the OBDD for the function $f(x_1, x_2, x_3, x_4) = (x_1 = x_2) \wedge (x_3 = x_4)$ of four variables x_1, x_2, x_3, x_4 with the total ordering $x_1 < x_2 < x_3 < x_4$.[2] Notice that the ordering of the labels on the vertices on any path from the root to the terminals of the OBDD corresponds to the total ordering of the Boolean variables. Consider the assignment $\langle x_1 \leftarrow 1, x_2 \leftarrow 1, x_3 \leftarrow 0, x_4 \leftarrow 0\rangle$. In the OBDD shown in Fig. 2.2, if we start at the root and follow the edges corresponding to the assignment, we end up at the terminal vertex labeled with 1. Therefore, the value of $f(1, 1, 0, 0)$ is 1."

Bryant mentions that OBDDs are in many cases a reasonable representation of boolean functions [Bry86]: "Although a function requires, in the worst case, a graph of size exponential in the number of arguments, many of the functions encountered in typical applications have a more reasonable representation."

However, there are also functions encountered in typical applications that have an efficient circuit representation but a lower bound for the size of the smallest OBDD representation which is exponential. For example integer multiplication with at least exponential OBDD representation [Bry91, Woe05] but efficient circuit representation of size $O(N \log N \log\log N)$ [SS71] which was recently further improved to $N \log N 2^{O(\log^* N)}$ [Für07]. Furthermore, finding an optimal variable ordering of OBDDs is NP-complete [BW96].

Although this restricts the generality of OBDDs, in many practical cases the OBDD can be minimized to a reasonable size. Algorithms to improve the variable ordering of OBDDs are Rudell's sifting algorithm [Rud93], the window permutation algorithm [FMK91], genetic algorithms [DBG96, LB05], or algorithms based on simulted annealing [BLW95].

2.3 Symmetric Encryption

Most of the protocols presented in this thesis use symmetric encryption schemes. This section summarizes the needed definitions. A good introduction to the basic concepts of encryption schemes is given in [Gol04, Chapter 5, Encryption Schemes].

[Gol04, Definition 5.1.1 (encryption scheme)]: An *encryption scheme* is a triple, (G, E, D), where G is the probabilistic *key-generation algorithm* that outputs on input 1^n a pair of bit strings (e, d), the encryption key e and the decryption key d. The parameter n serves as the *security parameter* of the encryption scheme. Encryption algorithm E and the decryption algorithm D satisfy for any message $m \in \{0,1\}^*$:

$$Pr[D_d(E_e(m)) = m] = 1 \tag{2.1}$$

[1] This figure is a correction of Figure 1 in [KJGB06] which contained a small error.

[2] "OBDDs are sensitive to variable ordering, e.g., with the ordering $x_1 < x_3 < x_2 < x_4$ the OBDD for f has 11 nodes."

In a symmetric encryption scheme, the encryption key equals the decryption key: $e = d$.

This basic definition defines only the correctness of an encryption scheme but says nothing about its security.

2.3.1 Semantic Security

A fundamental definition of security is *semantic security* [Gol04, 5.2 Definitions of Security]: "Semantic security is a computational complexity analogue of Shannon's definition of perfect privacy which requires that the ciphertext yields no information regarding the plaintext). Loosely speaking an encryption scheme is semantically secure if it is infeasible to learn anything about the plaintext from the ciphertext (i.e. impossibility is replaced by infeasibility)."

Goldreich shows, that the definition of semantic security is equivalent to *indistinguishable encryptions* which "interprets security as the infeasibility of distinguishing between encryptions of a given pair of messages. This definition is useful in demonstrating the security of a proposed encryption scheme and for the analysis of cryptographic protocols that utilize an encryption scheme."

Summarized, a semantically secure encryption scheme allows an adversary who does not know the encryption key neither to recover any information about the plaintext from the ciphertext, nor to distinguish whether a ciphertext is the encryption of a known plaintext or not.

2.3.2 Block Ciphers

A *block cipher* is an encryption scheme that operates on plaintexts of a specific length (which is a function of the security parameter). A block cipher can easily be extended to a general secure encryption scheme by block-wise encryption [Gol04, Construction 5.3.7].

[Gol04, Construction 5.3.9] shows a private-key block-cipher which is based on pseudorandom functions and is proven semantically secure: $E_k(x) = (r, V(k,r) \oplus x), D_k(r,y) = V(k,r) \oplus y$ where $r \in_R \{0,1\}^n$ and $V(k,r)$ is a pseudorandom function that can be constructed using any non-uniformly strong one-way functions.

Today, a very common practical instantiation for a block cipher is AES [NIS01] (Advanced Encryption Standard) which is not provably secure but widely believed to be practically secure within the next years. The U.S. Government announced that AES may also be used to protect classified information [NSA03]: "The design and strength of all key lengths of the AES algorithm (i.e., 128, 192 and 256) are sufficient to protect classified information up to the SECRET level. TOP SECRET information will require use of either the 192 or 256 key lengths. The implementation of AES in products in-

tended to protect national security systems and/or information must be reviewed and certified by NSA prior to their acquisition and use."

2.3.3 Symmetric Encryption with Special Properties

Some of the protocols presented in this thesis require two special properties for the semantically secure private-key encryption scheme that allow to decide whether a decryption succeeded or not:

- *elusive range*: an encryption under one key is in the range of an encryption with a different key with negligible probability, and
- *efficiently verifiable range*: given a key, a user can efficiently verify that a ciphertext is in the range of that key.

These special properties can be easily obtained from a semantically secure encryption scheme by appending 0^n to the plaintext as described in [LP04, Section 3.1].

2.4 Random Oracle Model

Some practical SFE protocols are provably secure in the *Random Oracle Model* that allows the usage of *Random Oracles (RO)*. The ability to access this very powerful primitive often results in substantial performance improvements compared to protocols that are secure in the *Standard Model* (using no ROs).

A *Random Oracle (RO)* is a randomly chosen function that maps any query from its finite input domain to a sequence of uniformly chosen random bits – a large object which cannot be fully stored or traversed by polytime players. The RO always answers to the same query with the same random sequence.

RO model gives oracle access to such a function to all players. Although it was shown [CGH98] that a protocol secure in the RO model may not be secure once RO is instantiated, "natural" RO protocols maintain their security in practice. *RO paradigm* [BR93, Bel98] shows how to use ROs to develop practical, provably secure (in the *RO model*) protocols: provide all parties, good and bad alike, with access to a (public) function H; prove correct a protocol assuming H is truly random, i.e. a RO; later, in practice, set H to some specific function derived in some way from a standard cryptographic hash function like SHA (Secure Hash Algorithm) [NIS02] or RIPEMD-160 [DBP96]. The protocol of Chapter 3.3.4 follows this paradigm and is provably secure in the Random Oracle Model.

2.5 Oblivious Transfer

Oblivious Transfer (OT) is a basic and very powerful cryptographic primitive executed between two parties. The protocols presented in this thesis use *1-out-of-2 OT* (OT_1^2) where one party, the *Sender*, inputs two values x_0 and x_1 and the other party, the *Receiver*, inputs his secret choice σ to the protocol. In the end of the OT_1^2 protocol, Receiver learns the chosen input x_σ while Sender learns nothing about the choice σ.

One of the most important applications of OT is SFE, as SFE can be reduced to a parallel execution of one-round OT_1^2 in an unconditional sense [Kil88, Kol06] or more efficiently based on cryptographic hardness assumptions. This reduction is explained in detail in Chapter 3.

Protocols for one-round OT_1^2 are a widely studied primitive in the Standard Model [BM89, AIR01] and improved implementations in the RO Model [NP01, BR93] that minimize the number of expensive modular exponentiations.

2.6 Adversaries

In the context of this thesis, the adversary is not a potential "man-in-the-middle" that attacks the communication channel between the parties and tries to read or modify exchanged messages. Standard cryptographic techniques like message authentication codes, symmetric encryption and authenticated key-exchange protocols can be used to establish a secure channel between the parties and lock out this type of adversary.

The adversaries that are considered in SFE are the communicating parties themselves that try to learn additional information during or after the execution of the protocol. Depending on their power there are two different types of adversaries:

- "*semi-honest* (or passive) adversaries follow the protocol specification, but attempt to learn additional information by analyzing the transcript of messages received during execution." [LP04]
- *malicious* (or active) adversaries may arbitrarily deviate from the protocol specification in order to learn additional information. Clearly, they are more powerful than semi-honest adversaries.

A detailed high level discussion of these different types of adversaries is given in [Kol06, Chapter 2.2].

All protocols presented in this thesis are provably secure against semi-honest adversaries as described in Chapter 3.2.2 and can be extended to be secure against malicious adversaries as described in Chapter 3.2.3.

3 Secure Function Evaluation (SFE)

3.1 Introduction

Secure Function Evaluation (SFE) allows two participants to implement a joint computation that, in real life, may be implemented using a trusted third party, but does this digitally without any trusted party. One classical example is the millionaires' problem [Yao86] where two millionaires want to know who is richer, without any of them revealing to the other his net worth.

More formally, SFE is a cryptographic protocol that allows two players, P_1 with private input $x = x_1, \ldots, x_{u_1}$ and P_2 with private input $y = y_1, \ldots, y_{u_2}$, to evaluate a function $f(x, y)$ on their private inputs. The SFE protocol ensures that both parties learn the result of the evaluation but nothing about the other party's private input. In SFE, the function f is known to both parties. It suffices to consider SFE protocols with deterministic same-output functionalities f only, as probabilistic or arbitrary functionalities (where both parties get different outputs) can easily be reduced to this general case [LP04].

There are many practical protocols that use a SFE protocol as basic building block. Examples include Privacy Preserving Auctions [NPS99], Private Contract Negotiation [FA05], Privacy-Preserving Credit Checking [FAZ05], Privacy-Preserving Remote Diagnostics [BPSW07], Secure Surveys [FPRJ04], Mobile Code [ACCK01], Autonomous Agents [CCKM00], Untraceable Payment Systems [PW92], Secure computation of the k-th ranked element [AMP04], etc.

Most of them use Yao's SFE protocol (described later in Chapter 3.3.1), but in many cases each of the SFE protocols described in this chapter can be used instead.

Chapter 3.2 summarizes commonalities of all SFE protocols that are described thereafter: Circuit-based SFE protocols in Chapter 3.3 and OBDD-based SFE protocols in Chapter 3.4.

3.2 Commonalities of the Protocols

3.2.1 General Protocol Structure

The following one-round SFE protocols allow two parties, *Constructor* and *Evaluator*, to securely evaluate a function f on their private inputs. In the beginning, Constructor creates a garbled function $\tilde{f}$ (garbled circuit resp. garbled OBDD) corresponding to f. In $\tilde{f}$, the garbled values of the wires are two (randomly chosen) secrets that correspond to the values 0 or 1. If $\tilde{f}$ evaluates to garbled output values, output translation tables are provided to translate the garbled output to the corresponding binary values. Constructor sends $\tilde{f}$ together with the input secrets that correspond to his private input to Evaluator. Evaluator receives the input secrets corresponding to his inputs via (a parallel run of) 1-out-of-2 OT protocol. This ensures that Constructor doesn't learn anything about Evaluator's inputs. After evaluating $\tilde{f}$ using the input secrets, Evaluator sends the result back to Constructor. The following constructions differ mainly in the representation of the function (circuit in Chapter 3.3 vs. OBDD in Chapter 3.4) and the methods for garbling and evaluating it.

3.2.2 Provable Security in the Semi-honest Model

Each of the following protocols has a proof of security in the semi-honest model (see Chapter 2.6 for the different adversary models). To prove security of protocol π which evaluates a deterministic function f against semi-honest adversaries, two simulators S_i are constructed that simulate the view $view_i^\pi$ of party P_i during execution of π (i.e. all inputs and randomness P_i provides as input to π together with the messages that it receives during the run of π). Simulator Sim_i has as input everything that party P_i knows (its secret input and the output of f) and simulates all messages that are sent from the other party. The protocol π is secure against semi-honest adversaries, if the output of the simulators is computationally indistinguishable from the real view of the parties in the protocol:

$$\{S_1(x, f(x, y))\}_{x,y\in\{0,1\}^*} \stackrel{c}{\equiv} \{view_1^\pi(x, y)\}_{x,y\in\{0,1\}^*} \tag{3.1}$$

$$\{S_2(y, f(x, y))\}_{x,y\in\{0,1\}^*} \stackrel{c}{\equiv} \{view_2^\pi(x, y)\}_{x,y\in\{0,1\}^*} \tag{3.2}$$

The simulator for Constructor is simple. It invokes the OT-simulator to simulate the view in the OT protocol and outputs $f(x, y)$. The simulator for Evaluator also invokes the OT-simulator and constructs a fake-garbled function that always evaluates to $f(x, y)$ and is computationally indistinguishable from the garbled representation of f. This fake garbled function is sent to Evaluator instead.

3.2.3 Security in the Malicious Model

If a SFE protocol is provably secure in the semi-honest model it can be extended to be also secure in the malicious model. The basic idea is to force both parties to follow the protocol (deviations from the protocol are caught with high probability).

A simple and efficient method to catch a cheating Constructor (used in [MNPS04]) is cut-and-choose where Constructor creates m garbled functions. Evaluator can choose $m-1$ of these and obtains all secrets to verify that the garbled functions were created correctly. The remaining garbled function is evaluated. This catches a cheating Constructor with probability $1-\frac{1}{m}$. In order to prevent Evaluator from responding with the wrong result of the function output he has to reply with the last key he used to decrypt the result from the output table. This is demonstrated by extending the OBDD-based SFE protocol of [KJGB06] (secure against semi-honest adversaries) to be secure against malicious adversaries in Chapter 3.4.

In order to reduce Constructor's cheating probability to be exponentially small in m, more expensive techniques must be used.

One possibility is the compiler of Goldreich, Micali and Wigderson (known as *GMW compiler*) [GMW87]. "This compiler converts any protocol that is secure for semi-honest adversaries into one that is secure for malicious adversaries, and as such is a powerful tool for demonstrating feasibility. However, it is based on reducing the statement that needs to be proved (in our case, the honesty of the parties behavior) to an NP-complete problem, and using generic zero-knowledge proofs to prove this statement. The resulting secure protocol therefore runs in polynomial time but is rather inefficient." [LP07]

Another method for more efficient SFE protocols that are provably secure in the malicious model is to extend the cut-and-choose step which is a basic building block of [LP07]. Their protocol opens $m/2$ of the garbled circuits in the cut-and-choose step, evaluates the remaining $m/2$ garbled circuits and takes a majority of the result. Additionally they ensure that Constructor provides the same input to all the evaluated circuits using perfectly-hiding commitments. Their approach can also be used to get exponential security against malicious Constructor in the OBDD-based SFE protocol of Chapter 3.4.

3.3 Circuit-based SFE

The following circuit-based protocols allow two parties, Constructor and Evaluator, to securely evaluate a function f which is represented as a corresponding boolean circuit C (see Chapter 2.2.1). For simplicity of demonstration circuits C that contain gates with two inputs are considered only. Generalizations for circuits that consist of gates with an arbitrary number of inputs are described in the original publications.

3.3.1 Yao's Protocol

Yao's protocol [Yao86] is the first and most famous one-round two-party SFE protocol. Lindell and Pinkas give a detailled description and a proof of security in the semi-honest model of Yao's protocol in [LP04]. Recently, Lindell and Pinkas published a version of Yao's protocol which is secure against malicious adversaries [LP07].

Protocol 3.1 shows Yao's two-party SFE protocol that is explained in the following.

Protocol 3.1 Yao's two-party SFE protocol

- **Inputs**:
  P_1 (Constructor) has private input $x = \langle x_1, .., x_{u_1} \rangle \in \{0,1\}^{u_1}$ and
  P_2 (Evaluator) has private input $y = \langle y_1, .., y_{u_2} \rangle \in \{0,1\}^{u_2}$.

- **Auxiliary input**: A boolean acyclic circuit C such that $\forall x \in \{0,1\}^{u_1}, y \in \{0,1\}^{u_2}$, it holds that $C(x,y) = f(x,y)$, where $f : \{0,1\}^{u_1} \times \{0,1\}^{u_2} \rightarrow \{0,1\}^{v}$. C is required to be such that if a circuit-output wire leaves some gate G, then gate G has no other wires leading from it into other gates (i.e. no circuit-output wire is also a gate-input wire). Likewise, a circuit-input wire that is also a circuit-output wire enters no gates.

- **The protocol**:

  1. P_1 constructs the garbled circuit and sends it (i.e. the garbled tables and output decryption tables) to P_2.
  2. Let $W_1, .., W_{u_1}$ be the circuit input wires corresponding to x, and let $W_{u_1+1}, .., W_{u_1+u_2}$ be the circuit input wires corresponding to y. Then,
     a) P_1 sends P_2 the garbled values $w_1^{x_1}, .., w_{u_1}^{x_{u_1}}$.
     b) For every $i \in \{1, .., u_2\}$, P_1 and P_2 execute a 1-out-of-2 oblivious transfer protocol, where P_1's input is $(k_{u_1+i}^0, k_{u_1+i}^1)$, and P_2's input is y_i.
     All u_2 OT instances can be run in parallel.
  3. P_2 now has the garbled circuit and the garblings of circuit's input wires.
     P_2 evaluates the garbled circuit using the input garblings and outputs $f(x,y)$.
  4. P_2 sends $f(x,y)$ to P_1.

In Step 1 of Yao's protocol, P_1 first creates a *garbled circuit (GC)* from the given boolean circuit C: for each wire W_i of C, he randomly chooses two N-bit secrets, w_i^0 and w_i^1, where w_i^j is a *garbled value*, or *garbling*, of the W_i's value j. (Note: w_i^j does not reveal j as it is chosen randomly.)

Further, for each gate G_i, P_1 creates a *garbled table* T_i, with the following property: given a set of garblings of G_i's inputs, T_i allows to recover the garbling of the corresponding G_i's output, and nothing else. For simplicity of presentation only the case of a 2-input gate is presented in the following which can easily be generalized to n-input gates. For gate G_i with $W_c = g_i(W_a, W_b)$ the garbled table T_i contains a random permutation of the following entries:
$E_{w_a^0}(E_{w_b^0}(w_c^{g_i(0,0)}))$
$E_{w_a^0}(E_{w_b^1}(w_c^{g_i(0,1)}))$
$E_{w_a^1}(E_{w_b^0}(w_c^{g_i(1,0)}))$
$E_{w_a^1}(E_{w_b^1}(w_c^{g_i(1,1)}))$.
For each of the 2^2 possible input combinations, the input garblings are used to encrypt the corresponding output garblings using a semantically secure symmetric encryption scheme E that has the special properties described in Chapter 2.3.3. These special properties allow to determine which decryption succeeded during evaluation of GC. For each output wire W_o of C, P_1 creates an *output decryption table* that contains the pairs $\langle w_o^0, 0\rangle$ and $\langle w_o^1, 1\rangle$. The garbled circuit consists of the garbled tables and the output decryption tables.

In Step 2, P_1 sends the garbled circuit together with the garblings that correspond to his inputs to P_2. The garblings that correspond to P_2's inputs are sent via parallel executions of 1-out-of-2 OT protocol for each input bit.

Afterwards in Step 3, P_2 can evaluate the garbled circuit using the input garblings he obtained. He computes the garbled output values by evaluating the garbled circuit gate by gate in topologic order, using the garbled tables T_i. The special properties of E are used to decide which entry of each garbled function table was decrypted successfully. W_i's garbling w_i^j are called *active* if W_i assumes the value j when C is evaluated on the given input. Observe that for each wire, P_2 can obtain only its active garbling. For the output wires, P_2 uses the output decryption tables to determine the unencrypted result of the evaluation. P_2 learns (only) the output of the circuit, and no internal wire values.

Finally in Step 4, (semi-honest) P_2 sends the output to P_1. (This step is trivial in the semi-honest model, and is usually not considered in the analysis.)

Correctness of GC follows from method of construction of garbled tables T_i. Neither party learns any additional information from the protocol execution. The proof of correctness and security in the semi-honest model of Yao's protocol can be found in [LP04].

3.3.2 Fairplay

Fairplay is "a full-fledged system that implements generic secure function evaluation (SFE)" [MNPS04]. It is based on Yao's protocol and extends it to be secure against malicious adversaries in the random oracle model. For performance reasons, Fairplay uses a cryptographic hash function H (implemented as *SHA-1*) instead of the symmetric encryption scheme E used in Yao's protocol. To show the security of the protocol, SHA-1 is modeled as RO.

Constructor assigns to each wire W_i two random n-bit strings (keys) k_i^0 resp. k_i^1 that represent the values 0 resp. 1 (Fairplay uses $n = 80$ as security parameter). Also, a permutation bit $p_i \in_R \{0,1\}$ is randomly chosen for each wire. The garblings w_i^j consist of the key and the permutation bit: $w_i^0 = \langle k_i^0, p_i \rangle, w_i^1 = \langle k_i^1, \overline{p_i} \rangle$.
For each gate G_i with $W_c = g_i(W_a, W_b)$ that computes, an Encrypted-Garbled-Truth-Table (EGTT) is constructed: $EGTT[x,y] = H(k_a^x||i||x'||y') \oplus H(k_b^y||i||x'||y') \oplus w_c^{g_i(x,y)}$ with $x' = x \oplus p_a, y' = y \oplus p_b$. The table entries of the EGTT are permuted based on the permutation bits p_a and p_b to obtain the permuted EGTT (PEGTT) that corresponds to the garbled table T_i in Yao's protocol described before.
For each of Evaluator's output wires W_i, an output decryption table is created that contains $\langle H(k_i^0), 0 \rangle$ and $\langle H(k_i^1), 1 \rangle$, where H is a collision resistant hash function, which is implemented as *SHA-1*. The garbled circuit $\tilde{C}$ consists of all PEGTTs and all output decryption tables.

The garbled circuit and the secrets corresponding to Constructor's inputs are sent to Evaluator. Security against malicious Constructor is provided by a cut-and-choose technique where Evaluator obtains m garbled circuits and asks Constructor to open the secrets for $m-1$ of them and evaluate the remaining one. Evaluator obtains the secrets corresponding to his inputs via 1-out-of-2 OT.

Now, Evaluator evaluates the garbled circuit gate by gate in topologic order. For each garbled gate G_i with $W_c = g_i(W_a, W_b)$ and input garblings $w_a = \langle k_a||x \rangle, w_b = \langle k_b||y \rangle$ he decrypts the garbled output value from the corresponding PEGTT entry: $w_c = H(k_a||i||x||y) \oplus H(k_b||i||x||y) \oplus PEGTT[x,y]$.
For each of his own circuit output values he uses the corresponding output decryption table to obtain the unencrypted output values. The garbled output values of Constructor are sent back to him who can (secured against malicious Evaluator) interpret them to obtain his unencrypted output values.

3.3.3 Gate Evaluation Secret Sharing (GESS)

Gate Evaluation Secret Sharing (GESS) was introduced by Vladimir Kolesnikov in [Kol05]. GESS is a secret sharing scheme designed for use in SFE. GESS provides a

non-cryptographic reduction from SFE to OT in the semi-honest model. This allows SFE with information-theoretic (IT) security when instantiated with an IT-secure OT protocol.

In GESS, the two garbled values s_i^0 and s_i^1 corresponding to the values 0 and 1 of wire W_i are called shares. Fig. 3.1 shows a GESS gate with two inputs. A GESS scheme consists of a pair of algorithms (Shr, Rec). Shr is a randomized algorithm that generates the input shares of a gate when given the output shares (executed by Constructor). Rec is a deterministic algorithm that combines the input shares of a gate to the output share (executed by Evaluator).

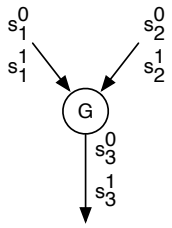


Figure 3.1: GESS gate

In the following two fundamental constructions of GESS are given using Fig. 3.1 as example. The paper contains generalizations and further optimizations (e.g. for AND/OR gates).

2-input XOR GESS gate G: $g(x_1, x_2) = x_1 \oplus x_2$
$Shr(s_3^0, s_3^1)$ with $s_3^0, s_3^1 \in \{0,1\}^n$: choose $R \in_R \{0,1,\}^n$, set $s_1^0 = R, s_1^1 = s_3^0 \oplus s_3^1 \oplus R, s_2^0 = s_3^0 \oplus R, s_2^1 = s_3^1 \oplus R$.
$Rec(s_1, s_2)$ with $s_1, s_2 \in \{0,1\}^n$: output $s_1 \oplus s_2$.

2-input arbitrary GESS gate: $g(x_1, x_2)$ arbitrary
$Shr(s_3^0, s_3^1)$ with $s_3^0, s_3^1 \in \{0,1\}^n$: choose $b \in_R \{0,1\}, R_0, R_1 \in_R \{0,1\}^n$, set blocks $B_{ij} = g(i,j) \oplus R_i$. Set input shares as follows

	wire 1	wire 2, if $b=0$	wire 2, if $b=1$
wire value 0	$s_1^0 = bR_0$	$s_2^0 = B_{00}B_{10}$	$s_2^0 = B_{10}B_{00}$
wire value 1	$s_1^1 = \bar{b}R_1$	$s_2^1 = B_{01}B_{11}$	$s_2^1 = B_{11}B_{01}$

$Rec(s_1, s_2)$ with $s_1 = b'r, s_2 = a_0a_1$, $b' \in \{0,1\}, r, a_0, a_1 \in \{0,1\}^n$: output $r \oplus a_{b'}$.

For SFE with GESS, Constructor creates the input shares for the circuit by calling Shr for each gate of the circuit (bottom up). Afterwards he sends the shares corresponding to the inputs to Evaluator (using OT for those corresponding to Evaluator's inputs). Evaluator combines the input shares gate-by-gate (bottom-down) using Rec and finally obtains the result. This result is sent back to Constructor.

The author of the paper notes, that the "GESS approach is especially efficient on small circuits, since it does not use encryption" and "that the efficiency of Yaos garbled

circuit technique in the standard model can be (slightly) improved by using IT GESS on 'the bottom part' of the circuit". GESS can also be used to improve "the bottom part" of the improved SFE protocol presented in the next section.

3.3.4 Improved SFE

This section presents our new improved SFE protocol which is a combination of Fairplay (Chapter 3.3.2) and the efficient information-theoretic SFE implementation of XOR-gates in GESS (Chapter 3.3.3). In Fairplay, XOR gates cost as much as other gates (i.e. in computation and communication required for creation, transfer and evaluation of the garbled tables). The XOR gates of GESS are free of these costs. However, his construction imposes a restrictive global relationship on the wire secrets, which prevents its use in previous GC schemes. In this section, we show how to overcome this restriction.

First, we show an SFE implementation of the XOR gate G, derived from GESS. Let G have two input wires W_a, W_b and output wire W_c. Garble the wire values as follows. Randomly choose $w_a^0, w_b^0, R \in_R \{0,1\}^N$. Set $w_c^0 = w_a^0 \oplus w_b^0$, and $\forall i \in \{a,b,c\} : w_i^1 = w_i^0 \oplus R$. It is easy to see that the garbled gate output is simply obtained by XORing garbled gate inputs (see Table 3.1 for explicit calculation). Further, garblings w_i^j do not reveal the wire values they correspond to.

Table 3.1: Correctness of garbled XOR-gate $W_c = XOR(W_a, W_b)$.

w_a	w_b	$w_c = w_a \oplus w_b$
w_a^0	w_b^0	$w_c^0 := w_a^0 \oplus w_b^0$
w_a^0	w_b^1	$w_c^1 := w_c^0 \oplus R = w_a^0 \oplus (w_b^0 \oplus R) = w_a^0 \oplus w_b^1$
w_a^1	w_b^0	$w_c^1 := w_c^0 \oplus R = (w_a^0 \oplus R) \oplus w_b^0 = w_a^1 \oplus w_b^0$
w_a^1	w_b^1	$w_c^0 := w_a^0 \oplus w_b^0 = (w_a^0 \oplus R) \oplus (w_b^0 \oplus R) = w_a^1 \oplus w_b^1$

We can now pinpoint the restriction that the above XOR construction imposes on the garbled values – the garblings of the two values of each wire in the circuit must differ by the same value, i.e. $\forall i : w_i^1 = w_i^0 \oplus R$, for some global R. In contrast, in previous GC constructions, *all* garblings w_i^j were chosen independently at random, and proofs of security relied on that property.

Our main observation is that it is not necessary to select all garblings independently. In our construction (Chapter 3.3.4), we choose a random R once, and garble wire values, so that $\forall i : w_i^1 = w_i^0 \oplus R$.

Improved Garbled Circuit Construction

Let C be a circuit. We first note that NOT gates can be implemented "for free" by simply eliminating them and inverting the correspondence of the wires' values and garblings. We thus do not further consider NOT gates.

We implement XOR gates as discussed above. Further, we replace each XOR-gate with $n > 2$ inputs with $n - 1$ two-input XOR-gates.

We implement all other gates using standard garbled tables [MNPS04]. Namely, each gate with n inputs is assigned a table with 2^n randomly permuted entries. Each entry is an encrypted garbling of the output wire, and garblings of the input wires serve as keys to decrypt the "right" output value. For simplicity, we present our construction and proof for the case $n = 2$. The generalization to n-input gates ($n \geq 1$) is straightforward.

In Algorithm 3.2 below, each garbling $w = \langle k, p \rangle$ consists of a key $k \in \{0,1\}^N$ and a permutation bit $p \in \{0,1\}$. The key is used for decryption of the table entries, and p is used to select the entry for decryption. The two garblings w_i^0, w_i^1 of each wire W_i are related as required by the XOR construction: for a chosen $R \in_R \{0,1\}^N$, $\forall i : w_i^1 = \langle k_i^1, p_i^1 \rangle = \langle k_i^0 \oplus R, p_i^0 \oplus 1 \rangle$, where $w_i^0 = \langle k_i^0, p_i^0 \rangle$. $H : \{0,1\}^* \mapsto \{0,1\}^{N+1}$ is a RO.

We now formalize the above intuition and present the GC construction (Algorithm 3.2) and evaluation (Algorithm 3.3). In SFE, Algorithm 3.2 is run by P_1 and Algorithm 3.3 is run by P_2.

Note, our encryption of table entries (Step 3(c)iii) is similar to that of Fairplay [MNPS04, Section 4.2]. Fairplay uses $e_{v_a,v_b} = H(k_a^{v_a}||i||p_a^{v_a}||p_b^{v_b}) \oplus H(k_b^{v_b}||i||p_a^{v_a}||p_b^{v_b}) \oplus w_c^{g_i(v_a,v_b)}$. This is a non-essential difference; we could use Fairplay's encryption.

Intuition for security. (A formal proof is given in Chapter 3.3.4.) Algorithm 3.2 uses the output of the RO H as a one-time pad to encrypt the garbled output values in the garbled tables (Step 3(c)iii) and the garbled output tables (Step 4a). Note, any specific combination of H's inputs (keys and gate indices) is used for encryption of at most one table entry throughout our construction. (We assume that concatenation and string representation inside H is done "right".) Further, since the evaluator of the garbled circuit only knows one garbled value per wire, he can decrypt exactly one entry of G_i's garbled table. All other entries are encrypted with at least one key that cannot be guessed by a polytime evaluator. Therefore, one of the two of garbled values of every wire looks random to him.

We now give the corresponding GC evaluation algorithm, run by P_2. Recall, P_2 obtains all garbled tables and the garblings of P_1's input values from P_1. Garblings of input values held by P_2 are sent via OT.

The GC construction and evaluation algorithms can be directly used to obtain the GC-based SFE protocol, in the standard manner described in Chapter 3.2.1. Protocol 3.1 shows the complete two-party SFE protocol.

Proof of correctness

Claim. Let $x = \langle x_1, .., x_{u_1} \rangle$ and $y = \langle y_1, .., y_{u_2} \rangle$ be the inputs of C on input wires $W_1, .., W_{u_1}$ and $W_{u_1+1}, .., W_{u_1+u_2}$. Then, given the garbled tables and the garbled input values $w_1^{x_1}, .., w_{u_1}^{x_{u_1}}$ and $w_{u_1+1}^{y_1}, .., w_{u_1+u_2}^{y_{u_2}}$, produced by Algorithm 3.2, Algorithm 3.3 out-

Algorithm 3.2 Construction of improved garbled circuit

1. Randomly choose global key offset $R \in_R \{0,1\}^N$
2. For each input wire W_i of C
   a) Randomly choose its garbled value $w_i^0 = \langle k_i^0, p_i^0 \rangle \in_R \{0,1\}^{N+1}$
   b) Set the other garbled output value $w_i^1 = \langle k_i^1, p_i^1 \rangle = \langle k_i^0 \oplus R, p_i^0 \oplus 1 \rangle$
3. For each gate G_i of C in topological order
   a) label $G(i)$ with its index: $label(G_i) = i$
   b) If G_i is an XOR-gate $W_c = XOR(W_a, W_b)$ with garbled input values $w_a^0 = \langle k_a^0, p_a^0 \rangle, w_b^0 = \langle k_b^0, p_b^0 \rangle, w_a^1 = \langle k_a^1, p_a^1 \rangle, w_b^1 = \langle k_b^1, p_b^1 \rangle$:
      i. Set garbled output value $w_c^0 = \langle k_a^0 \oplus k_b^0, p_a \oplus p_b \rangle$
      ii. Set garbled output value $w_c^1 = \langle k_a^0 \oplus k_b^0 \oplus R, p_a \oplus p_b \oplus 1 \rangle$
   c) If G_i is a 2-input gate $W_c = g_i(W_a, W_b)$ with garbled input values $w_a^0 = \langle k_a^0, p_a^0 \rangle, w_b^0 = \langle k_b^0, p_b^0 \rangle, w_a^1 = \langle k_a^1, p_a^1 \rangle, w_b^1 = \langle k_b^1, p_b^1 \rangle$:
      i. Randomly choose garbled output value $w_c^0 = \langle k_c^0, p_c^0 \rangle \in_R \{0,1\}^{N+1}$
      ii. Set garbled output value $w_c^1 = \langle k_c^1, p_c^1 \rangle = \langle k_c^0 \oplus R, p_c^0 \oplus 1 \rangle$
      iii. Create G_i's garbled table. For each of 2^2 possible combinations of G_i's input values $v_a, v_b \in \{0,1\}$, set

$$e_{v_a,v_b} = H(k_a^{v_a} || k_b^{v_b} || i) \oplus w_c^{g_i(v_a,v_b)}$$

      Sort entries e in the table by the input pointers, i.e. place entry e_{v_a,v_b} in position $\langle p_a^{v_a}, p_b^{v_b} \rangle$
4. For each circuit-output wire W_i (the output of gate G_j) with garblings $w_i^0 = \langle k_i^0, p_i^0 \rangle, w_i^1 = \langle k_i^1, p_i^1 \rangle$:
   a) Create garbled output table for both possible wire values $v \in \{0,1\}$. Set

$$e_v = H(k_i^v || \text{“}out\text{”} || j) \oplus v$$

   Sort entries e in the table by the input pointers, i.e. place entry e_v in position p_i^v. (There is no conflict, since $p_i^1 = p_i^0 \oplus 1$.)

puts $C(x, y)$.

Algorithm 3.3 Evaluation of improved garbled circuit

1. For each input wire W_i of C
   a) Receive corresponding garbled value $w_i = \langle k_i, p_i \rangle$
2. For each gate G_i (in the topological order given by labels)
   a) If G_i is an XOR-gate $W_c = XOR(W_a, W_b)$ with garbled input values $w_a = \langle k_a, p_a \rangle, w_b = \langle k_b, p_b \rangle$
      i. Compute garbled output value $w_c = \langle k_c, p_c \rangle = \langle k_a \oplus k_b, p_a \oplus p_b \rangle$
   b) If G_i is a 2-input gate $W_c = g_i(W_a, W_b)$ with garbled input values $w_a = \langle k_a, p_a \rangle, w_b = \langle k_b, p_b \rangle$
      i. Decrypt garbled output value from garbled table entry e in position $\langle p_a, p_b \rangle$: $w_c = \langle k_c, p_c \rangle = H(k_a || k_b || i) \oplus e$
3. For each C's output wire W_i (output of gate G_j) with garbling $w_i = \langle k_i, p_i \rangle$
   a) Decrypt output value f_i from garbled output table entry e in row p_i : $f_i = H(k_i || \text{“}out\text{”} || j) \oplus e$

Proof. First we show that Step 2 of Algorithm 3.3 correctly computes all garbled output values by correctly evaluating each garbled gate. We show this by induction on the gates of the circuit in the order of the labels. A garbled gate $\tilde{G}_i$ is evaluated correctly if its garbled output value corresponds to the output value of the corresponding un-garbled gate G_i in C. Step 2a correctly computes a garbled XOR gate which follows directly from the construction of a garbled XOR gate in Algorithm 3.2, Step 3b and Table 3.1. Step 2b correctly computes a garbled 2-input gate $\tilde{G}_i$: This garbled gate was constructed by Algorithm 3.2, Step 3c from the corresponding un-garbled gate G_i with functionality g_i. Let w_a, w_b be the garbled input values of $\tilde{G}_i$ that correspond to the input values v_a, v_b of G_i. Thus, $\langle k_a, \tilde{p}_a \rangle = w_a = w_a^{v_a} = \langle k_a^{v_a}, \tilde{p}_a^{v_a} \rangle ; \langle k_b, \tilde{p}_b \rangle = w_b = w_b^{v_b} = \langle k_b^{v_b}, \tilde{p}_b^{v_b} \rangle$. The evaluation of $\tilde{G}_i$ decrypts the permuted entry e in row $\tilde{p}_a, \tilde{p}_b$. This is the correct row as it corresponds to the right values v_a, v_b in the construction: $\forall j \in \{a, b\} : \tilde{p}_j = v_j \oplus p_j = \tilde{p}_j^{v_j}$. If $v_j = 0$ then $\tilde{p}_j^0 = p_j$, otherwise $\tilde{p}_j^1 = 1 \oplus p_j$ in accordance to the construction. The decryption of entry e in this row computes the correct garbled output value of the gate $w_c = H(k_a || k_b || i) \oplus e = H(k_a || k_b || i) \oplus H(k_a^{v_a} || k_b^{v_b} || i) \oplus w_c^{g_i(v_a, v_b)} = w_c^{g_i(v_a, v_b)}$.

Now we show that Step 3 of Algorithm 3.3 correctly decrypts the correctly computed garbled output values. Each $\tilde{C}$'s garbled output wire $\tilde{W}_i$ (output of gate $\tilde{G}_j$) was constructed by Algorithm 3.2, Step 4 from a un-garbled output wire W_i of C. Let $w_i = \langle k_i, \tilde{p}_i \rangle$ be the correctly computed garbled value of output wire $\tilde{W}_i$ that corresponds to the output value v_i of W_i. Thus, $\langle k_i, \tilde{p}_i \rangle = w_i = w_i^{v_i} = \langle k_i^{v_i}, \tilde{p}_i^{v_i} \rangle$. Step 3a of Algo-

rithm 3.3 decrypts the correct permuted entry e in row $\tilde{p}_i$: $\tilde{p}_i = v_i \oplus p_i = \tilde{p}_i^{v_i}$. If $v_i = 0$ then $\tilde{p}_i^0 = p_i$, otherwise $\tilde{p}_i^1 = 1 \oplus p_i$ in accordance to the construction. The decryption of entry e in this row yields the correct output value: $f_i = H(k_i||\text{“}out\text{”}||j) \oplus e = H(k_i||\text{“}out\text{”}||j) \oplus H(k_i^{v_i}||\text{“}out\text{”}||j) \oplus v_i = v_i$. □

Proof of Security

Our protocol is secure against semi-honest adversaries, who are not allowed to deviate from the protocol. Analogously to Fairplay (Chapter 3.3.2), (w.h.p) malicious behavior of players can be prevented by using cut-and-choose, providing output decryption tables corresponding to Evaluator's outputs only and sending the garblings of Constructor's outputs back; we don't discuss malicious players further.

We prove security in the simulator paradigm. Intuitively, a protocol π is secure if whatever is seen by its party, can be computed only from that party's input and output. The view of a party P_i, $view_{P_i}^{\pi}(x, y)$, consists of the party's own input, randomness, and all messages that P_i receives in the execution of π. Thus, a protocol is secure, if there exist *simulators* S_1, S_2, such that $\{S_1(x, f(x, y)\} \stackrel{c}{\equiv} \{view_{P_1}^{\pi}(x, y)\}$ and $\{S_2(y, f(x, y)\} \stackrel{c}{\equiv} \{view_{P_2}^{\pi}(x, y)\}$.

Case 1 - P_1 is corrupted. P_1's view in Protocol 3.1 consists only of the view in the OT protocols in Step 2b. The following $S_1(x, f(x, y))$ simulates the view of P_1. Let S_1^{OT} be the simulator that is guaranteed to exist for P_1 in the secure 1-out-of-2 OT protocol. S_1 constructs a garbled circuit using Algorithm 3.2. Then S_1 feeds the constructed garblings of the input wires corresponding to y to S_1^{OT}, and obtains the simulated transcript of the OT, which he outputs. S additionally outputs x and the randomness used in construction of GC. It is not hard to see that the output of the simulator is indistinguishable from the view of P_1.

Case 2 - P_2 is corrupted. We construct a simulator S_2 that given input $(y, f(x, y))$ simulates the view of P_2. P_2 receives a garbled circuit (including garbled inputs), which S_2 must simulate. However, S_2 doesn't know P_1's input x. Thus, S_2 can not honestly generate the garbled circuit, since it doesn't know which of the input garblings corresponding to x to hand to P_2 in Step 2a of the protocol. Instead, S_2 generates a fake garbled circuit that always evaluates to $f(x, y)$, using a slightly modified Algorithm 3.2. The only modification, in Step 4a, appropriately forges the output tables:

4. For each circuit-output wire W_i (the output of gate G_j) with garblings $w_i^0 = \langle k_i^0, p_i^0 \rangle, w_i^1 = \langle k_i^1, p_i^1 \rangle$:

   a) Create **fake** garbled output table for both possible wire values $v \in \{0, 1\}$ **of the same encrypted output value**. Set

$$e_v = H(k_i^v||\text{“}out\text{”}||j) \oplus \mathbf{f_i}(\mathbf{x}, \mathbf{y})$$

Sort entries e in the table by the input pointers, i.e. place entry e_v in position p_i^v.

Let S_2^{OT} be an OT simulator for P_2. S_2 outputs y, and the fake garbled circuit (i.e. its tables). Further, for each input wire W_i held by P_2, S_2 runs and outputs $S_2^{OT}(y_i, w_i^{y_i})$. Finally, S_2 simulates the received garblings of the input wires W_j held by P_1 simply by outputting w_j^0 (fake garblings corresponding to $x = 0..0$).

Theorem. The output of S_2 is indistinguishable from the real view of P_2.

Proof (sketch). First, observe that S_2 feeds S_2^{OT} proper inputs (i.e. y and the corresponding honestly generated garblings). Thus, simulation of Step 2b of the protocol is indistinguishable from the real execution. The crux of the proof is in showing the indistinguishability of the fake and real circuits (which include the tables and the input garblings that P_2 sees). This is addressed next.

First, observe, pointers p_i^j are independent of the parties' inputs, and thus are easily simulated by S_2. For ease of presentation, we omit the details of pointer simulation from the proof.

We now show that no polytime procedure D can distinguish simulated and real garbled circuit transcripts with non-negligible probability. We proceed inductively, gate by gate in topological order, in proving this for each partial transcript τ_i, where τ_0 includes all active secrets on the input wires, and each τ_i additionally includes the garbled tables of first i gates.

Induction base. It is easy to see that the partial transcript τ_0 – active secrets on the input wires – is distributed identically in real and simulated cases. Indeed, these secrets are uniformly random in the domain. Moreover, clearly, no distinguisher D_0 can output with non-negligible probability the global key offset $\hat{R}$ used in the construction of the (either simulated or real) transcript.

For the induction step, suppose no polytime D_{i-1} can with non-negligible advantage distinguish the τ_{i-1} transcripts (i.e. those including the active secrets on the inputs and the first $i-1$ garbled tables). Moreover, assume that no polytime D_{i-1} can output the global key offset $\hat{R}$ with non-negligible probability when given τ_{i-1}. We show that these properties hold also when additionally given the i-th garbled table.

Recall, the i-th garbled table contains (a permutation of) entries:

$H(k_a||k_b||i) \oplus v_{00}$
$H(k_a||k_b \oplus \hat{R}||i) \oplus v_{01}$
$H(k_a \oplus \hat{R}||k_b||i) \oplus v_{10}$
$H(k_a \oplus \hat{R}||k_b \oplus \hat{R}||i) \oplus v_{11}$

where $v_{00}, .., v_{11} \in \{k_c, k_c \oplus \hat{R}\}$ are the output secrets that correspond to the four possible gate input combinations. (Garbled output tables have one input and consist of two entries. The corresponding claims hold for these cases as well, via a natural modification of the following argument addressing two-input gates.)

Without loss of generality, suppose the active gate input secrets are k_a and k_b. By the induction assumption, no polytime D_{i-1} can compute both k_a and $k_a \oplus \hat{R}$, and both k_b and $k_b \oplus \hat{R}$ (otherwise D_{i-1} can output $\hat{R}$). Thus, D_{i-1} can call functions $H(k_a||k_b \oplus \hat{R}||i), H(k_a \oplus \hat{R}||k_b||i)$, or $H(k_a \oplus \hat{R}||k_b \oplus \hat{R}||i)$ only with negligible probability. Further, because of the inclusion of the gate index i, these function calls have not been made in the construction of (real or simulated) τ_i. Therefore, due to RO properties, except with negligible probability, all the inactive entries in the i-th table are distributed identically to random strings, from the point of view of D_{i-1}, and thus do not provide help to D_{i-1} in computing $\hat{R}$. Therefore, polytime D_i cannot output $\hat{R}$ or call any of $H(k_a||k_b \oplus \hat{R}||i), H(k_a \oplus \hat{R}||k_b||i)$, or $H(k_a \oplus \hat{R}||k_b \oplus \hat{R}||i)$, except with negligible probability. Therefore, no polytime D_i can distinguish the real and simulated transcripts τ_i with non-negligible probability.

This completes the induction and the proof of the theorem. □

Application of Improved SFE

We now present several motivating examples – practical functions whose SFE benefits from improvements of our construction. Universal circuit (UC) constructions (Chapter 5) do not explicitly use many XOR gates. We show how to modify these circuits to mainly consists of XOR gates, achieving fourfold reduction of garbled circuit size. Further, we show how to reduce in half the size of garbled circuits of commonly used blocks, such as integer addition and equality test.

Universal Circuits (Chapter 5) and Permutation Networks (Chapter 5.3.3). As described later in Chapter 5, the size of a UC mainly comes from programmable switching networks (such as the permutation networks described in Chapter 5.3.3) connecting the simulated gates. In turn, these networks are constructed from two types of switching blocks shown in Fig. 3.2, as discussed in [Wak68, Val76, KS08]. The *Y-block* can be programmed to output one of its two inputs. The *X-block* can be programmed to either pass or cross over its two inputs to the two outputs. A natural SFE implementation of the Y-block uses a 2-input garbled gate with a garbled table with $2^2 = 4$ encrypted table entries. Similarly, X-block is implemented by two 2-input garbled gates (one for each of its two outputs), resulting in a garbled table of $2 \cdot 2^2 = 8$ entries.

(a) Y switching block (b) X switching block

Figure 3.2: Switching blocks

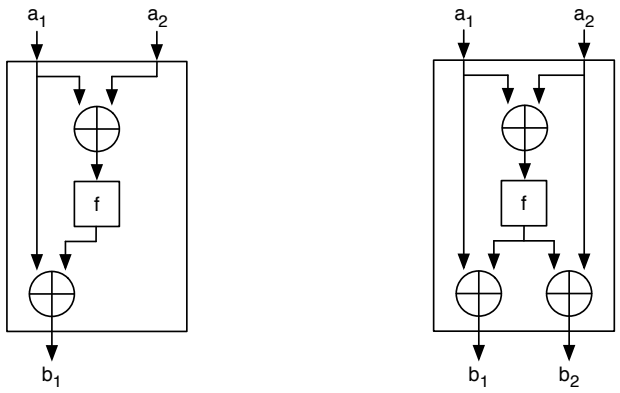
We show how to take advantage of free XOR gates and implement both X- and Y-blocks with only two garbled table entries each. Since permutation network [Wak68]

consists only of X-gates, this results in 75% size reduction of its SFE. UC consists *almost exclusively* of X-gates. Valiant's UC [Val76] for a circuit of k gates has size $\sim 19k \log k$. The $\sim 19k \log k - k$ overhead gates are X-gates that come from switching networks. Our new practical UC construction described in Chapter 5.3 similarly consists almost exclusively of X-blocks, and of very few Y-blocks and simulated gates. Thus, UC enjoys almost 75% garbled table size reduction.

Let $f : \{0,1\} \mapsto \{0,1\}$ be a function (implemented with two garbled table entries). We implement X- and Y-blocks as follows (see Fig. 3.3). $Y(a_1, a_2) = b_1 = f(a_1 \oplus a_2) \oplus a_1$; $X(a_1, a_2) = (b_1, b_2)$, where $b_1 = f(a_1 \oplus a_2) \oplus a_1, b_2 = f(a_1 \oplus a_2) \oplus a_2$. It is easy to see that setting $f = f_0$ to the zero function results in Y choosing left input, and X passing the inputs. Further, setting $f = f_{id}$ to the identity function results in Y choosing the right input, and in X crossing its inputs:

$f = f_0$: $\quad b_1 = 0 \oplus a_1 = a_1; \quad b_2 = 0 \oplus a_2 = a_2.$

$f = f_{id}$: $\quad b_1 = (a_1 \oplus a_2) \oplus a_1 = a_2; \quad b_2 = (a_1 \oplus a_2) \oplus a_2 = a_1.$



(a) Y switching block (b) X switching block

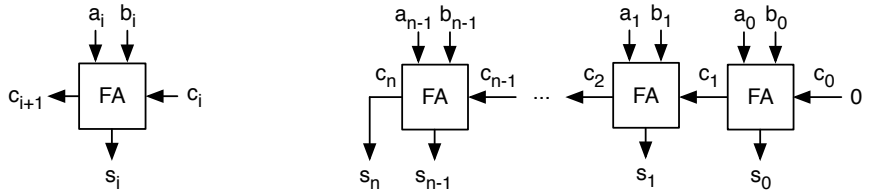
Figure 3.3: Efficient implementation of switching blocks

This construction can be extended to implement programmable switching blocks X and Y, which take an additional programming input bit p. This bit determines behavior of X- (pass or cross) and Y-blocks (left or right input). The natural construction for the Y- (resp. X-) switching block uses one (resp. two) 3-input gate(s) with $2^3 = 8$ (resp. 16) encrypted table entries. In our XOR-based construction, function f is then replaced by a two-input AND-gate (with p being the second input) with $2^2 = 4$ encrypted table entries. Clearly, $p = 0$ sets $f = f_0$, and $p = 1$ sets $f = f_{id}$, allowing to program X- and Y-blocks. As above, the size of Y- and X-blocks is reduced by 50% and 75% respectively.

Integer Adder and Multiplier. An adder for n-bit integers a, b is composed from a chain of n full adder (FA) blocks as shown in Fig. 3.4(b)[1]. A FA block (see Fig. 3.4(a)) has as inputs a carry-in c_i from the previous FA block and the two input bits

[1]The last FA block can be replaced by a smaller half-adder block.

a_i and b_i. It outputs two bits: carry-out $c_{i+1} = (a_i \wedge b_i) \vee (a_i \wedge c_i) \vee (b_i \wedge c_i)$ and sum $s_i = a_i \oplus b_i \oplus c_i$. The straightforward implementation of a FA uses two 3-input gates with $2 \cdot 2^3 = 16$ encrypted table entries. We can compute s_i "for free" using free XOR-gates, and use only one 3-input gate with $2^3 = 8$ encrypted table entries to compute c_{i+1}. The size of a FA block, and hence that of an n-bit adder is reduced by 50%.



(a) Full Adder (FA) (b) n-bit Adder built from n FA blocks

Figure 3.4: Adder for two n-bit integers a and b

As circuits for integer multiplication consist of bit-multipliers (2-input AND-gates) and adders, the improved implementation of adders can directly be used to correspondingly improve integer-multiplication circuits.

Integer Equality Test. A similar construction is used to test equality of two n-bit integers a and b. Now, we do not compute s_i, and use carry bits as inequality flags. The carry-out bit is defined as $c_{i+1} = (a_i \neq b_i) \vee c_i = (a_i \oplus b_i) \vee c_i$. A simple implementation uses two 2-input gates or one 3-input gate (each costs 8 encrypted table entries). Free XOR gate reduces the cost to that of one 2-input OR gate (4 encrypted table entries). The size of equality test block is thus reduced by 50%.

3.4 OBDD-based SFE

Kruger et al. [KJGB06] give a protocol for SFE based on the OBDD representation of a boolean function (see Chapter 2.2.2). They show that for many functions their OBDD-based SFE implementation outperforms the currently best circuit-based SFE implementation Fairplay (bitwise AND, integer addition, equality test, Millionaires Problem, parity checking). For functions with a large OBDD representation[2] (integer multiplication and keyed database lookup) Fairplay performs better.

Depending on the size of the OBDD, OBDD-based SFE is a good alternative to circuit-based SFE protocols and both methods can also be combined securely.

3.4.1 Improved OBDD-based SFE

In the following a scheme for two party SFE of functions represented as OBDDs is introduced which is secure in the malicious model. The scheme applies known techniques

[2]See the discussion about efficiency of OBDD representations for certain functions in Chapter 2.2.2.

to the protocol of [KJGB06] who present a provably secure OBDD-based SFE protocol in the semi-honest model. Their scheme uses a semantically secure encryption scheme E that has the special properties described in Chapter 2.3.3 to determine whether a decryption succeeded. Performance can be improved using a point-and-permute technique as shown in the end of this section. Protocol 3.4 shows how to construct and evaluate a garbled OBDD $\tilde{O}$ to securely evaluate a function represented as OBDD O.

Protocol 3.4 differs from Protocol 1 of [KJGB06] by the following minor changes to provide security in the malicious model:

Step 1: Creator garbles the whole OBDD and sends the secrets corresponding to its own inputs to Evaluator instead of traversing the OBDD with its own inputs and garbling the remaining part of the OBDD. This clearly results in an overhead as the garbled OBDD contains more encrypted nodes but allows cut-and choose to catch a malicious Evaluator as described in the following.

Step 2: Evaluator obtains the keys corresponding to his inputs via an OT protocol which is secure in the malicious model [LP07]. Afterwards he is able to evaluate the garbled OBDD. In the end, Evaluator sends the secret used to decrypt the terminal node to Constructor who can use this to get the function output, too and to catch a malicious Evaluator. Cut-and-choose provides security against malicious Creator similar to the approach used in Fairplay (Chapter 3.3.2): Creator constructs m garbled OBDDs and opens the secrets to all but one garbled OBDD which is randomly chosen by Evaluator. This allows to catch a cheating Creator with probability $1 - 1/m$.

This protocol and the original protocol can be improved to directly find the intended successor to decrypt instead of trying to decrypt both successors and decide which one is the right one using the special properties of the symmetric encryption scheme described in Chapter 2.3.3. This also allows usage of a semantically symmetric encryption scheme without the overhead to provide these special properties. The improvement uses the *Permute and Point (PP) Technique* of [Kol05] which is also used in Fairplay (Chapter 3.3.2): For all nodes v_i at level $level(v_i) = \ell$, the encrypted successor nodes in $E^{(v_i)}$ are either all permuted or not (random for each level). Based on this, a single bit is appended to the secrets s_ℓ^0 and s_ℓ^1 to indicate which successor node to decrypt during evaluation.

3.5 Summary

Table 3.2 shows a comparison of the SFE protocols explained in the last sections w.r.t. the underlying proof of security and if they use the Random Oracle Assumption (RO). The used encryption method to create/evaluate the garbled tables are (from fast to slow):

Protocol 3.4 Improved OBDD-based SFE

Input: Both parties inputs include the $OBDD(f)$ O for the Boolean function $f(x_1, \ldots, x_n)$ with the ordering $x_1 < \ldots < x_n$. Furthermore, Evaluator holds the inputs $(i_1, \ldots, i_k)$ corresponding to the first k variables $x_1, \ldots, x_k$, and Constructor has the inputs $(i_{k+1}, \ldots, i_n)$.

1. Constructor performs the following steps:

   a) First, Constructor augments O with some number of dummy nodes (to ensure that Evaluator always traverses n nodes in his phase of the protocol).

   b) Constructor uniformly and independently at random creates n pairs of secrets $(s_1^0, s_1^1), \ldots, (s_n^0, s_n^1)$. In addition, for each node v he creates a secret s_v.

   c) He assigns a uniformly random label, $label(v)$, to each node v.

   d) Constructor garbles all nodes whose level is between 0 and $n-1$ in the following manner. Let v be a node such $0 \leq level(v) \leq n-1$ and define $level(v) = \ell$. The encryption of v, denoted by $E^{(v)}$, is a label and a randomly ordered ciphertext pair

   $$(label(v), E_{s_v \oplus s_\ell^0}(label(low(v)) || s_{low(v)}), E_{s_v \oplus s_\ell^1}(label(high(v)) || s_{high(v)}))$$

   where the labels are pre-pended to the secret with a separator symbol and the order of the ciphertexts is determined by a fair coin flip. Roughly speaking, the secrets corresponding to the 0-successor and 1-successor of node v are encrypted with the secret corresponding to v and its level.
   Note that dummy nodes have the same structure as normal nodes, except that the ciphertext pair contain encryptions of the same message since dummy nodes have the same 0 and 1-successors. Provided the encryption scheme is semantically secure, this poses no problem since the keys are chosen uniformly at random.
   Lastly, there are two terminal nodes of the form $(label(t_b), E_{s_{t_b}}(b))$ for $b = 0$ or 1.

   e) Constructor sends to Evaluator the encryption of all nodes and the secret $s_{v_{root}}$ corresponding to the root, called the garbled OBDD $\tilde{O}$.

   f) Constructor sends the secrets corresponding to his input $s_1^{x_1}, \ldots, s_k^{x_k}$ to Evaluator.

2. Evaluator performs the following steps:

   a) He engages in $n-k$ 1-out-of-2 OTs to obtain the secrets corresponding to his input $s_{k+1}^{x_{k+1}}, \ldots, s_n^{x_n}$.

   b) Now Evaluator is ready to start his computation. Suppose $x_1 = 0$. With s_1^0 and $s_{v_{root}}$, he decrypts both ciphertexts in $E^{(v_{root})}$ and decides which gives the correct result by using the verifiable range property of the encryption scheme. Evaluator now has both $s_{low(v)}$ (the secret corresponding to the 0-successor of v_{root}) and $label(low(v))$ (which tells Evaluator which encrypted node is used to evaluate his next input). Continuing this way, Evaluator eventually obtains a label and a secret corresponding to one of the terminal nodes, which he decrypts to determine the result of the OBDD on the shared inputs.

   c) Evaluator sends the secret he used to decrypt the terminal node to Constructor who uses this to verify honesty of Evaluator and to obtain the result, too.

XOR of bitstrings (XOR), semantically secure symmetric encryption (E), cryptographic hash function as RO instantiation (H) and semantically secure symmetric encryption with the special properties described in Chapter 2.3.3 (E_{spec}).

Function Represent.	Protocol	Reference	Proof of Security			Encrypt
			semi-honest	malicious	RO	
Circuit	Yao	3.3.1	X			E_{spec}
	Fairplay	3.3.2	X	X	X	H
	GESS	3.3.3	X			XOR
	Improved SFE	3.3.4	X	X	X	H/XOR
OBDD	OBDD SFE	[KJGB06]	X			E_{spec}
	Impr. OBDD SFE	3.4	X	X		E

Table 3.2: Comparison of the described SFE protocols

For each function representation, the different protocols might also be combined to further improve performance. Improved SFE combined with GESS in the bottom part of the circuit is a promising combination for efficient and practical SFE in the semi-honest model.

4 Secure Evaluation of Private Functions (PF-SFE)

4.1 Introduction

In practice, not only the inputs, but also the function being evaluated needs to be protected. One example is checking a passenger against the no-fly list (or, more generally, no-fly function of passenger's data). Here, a compromise of the function weakens the security of the system significantly. Other examples include credit checking or background- and medical history checking functions.

This extension of SFE is called SFE of private function (PF-SFE) and addressed by a large amount of work like [FAZ05, CCKM00, SYY99, Pin02].

In PF-SFE, the evaluated function is known only by one party and needs to be kept secret (i.e. everything besides the size, the number of inputs and the number of outputs is hidden from the other party). Full or even partial revelation of these functions opens vulnerabilities in the corresponding process, exploitable by dishonest participants (e.g. credit applicants), and should be prevented.

Based on the representation of the private function there are different approaches for PF-SFE that are described in the following.

4.2 Universal Circuit-based PF-SFE

It is well known that the problem of PF-SFE can be reduced to the "regular" circuit-based SFE [SYY99, Pin02]. This is done by parties evaluating a *Universal Circuit* (UC) instead of a circuit defining the evaluated function. UC can be thought of as a "program execution circuit", capable of simulating any circuit C of certain size, given the description of C as input. Therefore, disclosing the UC does not reveal anything about C, except its size. At the same time, the SFE computes output correctly and C remains private, since the player holding C simply treats description of C as additional (private) input to SFE. This reduction is the most common (and often the most efficient) way of securely evaluating private functions [SYY99, Pin02]. The next chapter shows constructions for universal circuits.

4.2.1 Applications

As discussed above, UC is naturally used to extend the functionality or privacy in numerous practical SFE applications, in particular those based on Yao's garbled circuit [Yao82, Yao86, LP04].

To perform PF-SFE, instead of evaluating the circuit directly, a UC that is programmed with the original circuit is evaluated. As UC can be programmed with any circuit, the evaluated function is entirely hidden from the evaluator.

Next, some natural applications that can be extended by using universal circuits are discussed.

Frikken et. al [FAZ05] show a privacy-preserving credit checking scheme that is based on the evaluation of a garbled circuit. Their scheme is limited to the special class of credit-checking policies that can be expressed as the weighted sum of criteria. By evaluating a universal circuit their scheme can be extended to arbitrary, more complicated, private credit-checking policies.

Cachin et al. [CCKM00] describe autonomous mobile agents which migrate between several distrusting hosts. Garbled-circuit-based, their scheme ensures the privacy of the inputs of the visited hosts but not the structure of the mobile agent's code. The privacy of the executed code can be guaranteed by evaluating universal circuits instead.

Ostrovsky and Skeith [OS05] show how to filter remote streaming data (e.g airports' passenger lists, on-line news feeds or internet chat-rooms) using secret keywords and their combinations, such as no-fly lists. Their protocol allows Collector (e.g. airport) to obliviously filter out entries that match the (encrypted) query, which are then sent back for decryption. Their scheme can be naturally extended to allow a much finer private matching criteria, additionally preserving data privacy, as follows. The Collector encrypts each filtered stream element with a random pad. The querying party thus obtains the list of encrypted matches. In the second round, the querying party uses PF-SFE (e.g. using our UC_k) to search the matching data with an arbitrary, more detailed private search function.

4.3 OBDD-based PF-SFE

The protocol for SFE with OBDDs of Chapter 3.4 (resp. the original protocol of [KJGB06]) can directly be used as PF-SFE protocol for a private function represented as OBDD, which is secure semi-honest Evaluator (security against malicious players can be achieved by zero-knowledge proofs appropriately). In the protocol of Chapter 3.4 the cut-and-choose step must be left out, of course, as otherwise the function would be revealed to Evaluator. Hence, Constructor might also traverse the OBDD with his inputs and garble the remaining OBDD starting from node v_{init} again as in the original construction to reduce the size of the garbled OBDD and avoid the transfer of his input

secrets. Evaluator is able to decrypt the label and the secret for exactly the n (resp. $n-k$) nodes on the path through the OBDD for the inputs. Thus, he learns nothing about the structure of the OBDD. The only additional information (besides OBDD's size, number of inputs and outputs) Evaluator obtains is that he has to use the inputs in the given order $x_1 < \ldots < x_n$ (resp. $x_{k+1} < \ldots < x_n$) which results in a OBDD representation of the function that has the observed size. This very small leak of information can also be eliminated: Evaluator first securely evaluates a programmable permutation block P_n^n (resp. P_{n-k}^{n-k}) as described later in Chapter 5.3.3 with a circuit-based SFE-protocol. This permutation block is programmed by Constructor to permute the secrets corresponding to the n (resp. $n-k$) inputs from a pre-defined order to the order of the garbled OBDD and hence hides the order of the inputs entirely.

While the UC-based PF-SFE protocols described before increase the size of the evaluated circuit dramatically (evaluation of a much larger universal circuit), the increase between SFE and PF-SFE for functions represented as OBDDs is much smaller.

5 Universal Circuit Constructions

As described in Chapter 4.2, universal circuits can be used to reduce PF-SFE to SFE for functions represented as circuit. In this chapter different constructions for universal circuits are presented. Intuitively, a universal circuit (UC) is a circuit that can be programmed to simulate any other circuit.

First, basic notations and building blocks are defined in Chapter 5.1 before the actual UC constructions are presented and compared in the following sections.

5.1 Definitions and Preliminaries

In the following, a *gate* is the implementation of a boolean function $\{0,1\}^2 \rightarrow \{0,1\}$ that has two *inputs* and one *output*. Acyclic *circuits* are considered that consist of connected gates with arbitrary fan-out, i.e. the (single) output of each gate can be used as input to an arbitrary number of gates. Further, each output of the circuit C is the output of a gate and not a redirected input of C.

A *block* B_v^u is a circuit that has u inputs $in_1, .., in_u$ and v outputs $out_1, .., out_v$ (the variable u is always associated with inputs and v with outputs). B_v^u computes a function $f_B : \{0,1\}^u \rightarrow \{0,1\}^v$ that maps the input values to the output values. For simplicity, B_v^u is identified with f_B, written as: $B(in_1, \ldots, in_u) = (out_1, \ldots, out_v)$. The *size* of a block B, $size(B)$, is the number of gates B consists of; its depth, $depth(B)$, is the maximum number of gates between any input and any output of B. A block can be a sub-block of a larger block. A circuit is constructed as a collection of functional blocks, as this simplifies presentation.

A *programmable* block is a block that consists of connected programmable gates with unspecified function tables. *Programming* a programmable block is done by providing a specific function table for each of its gates.

A *Universal Circuit* $\mathrm{UC}_{u,v,k}$ is a programmable block with u inputs and v outputs that can be programmed to simulate any circuit C with up to u inputs, v outputs and k gates. UC_C denotes UC that is programmed to simulate circuit C, that is $\forall(in_1, \ldots, in_u) : \mathrm{UC}_C(in_1, \ldots, in_u) = C(in_1, \ldots, in_u)$.

A *one-output switching block* Y is a programmable block that computes $(in_1, in_2) \rightarrow in_1$ or in_2, as shown in Fig. 3.2(a). It is implemented by one gate programmed with the corresponding function table. $size(Y) = depth(Y) = 1$.

A *two-output switching block* X is a programmable block shown on Fig. 3.2(b) that computes $(in_1, in_2) \to (in_1, in_2)$ or (in_2, in_1). It is implemented by using (in parallel) two Y blocks: one for each of the outputs. $size(X) = 2; depth(X) = 1$.

A *selection block* S_v^u is a programmable block that selects for each of its v outputs one of the u input values (with duplicates). S_v^u is programmed according to the selection mapping $(\sigma_i)_{i=1}^v, \sigma_i \in \{1..u\}$ that selects the σ_i-th input as the i-th output. That is, a programmed S_v^u computes $S(in_1, \ldots, in_u) = (in_{\sigma_1}, \ldots, in_{\sigma_v})$.

A S_1^u selection block can be implemented by $(u-1)$ Y blocks that are programmed to switch the desired input value in_{σ_1} to the output. Shallow S_1^u is obtained by arranging Y blocks in a tree as shown in Fig. 5.1. Thus, $size(S_1^u) = u - 1, depth(S_1^u) = \log u$.

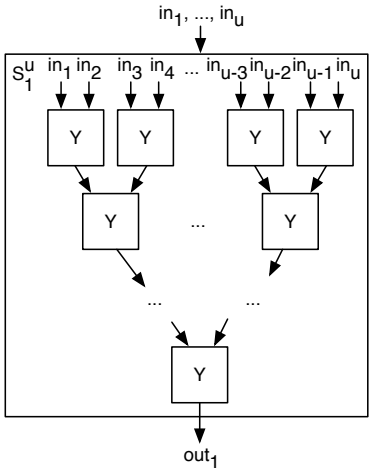


Figure 5.1: S_1^u selection block with minimal $depth(S_1^u) = \log u$

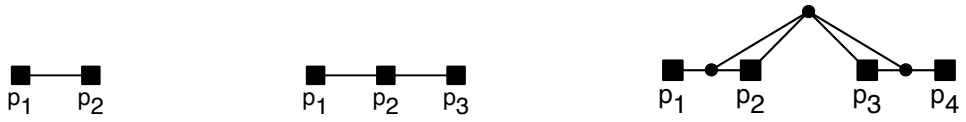
A naive implementation of S_v^u selection block uses a S_1^u selection block for each of the v outputs, resulting in $size(S_v^u) = v(u-1)$ and $depth(S_v^u) = \log u$. Selection blocks are crucial for our UC construction. Much more efficient S_v^u constructions are described in Chapter 5.3.4.

5.2 Valiant's UC Construction

Valiant [Val76] gave an asymptotically optimal universal circuit construction. Including all optimizations, he shows how to construct a UC of $size(\mathrm{UC}_{u,v,k}^{Valiant}) = (19k + 9.5u + 9.5v)\log k + O(k)$. In the following, Valiant's basic construction described in [Weg87, 4.2 Universal Circuits] is summarized while the optimizations to obtain the small pre-factors are omitted.

It suffices to construct a universal circuit that is able to simulate circuits with fan-out 2 only as each circuit C with arbitrary fan-out can easily converted into such one

by duplicating gates with fan-out > 2. Let $m = u + \tilde{k}$, where u is the number of inputs of the simulated circuit C and $\tilde{k}$ is the number of gates with fan-out 2 after conversion. Valiant's UC consists of m *distinguished nodes* where the first u of them are identified with the inputs of C. The remaining $\tilde{k}$ distinguished nodes are connected to gate simulation blocks that can be programmed to simulate the corresponding gates. UC now must ensure that the 2 outputs of each distinguished node can be connected to each successor distinguished node. This switching can be reduced to two universal graphs $U(m)$ of size $O(m \log m)$, fan-out and fan-in 2 for all nodes, fan-out and fan-in 1 for m distinguished nodes, such that all directed acyclic graphs of size m and fan-out and fan-in 1 can be simulated.



(a) $U(2)$ (b) $U(3)$ (c) $U(4)$

Figure 5.2: Universal Graphs $U(m)$ - recursion base

Fig. 5.2 shows $U(m)$ for some constant m. The direction of the edges is from left to right and omitted. Distinguished nodes p_i (■) correspond to the u inputs of C resp. $\tilde{k}$ gate simulation blocks while switching nodes (•) are replaced with X switching blocks in the UC construction.

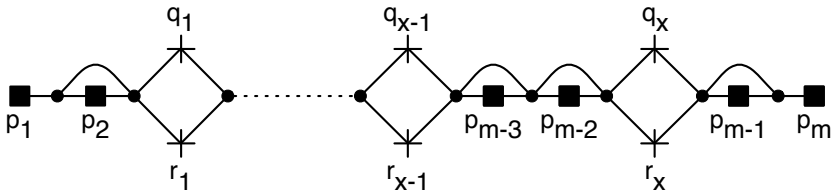


Figure 5.3: Universal Graph $U(m)$ - recursive construction

For larger $m = 2x+2$, $U(m)$ can be recursively constructed from two $U(x)$, $U_q(x)$ and $U_r(x)$, as shown in Fig. 5.3. Marked nodes (+) $q_1, \ldots, q_x$ resp. $r_1, \ldots, r_x$ are replaced with the distinguished nodes (■) of the two smaller universal graphs $U_q(x)$ resp. $U_r(x)$.

To simulate a given circuit C with Valiant's UC, C is first converted into a circuit with $\tilde{k} \geq k$ gates of fan-out 2. Afterwards, two universal graphs $U(m)$ where $m = \tilde{k} + u$ are constructed where the distinguished nodes (■) are connected to the u inputs and the $\tilde{k}$ gate simulation blocks. The switching nodes (•) are replaced by X switching blocks and can be programmed corresponding to the structure of the wires of C by finding an embedding of C into the two universal graphs $U(m)$.

In the next section our new practical UC construction is presented which is not asymptotically optimal but smaller for circuits used in practical PF-SFE. Further, we believe that implementation and programming of our practical UC construction is more self-contained and straightforward as it does not rely on graphs and graph embedding but only on the structure of the simulated circuit itself.

5.3 A Practical UC Construction

In this section, we present our modular UC construction. All of the necessary building blocks were introduced in Chapter 5.1; in this Section we show how to assemble them to a UC. Then, starting from Chapter 5.3.3, we design improved versions of some building blocks, which results in performance improvement of our UC.

In our UC construction, we simulate each gate G_i of the original circuit C. That is, for each G_i, $\mathrm{UC}_{u,v,k}$ has a corresponding programmable G_i-simulation gate G_i^{Sim}. In our construction, we always ensure that inputs, outputs and semantics of G_i^{Sim} correspond to G_i. Additionally, we hide the wiring of C by ensuring that every possible wiring can be implemented in $\mathrm{UC}_{u,v,k}$. This is the natural method of construction of UC, and is, in fact, employed by Valiant [Val76].

We design our UC construction recursively (we build a circuit from two circuits of smaller size). We first note that the input/output interface of $\mathrm{UC}_{u,v,k}$ is different from that of the natural recursion step. This is why we introduce a *universal block* U_k. U_k can be viewed as a UC with specific input and output semantics. Namely, U_k has $2k$ inputs and k outputs, since this is a maximum $\mathrm{UC}_{u,v,k}$ can have. Further, we restrict that U_k's inputs in_{2i-1}, in_{2i} are only delivered to the simulation gate G_i^{Sim}, and U_k's i-th output comes from G_i^{Sim}. (Of course, input of some gates G_i may come from any other gates' outputs, and not from in_{2i-1} or in_{2i}, which may not be used at all. U_k allows this; it only restricts that G_i's input cannot come from other in_j). U_k is thus a UC for the class of circuits of size k with the above input/output restrictions.

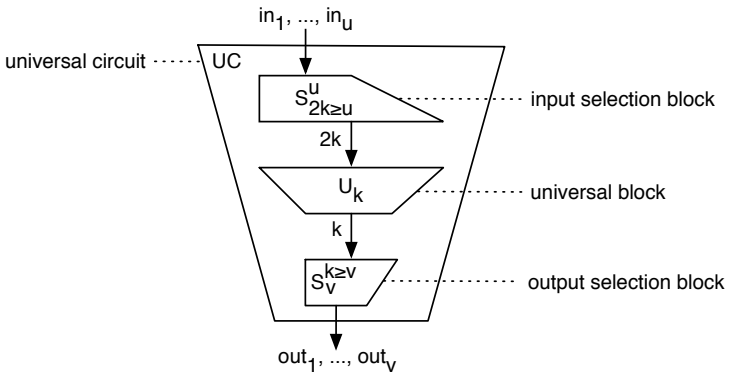


Figure 5.4: Modular universal circuit construction

Now, given an implementation of U_k, it is easy to construct $\mathrm{UC}_{u,v,k}$ (shown on Fig. 5.4). We need to provide the input selection block, which directs inputs of UC to the proper inputs of U_k. Finally, we need the output selection block, directing outputs of U_k to the proper outputs of UC, and discarding unused outputs. Both blocks are instances of selection blocks discussed above.

In the following, we refer to the gates of the circuit C_k by their index. We choose a topologic order of the gates $G_1, \ldots, G_k$, which ensures that the i-th gate G_i has no inputs that are outputs of a successive gate G_j, where $j > i$. Since we only consider acyclic circuits, we can always obtain this ordering by topological sorting with complexity $O(k)$.

Now we present two U_k constructions. Plugged in the construction of Fig. 5.4, they both give a complete UC construction.

5.3.1 Simple Universal Block Construction

A straight-forward implementation of a universal block U_k can be constructed as shown in Fig. 5.5: For the first (second) input of every gate simulation block G_i^{Sim}a selection block S_1^i selects the input for $G_i^{Sim}$from the corresponding input to the universal block in_{2i-1} (in_{2i}) and the outputs of the previous ($i-1$) gate simulation blocks $G_1^{Sim}, \ldots, G_{i-1}^{Sim}$. This results in $size(U_k^{simple}) = k^2$ which is not practical if k grows.

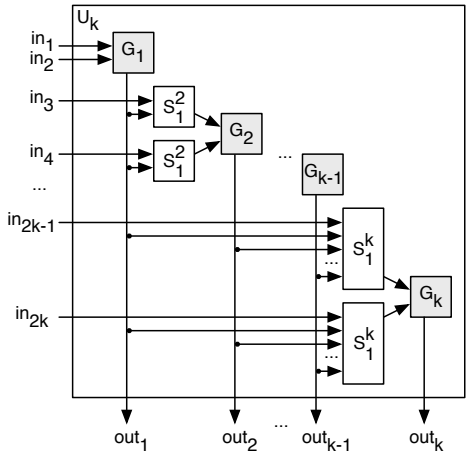


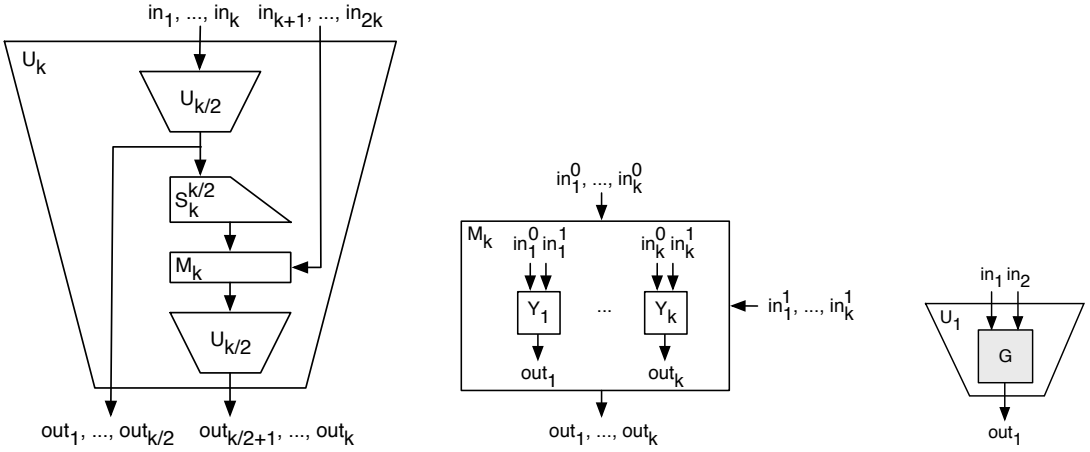
Figure 5.5: Simple universal block construction

Next we will present a more efficient practical construction for the universal block U_k.

5.3.2 Recursive Universal Block Construction

In this section, we describe a natural divide-and-conquer procedure for constructing U_k, capable of simulating any circuit C_k of size k, with the input/output restrictions mentioned above.

Suppose we have two universal blocks $\mathrm{U}_{k/2}$, universal for circuits $C_{k/2}$ of size $k/2$. We would like to combine them to obtain U_k. Clearly, because of their universality, one of $\mathrm{U}_{k/2}$ could simulate the "upper" half of C_k (i.e. gates G_1 through $G_{k/2}$) , and the other $\mathrm{U}_{k/2}$ could simulate the lower half (gates $G_{k/2+1}, \ldots, G_k$). Note, by the topological ordering, there is no data going into the upper $\mathrm{U}_{k/2}$ from the lower one. Thus, U_k must only direct its inputs/outputs and allow implementation of all possible data paths from the upper $\mathrm{U}_{k/2}$ to the lower one. This can be naturally done, as shown on Fig. 5.6(a). We describe this in detail below.



(a) Recursive construction of U_k (b) Mixing block M_k (c) U_1

Figure 5.6: Recursive universal block construction

The first k inputs to U_k $in_1, .., in_k$ are directly sent to the upper $\mathrm{U}_{k/2}$. Note, the order of the inputs matches the interface perfectly, so no additional manipulation is required. The $k/2$ outputs of the upper (resp. lower) $\mathrm{U}_{k/2}$ are sent directly to the first (resp. second) half of the outputs of U_k. Again, interfaces match, and no manipulation is required.

We now only need to show how the inputs to the lower $\mathrm{U}_{k/2}$ are provided. These inputs could come from (any G_i^{Sim} gate of) the upper $\mathrm{U}_{k/2}$. Therefore, we also wire the outputs of upper $\mathrm{U}_{k/2}$ into a selection block $S_k^{k/2}$. This allows to direct, with duplicates, the output of any gate of upper $\mathrm{U}_{k/2}$ to any position of the input interface of lower $\mathrm{U}_{k/2}$ (and thus to any gate of lower $\mathrm{U}_{k/2}$). Additionally, (some of) lower $\mathrm{U}_{k/2}$'s inputs could come from the U_k inputs $in_{k+1}, ...in_{2k}$. Since the lower $\mathrm{U}_{k/2}$ simulates gates $G_{k/2+1}$ through G_k of C_k, inputs $in_{k+1}, ...in_{2k}$ are already ordered to match lower $\mathrm{U}_{k/2}$'s interface. Now, for each input of lower $\mathrm{U}_{k/2}$, we need to switch between the two input wires: one provided by upper $\mathrm{U}_{k/2}$ via $S_k^{k/2}$, and the other coming from U_k's input directly. This is easily achieved by a Y switching block. On the diagram, for ease of presentation, we combine the k of these Y blocks into a *mixing block* M_k, shown on Fig. 5.6(b) with $size(M_k) = k \cdot size(Y) = k$ and $depth(M_k) = 1$.

The base case of the recursive construction is U_1, a universal block implementing a single gate. U_1 is implemented by a single programmable gate as shown in Fig. 5.6(c). This completes the description of the recursive U_k construction.

The above immediately implies efficient methods of UC programming, given the circuit C_k. In particular, if the first (resp. second) input of a gate G_j in the lower half of C_k ($k/2 < j \leq k$) is connected to an input of C_k, the mixing block M_k is programmed to select the corresponding input in_{2j-1} (resp. in_{2j}) of U_k by programming Y_{2j-k-1} (resp. Y_{2j-k}) of M_k correspondingly (see Fig. 5.6(b)). Otherwise, if G_j is connected to an output of a gate G_i in the upper half of C_k ($1 \leq i \leq k/2$), M_k and $S_k^{k/2}$ are programmed to select the corresponding output from the upper $U_{k/2}$ block by programming Y_{2j-k-1} (resp. Y_{2j-k}) correspondingly and programming $S_k^{k/2}$ with $\sigma_{2j-k-1} = i$ (resp. $\sigma_{2j-k} = i$).

We now compute the complexity of our constructions U_k and UC (using selection block constructions of Chapter 5.3.4). Recall, the cost of Yao's garbled circuit depends only on its size, and not on depth. Note, $size(U_1) = 1, depth(U_1) = 1$.

$$
\begin{aligned}
size(U_k) &= 2size(U_{k/2}) + size(S_k^{k/2}) + size(M_k) \\
&= k \cdot size(U_1) + \sum_{i=0}^{\log(k)-1} 2^i (size(S_{k/2^i}^{k/2^{i+1}}) + size(M_{k/2^i})) \\
&= k + \sum_{i=0}^{\log(k)-1} 2^i (6\frac{k}{2^{i+1}} \log(\frac{k}{2^{i+1}}) + 3 + \frac{k}{2^i}) \\
&= k + 3k\log^2 k - 2k\log k - 3k \sum_{i=0}^{\log(k)-1} i + 3 \sum_{i=0}^{\log(k)-1} 2^i \\
&= k + 3k\log^2 k - 2k\log k - 3k \cdot 0.5(\log(k)(\log(k)-1)) + 3(k-1) \\
&= 1.5k\log^2 k - 0.5k\log k + 4k - 3; \\
depth(U_k) &= 2depth(U_{k/2}) + depth(S_k^{k/2}) + depth(M_k) = \ldots \\
&= k\log k + k + 4\log k - 12.
\end{aligned}
$$

Using the optimization of Chapter 5.3.5, U_k has complexity $size(U_k) = 1.5k\log^2 k - 1.5k\log k + 6k - 5$ and $depth(U_k) = k\log k + 4\log k - 11$.

U_k combined with input- and output-selection blocks of Chapter 5.3.4 as shown in Fig. 5.4, results in a UC construction of complexity

$$
\begin{aligned}
size(UC) &= 1.5k\log^2 k + 2.5k\log k + 9k + (u+2k)\log u + (k+3v)\log v \\
&\quad -2u - 4v + 1; \\
depth(UC) &= k\log k + 2k + v + 7\log k + 2\log u + 3\log v - 14.
\end{aligned}
$$

5.3.3 Generalized Permutation Blocks

To construct the efficient selection blocks S_v^u presented in Chapter 5.3.4 we need two useful generalizations of the permutation blocks of Waksman [Wak68].

P_u^u Permutation Block

A *permutation block* P_u^u is a programmable block that can be programmed to output any permutation of the inputs. Formally, given a permutation $(\pi_i)_{i=1}^u, \pi_i \in \{1, \ldots, u\}, \forall i \neq j : \pi_i \neq \pi_j$ that selects for the i-th output a unique input π_i, P_u^u computes $P(in_1, .., in_u) = (in_{\pi_1}, .., in_{\pi_u})$.

When u is a power of 2, Waksman [Wak68] describes an efficient recursive P_u^u construction built from X switching blocks. His P_u^u has $size(P_u^u) = 2u \log u - 2u + 2$ and $depth(P_u^u) = 2 \log u - 1$.

Waksman also gives an efficient recursive algorithm to program the X switching blocks of his construction. (Fig. 5.7 describes a slight generalization of Waksman's construction; fixing $u = v$ in Fig. 5.7 corresponds to Waksman's P_u^u.) The programming algorithm takes a $u \times u$ permutation matrix for the permutation (π_i) as input. It splits this $u \times u$ permutation matrix into two $u/2 \times u/2$ permutation matrices that are recursively implemented by the left and the right $P_{u/2}^{u/2}$ permutation sub-block and programs the X switching blocks correspondingly. Using a sparse matrix representation for the permutation matrices, this algorithm can be efficiently implemented in $O(u \log u)$.

We note that Waksman's construction can be naturally generalized to the cases where $u \neq v$, i.e. the number of inputs and outputs differ. Below we define the resulting objects (which we call "truncated permutation" and "expanded permutation" blocks), and present their efficient constructions.

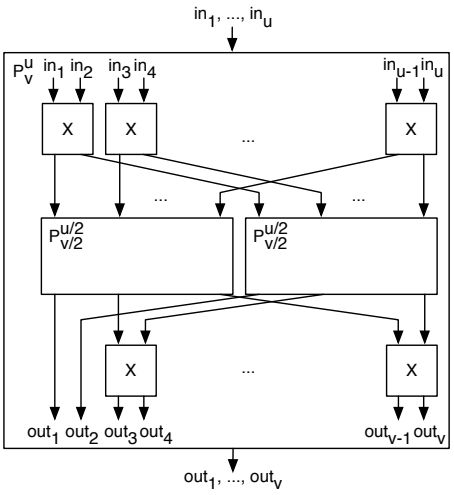


Figure 5.7: Recursive construction of a P_v^u permutation block

$TP_v^{u\geq v}$ Truncated Permutation Block

A $TP_v^{u\geq v}$ truncated permutation block permutes a subset of v of the u inputs to the $v \leq u$ outputs. The remaining $u - v$ input values are discarded. Formally, an output mapping $(\mu_i)_{i=1}^{v}$, $\mu_i \in \{1, \ldots, u\}, \forall j \neq i : \mu_i \neq \mu_j$ selects the μ_i-th input as the i-ths output. The truncated permutation block computes $TP(in_1, \ldots, in_u) = (in_{\mu_1}, \ldots, in_{\mu_v})$.

The $TP_v^{u\geq v}$ block is constructed recursively analogous to Waksman's permutation network construction as seen in Fig. 5.7. W.l.o.g we assume u and v are even at each recursion step (otherwise we introduce an unused dummy input or output with small overhead). If $u \geq 2$ the $TP_v^{u\geq v}$ truncated permutation block is divided into two $TP_{v/2}^{u/2\geq v/2}$ truncated permutation sub-blocks. The upper $u/2$ X switching blocks distribute the inputs of $TP_v^{u\geq v}$ to the two sub-blocks. The lower $(v/2-1)$ X switching blocks distribute the outputs of the two sub-blocks to the outputs of $TP_v^{u\geq v}$ as shown in Fig. 5.7. At the base of the recursion, if $v = 1$, a S_1^u selection block selects the intended input.

The $TP_v^{u\geq v}$ block is programmed using a natural generalization of Waksman's recursive programming algorithm. The intended output mapping (μ_i) is expressed as a $u \times v$ truncated permutation matrix. In each recursion step the algorithm splits the $u \times v$ matrix into two $u/2 \times v/2$ truncated permutation matrices implemented by the left and right sub-block and programs the X switching blocks accordingly. In the end of the recursion, if the truncated permutation matrix is a $u \times 1$ matrix with a one in the i-th row, the S_1^u selection block is programmed to select the i-th input value as output: $\sigma_1 = i$. This algorithm can be implemented in $O((u + v) \log v)$ using sparse matrix representations.

The complexity of this construction is

$$\begin{aligned}
size(TP_v^{u\geq v}) &= v \cdot size(S_1^{u/v}) + \sum_{i=0}^{\log(v)-1} 2^i \left(\frac{u}{2^{i+1}} + \frac{v}{2^{i+1}} - 1\right) \cdot size(X) \\
&= \left(\frac{u+v}{2} \log(v) - v + 1\right) \cdot size(X) + (u - v) \cdot size(Y) \\
&= (u + v) \log v + u - 3v + 2 \ ; \\
depth(TP_v^{u\geq v}) &= (2 \log v - 1) \cdot depth(X) + depth(S_1^{u/v}) \\
&= \log u + \log v - 1 \ .
\end{aligned}$$

$EP_{v\geq u}^u$ Expanded Permutation Block

An $EP_{v\geq u}^u$ expanded permutation block permutes the u inputs to a subset of u of the $v \geq u$ outputs. The remaining $v - u$ outputs are allowed to obtain any input value (they are intended to be discarded later and are called *dummy* outputs). Formally, an input mapping $(\mu_i)_{i=1}^{u}, \mu_i \in \{1, \ldots, v\}, \forall j \neq i : \mu_i \neq \mu_j$ specifies that the i-th input should be mapped to the μ_i-th distinct output. The expanded permutation block computes $EP(in_1, \ldots, in_u) = (out_1, \ldots, out_v)$ where $(out_s = in_r) \leftrightarrow (\mu_r = s), s \in \{1, \ldots, v\}, r \in \{1, \ldots, u\}$.

The construction of the $EP^u_{v\geq u}$ is analogous to the previously described $TP^{u\geq v}_v$ block. At the base of the recursion, if $u = 1$, the single input in_1 is connected to each of the v outputs.

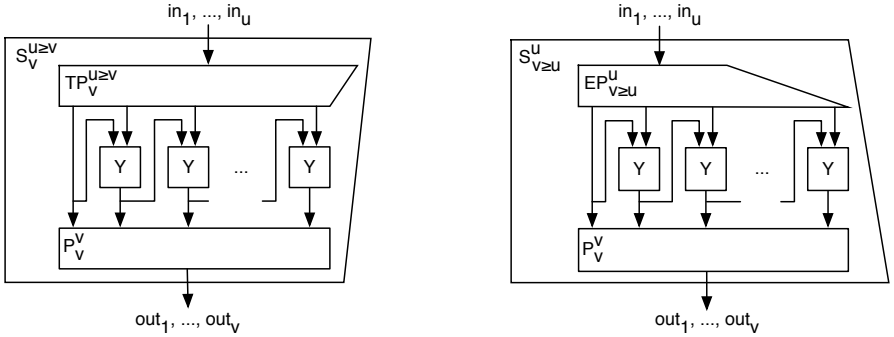
The programming algorithm of $EP^u_{v\geq u}$ is analogous to that of $TP^{u\geq v}_v$ as well. The input is a $u \times v$ matrix that corresponds to (μ_i) and it can be implemented in $O((u + v)\log u)$.

The construction has complexity

$$\begin{aligned} size(EP^u_{v\geq u}) &= \sum_{i=0}^{\log(u)-1} 2^i\left(\frac{u}{2^{i+1}} + \frac{v}{2^{i+1}} - 1\right)\cdot size(X) \\ &= (u+v)\log u - 2u + 2\ ; \\ depth(EP^u_{v\geq u}) &= (2\log u)\cdot depth(X) \\ &= 2\log u\ . \end{aligned}$$

5.3.4 Efficient Selection Blocks

We use truncated and expanded permutation blocks of the previous section to build efficient selection blocks S^u_v, used directly in the UC construction.



(a) Efficient $S^{u\geq v}_v$ selection block (b) Efficient $S^u_{v\geq u}$ selection block

Figure 5.8: Efficient S^u_v selection blocks

Efficient $S^{u\geq v}_v$ Selection Block

We obtain the efficient $S^{u\geq v}_v$ selection block from one $TP^{u\geq v}_v$ truncated permutation block, one P^v_v permutation block, and $(v-1)$ Y switching blocks as shown in Fig. 5.8(a).

It is not hard to see that the above $S^{u\geq v}_v$ is indeed a selection block, i.e. it can be programmed with any selection mapping $(\sigma_i)^v_{i=1}, \sigma_i \in \{1,\ldots,u\}$. To program $S^{u\geq v}_v$, first count the frequency of occurrence c_j of each input value in the output: $c_j = \#\{\sigma_i : \sigma_i = j; i \in \{1\ldots v\}\}; j \in \{1\ldots u\}$. Note, $0 \leq c_j \leq v$ and $\sum_{j=1}^u c_j = v$. The $TP^{u\geq v}_v$ truncated permutation block is programmed to

1. map the needed inputs ($c_j \neq 0$) to its ($\sum_{k=1}^{j-1} c_k$)-th output and
2. map the unused inputs ($c_j = 0$) to an unused (dummy) output.

The $(v-1)$ Y switching blocks connected to the outputs of $TP_v^{u \geq v}$ duplicate the needed inputs as necessary and feed them to the P_v^v permutation block. They are programmed as follows. If the right input of a Y block is a needed output (produced by Step 1), then the Y block selects it as output. Otherwise, the output of the neighbor Y block is selected. For each j, this construction inputs c_j copies of in_j into the P_v^v permutation block. P_v^v then permutes these values to the corresponding outputs indicated by the selection mapping (σ_i). The complexity of this construction is

$$\begin{aligned} size(S_v^{u \geq v}) &= size(TP_v^{u \geq v}) + (v-1) \cdot size(Y) + size(P_v^v) \\ &= ((u+v)\log v + u - 3v + 2) + (v-1) + (2v \log v - 2v + 2) \\ &= (u+3v)\log v + u - 4v + 3 \ ; \\ depth(S_v^{u \geq v}) &= depth(TP_v^{u \geq v}) + (v-1) \cdot depth(Y) + depth(P_v^v) \\ &= (\log u + \log v - 1) + (v-1) + (2 \log v - 1) \\ &= \log u + 3 \log v + v - 3 \ . \end{aligned}$$

Efficient $S_{v \geq u}^u$ Selection Block

An efficient $S_{v \geq u}^u$ selection block can be constructed and programmed analogously, but using a $EP_{v \geq u}^u$ expanded permutation block instead as shown in Fig. 5.8(b). Its complexity is

$$\begin{aligned} size(S_{v \geq u}^u) &= size(EP_{v \geq u}^u) + (v-1) \cdot size(Y) + size(P_v^v) \\ &= ((u+v)\log u - 2u + 2) + (v-1) + (2v \log v - 2v + 2) \\ &= (u+v)\log u + 2v \log v - 2u - v + 3 \ ; \\ depth(S_{v \geq u}^u) &= depth(EP_{v \geq u}^u) + (v-1) \cdot depth(Y) + depth(P_v^v) \\ &= (2 \log u) + (v-1) + (2 \log v - 1) \\ &= 2 \log u + 2 \log v + v - 2 \ . \end{aligned}$$

Improved S_{2u}^u Selection Block

In this section, we improve the efficient $S_{v \geq u}^u$ selection block construction shown before for the case $v = 2u$, most frequently used in our recursive construction of the universal block U_k. We improve by replacing the $EP_{v \geq u}^u$ expanded permutation block in the construction of $S_{v \geq u}^u$ in Fig. 5.8(b) with a smaller P_u^u permutation block and a different connection of the $(v-1)$ Y blocks as shown in Fig. 5.9. Our construction achieves

$$\begin{aligned} size(S_{2u}^u) &= 6u \log u + 3 \ ; \\ depth(S_{2u}^u) &= 4 \log u + 2u - 1 \ . \end{aligned}$$

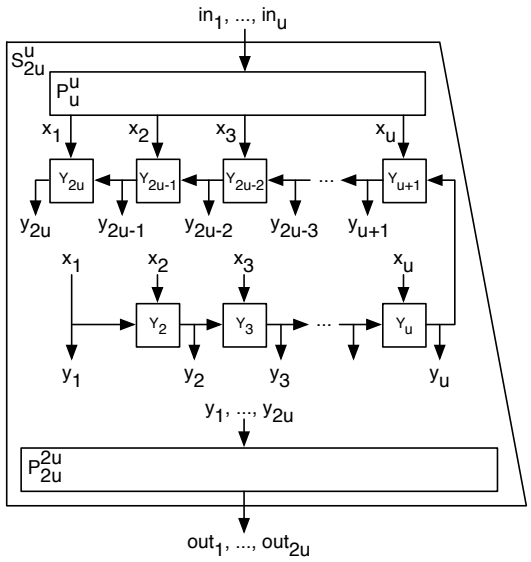


Figure 5.9: Improved S^u_{2u} selection block

Lemma. Construction of Fig. 5.9 is a S^u_{2u} selection block.

Proof. To prove this Lemma, we only need to show that the upper permutation block P^u_u together with the layer of Y blocks output the selected values (with the right number of duplicates each) in some order. (The rest, i.e. imposing the desired order, is done by the lower permutation block P^{2u}_{2u}.)

We use the network of Y blocks to duplicate (or omit) inputs as required by the selection block specification. The upper permutation block P^u_u can be programmed to deliver the desired input in_i to any Y-layer input x_j not already used by another input. For example, if input in_i needs to be duplicated c_i times, this can be achieved by programming the permutation to map in_i to x_j, and have blocks Y_j through Y_{j+c_i-1} to output x_j. This way, as required, the value in_i would be duplicated c_i times.

For efficiency reasons, the wiring of the Y-layer is limited. In particular, input x_i is delivered only to blocks Y_i and Y_{2u-i+1}, which are in column i. From there, x_i can be propagated "to the right" from Y_i (i.e. to blocks $Y_{i+1}, ...$, in the lower row) and/or "to the left" from Y_{2u-i+1} (i.e. to blocks $Y_{2u-i+2}, ...$, in the upper row). Note, blocks Y_i and Y_{2u-i+1} cannot receive different inputs from P^u_u. They, however, can produce different outputs, since one or both of them could be propagating the value of their neighbouring Y block.

It is not immediately clear that the inputs $in_1...in_u$ can be permuted such that the Y-layer can provide the right number of duplicates for each input. We show, that this in fact can be done. We observe that this permutation and the Y-layer programming can be reduced to the following box-packing problem.

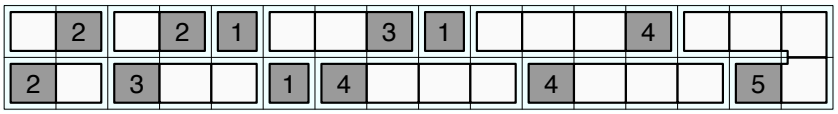


Figure 5.10: Valid arrangement of boxes produced by Algorithm 5.1 for boxes of size $(c_j) = \{2, 3, 1, 4, 4, 5, 4, 1, 3, 1, 2, 2, 0, 0, 0, 0\}$. Dark gray head cells contain size.

Box-packing. (See Fig. 5.10 for illustration.) There are u rectangular boxes of sizes $c_1, \ldots, c_u$, where $c_i \in \{0, \ldots, 2u\}$ and $\sum_{i=1}^{u} c_i = 2u$. Each non-empty i-th box consists of a head cell (dark gray), and $c_i - 1$ trailing cells (light gray). There is a rectangular $2 \times u$ grid of slots that consists of an upper row and a lower row. A box of size c_i occupies c_i consecutive slots in one row (one exception is that the right-most box might wrap around from the lower to the upper row, as seen on Fig. 5.10). The boxes in the upper row are oriented with heads to the right, and the boxes in the lower row are oriented with heads to the left. A *collision* occurs when two heads occupy slots in the same column. The arrangement of all u boxes is called *valid*, if it contains no collisions. (Note that a valid arrangement leaves no empty slots.) A solution to the box-packing problem is a valid arrangement.

A procedure for a valid arrangement of the boxes of sizes $c_1, \ldots, c_u$ gives the following natural programming of the P_u^u permutation block and the Y-layer. Associate (1-to-1) each input in_i of size c_i with a box of same size c_i and compute a valid arrangement. Then, input in_i is switched by P_u^u to x_j if the j-th column is occupied by the head of the box associated with in_i. Inputs in_i with $c_i = 0$ (unused inputs) are switched to the columns j which have no head boxes. Both switching blocks Y_i and Y_{2u-i+1} of each column i are programmed as follows. They select input x_i iff the corresponding slot in the valid arrangement is occupied by the head (otherwise, the output of the neighbored Y switching block is selected). It is not hard to see that this programming results in the desired output, given the corresponding valid arrangement of boxes.

The next Lemma shows an efficient box-packing procedure. This completes the proof of the previous Lemma. □

Lemma. Algorithm 5.1 efficiently produces a valid arrangement for any given set of u boxes of sizes $c_1, \ldots, c_u; 0 \leq c_j \leq 2u; \sum_{j=1}^{u} c_j = 2u$.

Proof. Note, since $\sum c_j = 2u$, for each box of size $2 + i$, there must be i boxes of size 1, or $i/2$ boxes of size 0, or a corresponding combination.

A) Algorithm 5.1 always puts all boxes and *terminates*. We first show that Step 2 eliminates all boxes of size 1. Indeed, suppose the contrary, a block of size 1 remains. Then, in each previous execution of Step 2a, we eliminated blocks of sizes $s_2 \geq s_1 \geq 2$ and $s_1 + s_2 - 4$ blocks of size 1, and in Step 2b we eliminated a block of size s_1 and $s_1 - 2$

Algorithm 5.1 Box-packing

0. Each box is always put in the leftmost unoccupied slots in the specified row.
1. Sort boxes by size in increasing order.
2. while there is at least one box of size 1, do
   a) if there are at least two boxes of minimal sizes $s_2 \geq s_1 \geq 2$ left
      i. put the box of size s_1 in the upper row
      ii. put remaining (but no more than s_1-2) boxes of size 1 in lower row
      iii. put the box of size s_2 in the lower row (possibly wrap around)
      iv. put remaining (but no more than s_2-2) boxes of size 1 in upper row
   b) else // there is only one box of size $s_1 \geq 2$ left
      i. put the remaining boxes of size 1 in the lower row
      ii. put the box of size $s_1 \geq 2$ in the lower row and wrap around
3. while there is at least one box of minimal size $s_3 \geq 2$ left, do
   a) if there is another box of minimal size $s_4 \geq s_3 \geq 2$ left
      i. put the box of size s_3 in the upper row
      ii. put the box of size s_4 in the lower row (possibly wrap around)
   b) else // there is only one box of size $s_3 \geq 2$ left
      i. put the box of size $s_3 \geq 2$ in the lower row and wrap around
4. end

blocks of size 1. Since $\sum c_j = 2u$, there could not have been more blocks of size 1 than we eliminated, and we arrive at contradiction. Further, Step 3 eliminates all remaining boxes of size ≥ 2. In each iteration, at least one box of size $s_3 \geq 2$ is eliminated either in Step 3(a)i or Step 3(b)i, until all boxes of size ≥ 2 are eliminated. (Observe, at each iteration, upper row "grows" not more than the lower. Thus, Algorithm's actions are always legal.)

B) Algorithm 5.1 produces a *valid* arrangement. We need to show that no step of Algorithm 5.1 causes a collision. It is easy to see that Step 2a and Step 2b never cause a collision. Further, once Step 2 has finished, the number of occupied slots in the upper row ω_{up} is less or equal to the number of occupied slots in the lower row ω_{down}, with $0 \leq \omega_{down} - \omega_{up} \leq s_2 - 2$ (here s_2 is the size of the most recently put block in Step 2(a)iv). Since the boxes are processed in increasing order, in Step 3, $s_3 \geq s_2 \geq 2$. If the box of size s_3 is the last remaining one, it is put in the lower

row in Step 3(b)i and, as is easy to see, doesn't cause a collision. Otherwise, in Step 3(a)i, the box of size s_3 is put in the upper row. The number of occupied slots in the upper row is now $\omega'_{up} = \omega_{up} + s_3$, and the upper row has at least two more occupied slots than the lower row: $\omega'_{up} - \omega_{down} = (\omega_{up} + s_3) - \omega_{down} \geq 2$. This implies that the next Step 3(a)ii doesn't cause a collision when putting the box of length $s_4 \geq s_3$ into the lower row. After Step 3(a)ii, the number of occupied slots in the lower row is $\omega'_{down} = \omega_{down} + s_4$. In the end of the current iteration of Step 3, the number of occupied slots in the upper row is again less or equal to the number of occupied slots in the lower row: $\omega'_{down} - \omega'_{up} = (\omega_{down} + s_4) - (\omega_{up} + s_3) = (\omega_{down} - \omega_{up}) + (s_4 - s_3) \geq 0$ and hence the length relationship between the upper and lower rows ($0 \leq \omega'_{down} - \omega'_{up} \leq s_4 - 2$) is the invariant of Step 3. Therefore, no iteration of Step 3 causes a collision. As no step causes a collision, Algorithm 5.1 produces a valid arrangement.

C) Algorithm 5.1 is *efficient*. Sorting of the u boxes in Step 1 costs $O(u \log u)$. Steps 2 and 3 have a runtime of $O(u)$, as in every iteration at least one box is eliminated. Hence the runtime of Algorithm 5.1 is in $O(u \log u)$. □

Depth-optimal Efficient Selection Blocks

In computationally secure SFE protocols, the only cost measurement is the size of a circuit - the number of gates that needs to be minimized. In some applications also the depth needs to be minimized as this determines the size of the shares like in GESS (see Chapter 3.3.3) or the number of rounds in multi-party protocols. This section shows how to reduce the depth of the practical universal circuit construction to $O(k)$ (which is asymptotically optimal) by reducing the depth of the selection blocks.

The previously described efficient selection block constructions $S_v^{u \geq v}$ and $S_{v \geq u}^{u}$ shown in Fig. 5.8 minimize the size to $O((u+v)\log(u+v))$ but NOT the depth which is $O(u+v)$. The depth of these efficient selection blocks can be minimized at the cost of a small increase in size as follows. If the chain of Y gates between the two permutation blocks is replaced with the extension of Waksman's permutation network $\overline{W}_{b'}$ described in [Hei86], the duplication of the values can be done with a depth of only $O(\log(u+v))$ and a small increase in size to $O((u+v)\log(u+v))$ instead of $v-1$ gates. This results in depth-optimal selection blocks $S^{depthOpt}$ with $size(S^{depthOpt}) \in O((u+v)\log(u+v))$ and $depth(S^{depthOpt}) \in O(\log(u+v))$.

Using these depth-optimal selection blocks instead, a depth-optimal practical universal circuit $\mathrm{UC}_{pract}^{depthOpt}$ can be constructed with a constant factor of increase in size only, $size(\mathrm{UC}_{pract}^{depthOpt}) \in O(k \log^2 k))$, but improved (asymptotically optimal) depth of $depth(\mathrm{UC}_{pract}^{depthOpt}) \in O(k)$ instead of $O(k \log k)$.

5.3.5 Optimization of the UC Construction

As the order of the two inputs of a gate simulation block G can be swapped by swapping its function table, we can omit the last row of X blocks in the lower P_k^k permutation block of the $S_k^{k/2}$ selection block in the construction of U_k (see Fig. 5.6(a), Fig. 5.9 and Fig. 5.7) and adapt the programming correspondingly. This results in a reduction of

$$\begin{aligned} \Delta size(U_k) &= \sum_{i=0}^{\log(k)-1} 2^i(k \cdot 2^{-i} - 2) \\ &= k \log k - 2k + 2 \ ; \\ \Delta depth(U_k) &= \sum_{i=0}^{\log(k)-1} 2^i \\ &= k - 1 \ . \end{aligned}$$

5.4 Comparison

We now compare our UC solution to the best previously known Valiant's UC [Val76]. Recall, we consider circuits $\mathrm{UC}_{u,v,k}$, universal for circuits of k gates, u inputs and v outputs. Valiant's UC is denoted by $\mathrm{UC}_{u,v,k}^{Valiant}$ and ours by $\mathrm{UC}_{u,v,k}$ with sizes

$$\begin{aligned} size(\mathrm{UC}_{u,v,k}^{Valiant}) = & & 19k \log k + & & 9.5u \log k + & & 9.5v \log k + O(k) \\ size(\mathrm{UC}_{u,v,k}) = & \ 1.5k \log^2 k + & 2.5k \log k + & \ (u + 2k) \log u + & & \ (k + 3v) \log v + O(k) \end{aligned} .$$

To help visualize the relationship, Table 5.1 shows sample relative sizes of our UC construction compared to Valiant's: $size_{rel} = \frac{size(\mathrm{UC}_{u,v,k})}{size(\mathrm{UC}_{u,v,k}^{Valiant})}$. We also note the break-even point $k_{eq} = k|_{size_{rel}=1}$ — the maximum size of circuits for which our UC is smaller.

Table 5.1: Comparison of Practical- and Valiant's UC construction

circuit inputs and outputs			break-even	relative size $size_{rel}$		
	u	v	point k_{eq}	$k = 1,000$	$k = 5,000$	$k = 10,000$
few	$o(k)$	$o(k)$	2,048	91.8%	110.2%	118.1%
	$0.5k$	$0.1k$	5,000	86.0%	100.1%	106.2%
	$0.5k$	$0.25k$	8,000	83.1%	96.4%	102.1%
	$1k$	$0.5k$	117,000	69.0%	79.5%	84.0%
many	$2k$	$1k$	26,663,000	53.6%	60.9%	64.1%

While Valiant's construction is asymptotically better, our UC is up to 50% smaller for small circuits, due to much lower constant factors. For PF-SFE, small circuits are of most interest, since only they can be evaluated efficiently today (indeed, UC for 5000-gate circuits has size $\approx 10^6$). In addition, our construction is more detailed and seems to

be much easier to implement than Valiant's. Thus, we believe that our UC construction is a good fit for *practical* PF-SFE.

In support of this contribution, FairplayPF was implemented as extension of the Fairplay SFE system [MNPS04] for general PF-SFE based on our UC construction as described in the next chapter.

6 Implementation of PF-SFE

FairplayPF is an extension of the Fairplay SFE system for general two-party PF-SFE based on UCs. The design and implementation of both systems, Fairplay for SFE and FairplayPF for PF-SFE, is described and compared in this chapter.

6.1 Fairplay

As already described in Chapter 3.3.2, Fairplay [MNPS04] implements general two-party SFE. It is written in JAVA and published under GPL (GNU General Public License). Source-code, binaries and example programs can be found on the Fairplay 1.0 project homepage [MNPS].

The left branch of Fig. 6.1 shows the high-level structure of Fairplay while Fig. 6.2(a) shows the design and program flow in detail:

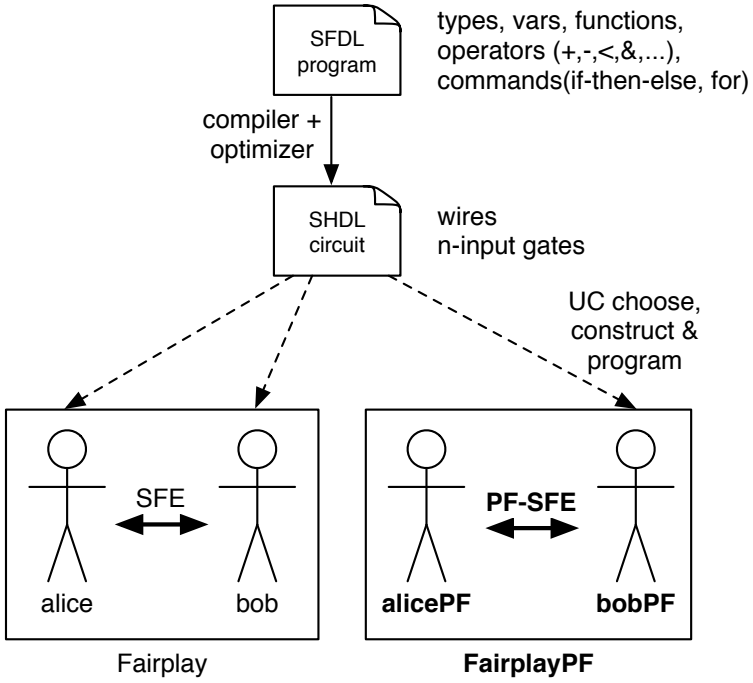


Figure 6.1: FairplayPF - Extending Fairplay for PF-SFE

The function the two parties Alice and Bob want to compute is described in SFDL (Secure Function Description Language). The SFDL program instructs a virtual "trusted party" what to do. SFDL resembles a simplified hardware description language like VHDL (Very high speed integrated circuit Hardware Description Language) including types, variables, functions, boolean operators ($+, -, <, \&, \ldots$) and control structures like if-then-else or for-loops with constant range. A detailed description of the syntax and semantics of SFDL can be found in [MNPS04, Appendix A]. Fairplay also includes a GUI (SFE.GUI.GUIMain) that assists the programmer in creating SFDL programs with graphical code templates.

Fairplay provides a secure hardware compiler and optimizer that compiles a SFDL file into a SHDL (Secure Hardware Description Language) circuit representation. SHDL circuits consist of wires and n-input gates with their input wires and function tables only. The compiler can be invoked by Alice (`run_alice -c SFDLprogram.txt`) and Bob (`run_bob -c SFDLprogram.txt`).

Using the SHDL circuit as input, Alice and Bob run the two-party SFE protocol to correctly and securely implement the fictional trusted party. The communication channel between Alice and Bob is a TCP connection. After having compiled the SFDL program to a SHDL circuit, Bob starts the server with

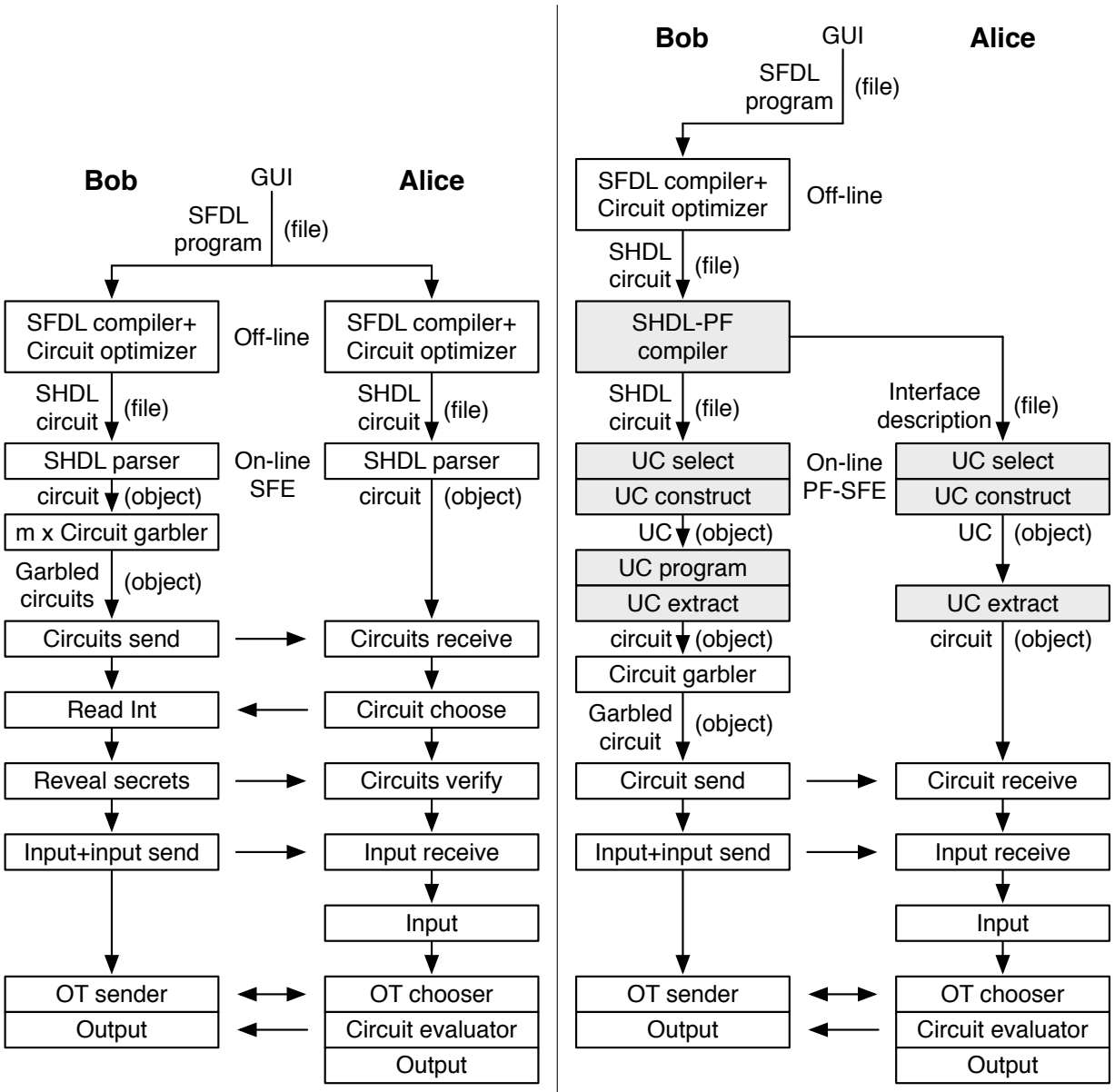
```
run_bob -r SFDLprogram.txt <seed> <ot_type>
```

where `<seed>` is a seed for the PRNG (Pseudo-Random Number Generator) and `<ot_type>` is the number of one of the several implemented OT protocols that should be used for OT. Similarly, Alice invokes the client on the compiled SHDL circuit with

```
run_alice -r SFDLprogram.txt <seed> <hostname> <num_iterations>
```

where `<hostname>` is the hostname (e.g. DNS name or IP address) of Bob and `<num_iterations>` is the number of rounds the protocol is executed.

After the TCP connection has been established, the client and the server execute the on-line SFE protocol as seen in Fig. 6.2(a). Bob garbles the circuit m times and sends them to Alice. Alice randomly choses one of them for evaluation and asks Bob to reveal the secrets of the other $m - 1$ garbled circuits. Alice verifies that these opened circuits are constructed according to the protocol and waits for Bob to send her the secrets corresponding to his inputs. The users Bob resp. Alice are asked to enter their secret inputs. Afterwards, Alice and Bob execute the specified OT protocol where Alice obtains the secrets corresponding to her inputs. In the end, Alice evaluates the garbled circuit using the input secrets, sends the output back to Bob and outputs it.



(a) Program flow of Fairplay

(b) Program flow of FairplayPF

Figure 6.2: Comparison of program flow

6.2 FairplayPF

In order to show practicability of universal circuit based PF-SFE described in Chapter 4.2 and the new practical universal circuit construction of Chapter 5.3, FairplayPF was implemented as part of this thesis. FairplayPF is an extension of the Fairplay 1.0 system for general two-party PF-SFE based on UCs. Source-code, binaries and example programs can be found on the FairplayPF homepage [KS].

A main design criteria of FairplayPF was to be as close to the original Fairplay as possible to allow easy migration between Fairplay (SFE) and FairplayPF (PF-SFE) back-and-forth. The similarities can be found in the high-level structure of FairplayPF shown in the right branch of Fig. 6.1 as well as the design and program flow shown in Fig. 6.2(b). The blocks that are newly introduced in FairplayPF are filled gray.

While Fairplay provides the SFDL program and the SHDL circuit to both parties Alice and Bob, in FairplayPF only Bob creates the SFDL program (optionally using the Fairplay GUI) and compiles the circuit (`run_bobPF -c SFDLprogram.txt`). This invokes first the original Fairplay SFDL compiler and afterwards the FairplayPF compiler that converts Fairplay's SHDL circuit containing n-input gates to a SHDL circuit that contains 2-input gates only. This is simply done by applying Shannon's expansion theorem [Sha49]:

$$f(x_1, x_2, \ldots, x_n) \quad = \quad \Big(\overline{x_1} \wedge f(0, x_2, \ldots, x_n)\Big) \vee \Big(x_1 \wedge f(1, x_2, \ldots, x_n)\Big)$$

Of course, this could be optimized further but the Fairplay compiler currently only produces $(n > 2)$-input gates in few cases like in full-adder cells that are compiled to 3-input gates (see also Chapter 3.3.4). The most efficient implementation of a full-adder has 5 two-input gates whereas the application of Shannon's expansion theorem produces 6 two-input gates.

The SHDL-PF compiler outputs an SHDL circuit for Bob (SFDLprogram.txt.Bob) and an interface description for Alice (SFDLprogram.txt.Alice). The SHDL circuit for Bob is compatible to those used in Fairplay but uses 2-input gates only (it can directly be used for SFE with Fairplay by invoking `run_bob -r SFDLprogram.txt.Bob ...` and `run_alice -r SFDLprogram.txt.Bob ...`). The interface description for Alice only contains the names of Alice's inputs and the dimension of the UC to be evaluated, namely the total number of inputs u, outputs v and gates k. Bob sends this generated interface description to Alice.

Now, Alice and Bob run the on-line PF-SFE protocol over a TCP connection. Bob starts the server with

```
run_bobPF -r SFDLprogram.txt <seed> <ot_type> [UC_type]
```

with the same semantics as in Fairplay's `run_bob` described before and an optional argument `[UC_type]` to specify the UC implementation that should be used. If this

parameter is omitted, the smallest UC implementation for the given circuit is selected automatically. Similarly, Alice invokes the client with

```
run_alicePF -r SFDLprogram.txt <seed> <hostname> <num_iterations> [UC_type]
```

After the TCP connection has been established, the client and the server execute the on-line PF-SFE protocol as seen in Fig. 6.2(b). Both players select the UC implementation that produces the smallest UC that is able to simulate the circuit with the given shape and construct it. Bob programs the UC with the given SHDL circuit and both parties extract a circuit representation of the (programmed in Bob's case) UC. Now, Bob garbles the programmed UC and sends it to Alice. The rest of the protocol is identical to the Fairplay protocol: Alice obtains the secrets corresponding to her inputs, evaluates the garbled UC, sends the result back to Bob and outputs it.

7 Conclusion

This thesis investigates and improves practical aspects of Secure Function Evaluation (SFE). The final chapter gives a summary of the contents of the previous chapters, concluding with an outlook w.r.t. possible directions for future work in practical SFE.

7.1 Summary

As explained in Chapter 1, general two-party SFE has become truly practical with Fairplay being the reference implementation for practical circuit-based SFE.

Boolean functions can be expressed either as circuits or - with some restrictions - as OBDDs which is shown in Chapter 2. The security of practical SFE protocols is based on fast symmetric cryptographic primitives such as semantically secure symmetric encryption or RO to improve efficiency. Oblivious transfer (OT) is used to send the garblings corresponding to evaluators' inputs while guaranteeing his privacy. Two different types of adversaries - semi-honest and malicious - with different power w.r.t. being able to deviate from the protocol in order to learn additional information.

Practical SFE protocols are given in Chapter 3. After the formal definition and practical examples for SFE, the commonalities of these SFE protocols are shown. They all have the same protocol structure and can be extended to be secure in the malicious model in a similar way. Circuit-based practical SFE protocols include the classical protocol of Yao, its RO based implementation Fairplay, the information-theoretically secure SFE protocol GESS and our newly introduced combination of the latter two, improved SFE that allows free evaluation of XOR gates with many practical applications. Also, OBDD representation of functions can be directly used for SFE. The combination of different SFE protocols as described in the summary can be used to obtain more efficient practical SFE protocols.

In some applications, the function to be evaluated needs to be private as explained in Chapter 4. PF-SFE can be reduced to SFE of a larger UC for functions expressed as circuit or the OBDD representation can be used directly with a small overhead only.

UCs can be programmed to simulate the functionality of any circuit. Two possible constructions of Valiant and our new practical UC construction are compared w.r.t. their application in PF-SFE in Chapter 5.

PF-SFE was implemented as extension of the Fairplay SFE system and the practical UC construction, called FairplayPF as described in Chapter 6. The design of FairplayPF is very close to the original Fairplay system and allows easy migration and re-use of

Fairplay SFE programs. FairplayPF shows practicability of PF-SFE for circuits with approximately 5000 gates. Using improved SFE as underlying protocol, this can be improved by another factor of 4.

7.2 Outlook

> Civilization is the progress toward a society of privacy. The savage's whole existence is public, ruled by the laws of his tribe. Civilization is the process of setting man free from men.
> Ayn Rand (1905 - 1982), The Fountainhead (1943)

Indeed, the right for privacy is a human right as stated in article 12 of The Universal Declaration of Human Rights [OHC]:

> No one shall be subjected to arbitrary interference with his privacy, family, home or correspondence, nor to attacks upon his honour and reputation. Everyone has the right to the protection of the law against such interference or attacks.

But are we still on the way of civilization toward a society of privacy? The 2007 International Privacy Ranking [Int07] again showed the alarming decay in the protection of privacy in many countries including Germany, the US and Canada. Fig. 7.1 shows the map of surveillance societies around the world from this study.

But not only the states violate our privacy - what if we have to reveal important parts of our privacy in form of confidential data in order to get something we really need? For example our credit history in order to qualify for a credit or our medical history to qualify for an insurance or a job? Of course the providers of a service have to check whether we qualify for the service but do they really need access to the appliers' personal data to do so? The problem is the potential misuse of personal data - even if not intentionally maybe accidentally [Sch06]. This thesis showed in many examples, how practical SFE resp. PF-SFE can be used to solve the problem of evaluating a public resp. private function on private data while maintaining data privacy.

Of course, protocols for SFE and PF-SFE are much more expensive (in terms of computation and communication) than giving the private data to the service provider who evaluates the function directly on it. But how can users insist on their right for privacy and demand privacy preservation from service providers? Simply by not giving out their private data.

One solution is to boycott those providers that do not provide privacy preservation. Further, SFE and PF-SFE need to be pushed further toward usability and practicability to minimize the overhead that service providers have to pay. Then they might seriously consider using these secure protocols in order to have new marketing arguments for privacy-aware customers.

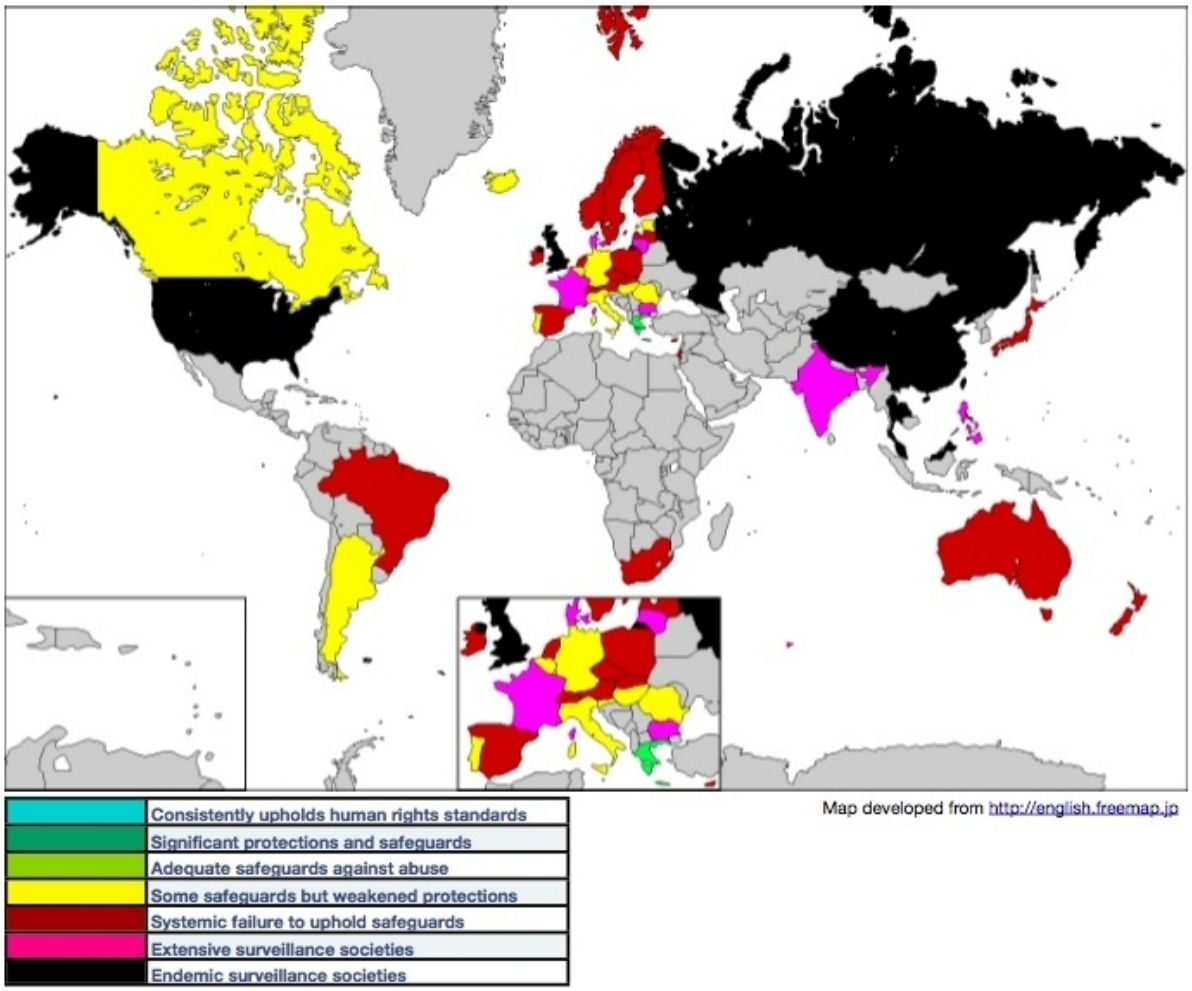


Figure 7.1: Map of Surveillance Societies around the world (from [Int07]).

This justifies further directions of research in practical SFE and PF-SFE, such as:

- Extending Fairplay and FairplayPF with our improved SFE protocol of Chapter 3.3.4.
- Using GESS in the bottom part of circuits as described in Chapter 3.5.
- Implementing the OBDD-based protocol for PF-SFE of Chapter 4.3 and compare it with FairplayPF.
- Combining circuit- and OBDD-based sub-functions for SFE or PF-SFE.
- Implementing Valiant's UC construction for practical use in FairplayPF to be able to evaluate larger circuits and to compare it with our practical UC construction.
- Using hardware acceleration like FPGAs or cryptographic coprocessors for SFE.

Appendices

List of Figures

List of Tables

List of Algorithms and Protocols

List of Acronyms

AES	Advanced Encryption Standard
DAG	Directed Acyclic Graph
FPGA	Field-Programmable Gate Array
GESS	Gate Evaluation Secret Sharing
GUI	Graphical User Interface
OBDD	Ordered Binary Decision Diagram
OT	Oblivious Transfer
PF-SFE	Secure Function Evaluation of Private Functions
PP	Permute and Point
PRNG	Pseudo-Random Number Generator
resp.	respectively
RO	Random Oracle
SFDL	Secure Function Description Language
SFE	Secure Function Evaluation
SHA	Secure Hash Algorithm
SHDL	Secure Hardware Description Language
UC	Universal Circuit
VHDL	Very high speed integrated circuit Hardware Description Language
w.h.p.	with high probability
w.r.t.	with respect to
XOR	exclusive-or

Index

P

R

S

T

U

X

Bibliography

[ACCK01] Joy Algesheimer, Christian Cachin, Jan Camenisch, and Gunter Karjoth. Cryptographic security for mobile code. In *SP '01: Proceedings of the IEEE Symposium on Security and Privacy*, page 2. IEEE Computer Society, 2001.

[AIR01] William Aiello, Yuval Ishai, and Omer Reingold. Priced oblivious transfer: How to sell digital goods. In *Advances in Cryptology – EUROCRYPT 2001*, volume 2045 of *LNCS*, pages 119–135. Springer, 2001.

[ALR99] Eric Allender, Michael C. Loui, and Kenneth W. Regan. Complexity classes. In Mikhail J. Atallah, editor, *Algorithms and Theory of Computation Handbook*, chapter 27. CRC Press, 1999.

[AMP04] Gagan Aggarwal, Nina Mishra, and Benny Pinkas. Secure computation of the k-th ranked element. In *Advances in Cryptology – EUROCRYPT 2004*, volume 3027 of *LNCS*. Springer, 2004.

[Bel98] Mihir Bellare. Practice-oriented provable-security. In *ISW '97: Proceedings of the First International Workshop on Information Security*, pages 221–231, London, UK, 1998. Springer-Verlag.

[BK06] Ian F. Blake and Vladimir Kolesnikov. Conditional encrypted mapping and comparing encrypted numbers. In *Financial Cryptography and Data Security, FC06*, volume 4107 of *LNCS*, pages 206–220. Springer, 2006.

[BLW95] Beate Bollig, Martin Löbbing, and Ingo Wegener. Simulated annealing to improve variable orderings for OBDDs. ACM/IEEE International Workshop on Logic Synthesis (IWLS), 1995.

[BM89] Mihir Bellare and Silvio Micali. Non-interactive oblivious transfer and applications. In *CRYPTO '89: Proceedings on Advances in cryptology*, pages 547–557, New York, NY, USA, 1989. Springer-Verlag New York, Inc.

[BPSW07] Justin Brickell, Donald E. Porter, Vitaly Shmatikov, and Emmett Witchel. Privacy-preserving remote diagnostics. In *Proc. ACM CCS*, pages 498–507, New York, NY, USA, 2007. ACM.

[BR93] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *ACM Conference on Computer and Communications Security*, pages 62–73, 1993.

[Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[Bry91] Randal E. Bryant. On the complexity of VLSI implementations and graph representations of boolean functions with application to integer multiplication. *IEEE Trans. Comput.*, 40(2):205–213, 1991.

[BW96] Beate Bollig and Ingo Wegener. Improving the variable ordering of OBDDs is NP-complete. *Transactions on Computers*, 45(9):993–1002, Sep 1996.

[Can96] Ran Canetti. *Studies in Secure Multiparty Computation and Applications*. PhD thesis, 1996.

[CCKM00] Christian Cachin, Jan Camenisch, Joe Kilian, and Joy Müller. One-round secure computation and secure autonomous mobile agents. In *ICALP '00: Proceedings of the 27th International Colloquium on Automata, Languages and Programming*, pages 512–523, London, UK, 2000. Springer-Verlag.

[CGH98] Ran Canetti, Oded Goldreich, and Shai Halevi. The random oracle methodology, revisited. In *Proc. 30th ACM Symp. on Theory of Computing*, pages 209–218, 1998.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, September 2001.

[DBG96] Rolf Drechsler, Bernd Becker, and Nicole Gockel. Genetic algorithm for variable ordering of OBDDs. *Computers and Digital Techniques, IEEE Proceedings*, 143(6):364–368, Nov 1996.

[DBP96] Hans Dobbertin, Antoon Bosselaers, and Bart Preneel. RIPEMD-160: A strengthened version of RIPEMD. In *Proceedings of the Third International Workshop on Fast Software Encryption*, pages 71–82, London, UK, 1996. Springer-Verlag.

[FA05] Keith B. Frikken and Mikhail J. Atallah. Achieving fairness in private contract negotiation. In *Financial Cryptography and Data Security, FC05*, pages 270–284, 2005.

[FAZ05] Keith B. Frikken, Mikhail J. Atallah, and Chen Zhang. Privacy-preserving credit checking. In *EC '05: Proceedings of the 6th ACM conference on Electronic commerce*, pages 147–154, New York, NY, USA, 2005. ACM Press.

[FMK91] Masahiro Fujita, Yusuke Matsunaga, and Taeko Kakuda. On variable ordering of binary decision diagrams for the application of multi-level logic synthesis. In *EURO-DAC '91: Proceedings of the conference on European design automation*, pages 50–54, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.

[FPRJ04] Joan Feigenbaum, Benny Pinkas, Raphael S. Ryger, and Felipe Saint Jean. Secure computation of surveys. In *EU Workshop on Secure Multiparty Protocols (SMP)*. ECRYPT, 2004.

[Für07] Martin Fürer. Faster integer multiplication. In *STOC '07: Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*, pages 57–66, New York, NY, USA, 2007. ACM.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game. In *Proc. 19th ACM Symp. on Theory of Computing*, pages 218–229. ACM, 1987.

[Gol01] Oded Goldreich. *Foundations of Cryptography*, volume 1: Basic Tools. Cambridge University Press, 2001. Draft available at `http://www.wisdom.weizmann.ac.il/~oded/foc-vol1.html`.

[Gol04] Oded Goldreich. *Foundations of Cryptography*, volume 2: Basic Applications. Cambridge University Press, 2004. Draft available at `http://www.wisdom.weizmann.ac.il/~oded/foc-vol2.html`.

[Hei86] Friedhelm Meyer auf der Heide. Efficient simulations among several models of parallel computers. *SIAM J. Comput.*, 15(1):106–119, 1986.

[Int07] Privacy International. The 2007 international privacy ranking, 2007. `http://www.privacyinternational.org/article.shtml?cmd[347]=x-347-559597`.

[Kil88] Joe Kilian. Founding cryptography on oblivious transfer. In *Proc. 20th ACM Symp. on Theory of Computing*, pages 20–31, Chicago, 1988. ACM.

[KJGB06] Louis Kruger, Somesh Jha, Eu-Jin Goh, and Dan Boneh. Secure function evaluation with ordered binary decision diagrams. In *Proc. ACM CCS*, pages 410–420. ACM Press, 2006.

[Kol05] Vladimir Kolesnikov. Gate evaluation secret sharing and secure one-round two-party computation. In *Advances in Cryptology – ASIACRYPT 2005*, volume 3788 of *LNCS*, pages 136–155. Springer, 2005.

[Kol06] Vladimir Kolesnikov. *Secure Two-Party Computation and Communication*. PhD thesis, University of Toronto, June 2006.

[KS] Vladimir Kolesnikov and Thomas Schneider. FairplayPF. `http://thomaschneider.de/FairplayPF`.

[KS08] Vladimir Kolesnikov and Thomas Schneider. A practical universal circuit construction and secure evaluation of private functions. In *Financial Cryptography and Data Security, FC08*, LNCS. Springer, 2008.

[LB05] Wolfgang Lenders and Christel Baier. Genetic algorithms for the variable ordering problem of binary decision diagrams. In *FOGA*, pages 1–20, 2005.

[LP04] Yehuda Lindell and Benny Pinkas. A proof of Yao's protocol for secure two-party computation. Cryptology ePrint Archive, Report 2004/175, 2004. `http://eprint.iacr.org/2004/175`.

[LP07] Yehuda Lindell and Benny Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In *Advances in Cryptology – EUROCRYPT 2007*, volume 4515 of *LNCS*, pages 52–78. Springer-Verlag, 2007.

[MNPS] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay 1.0 project. `http://www.cs.huji.ac.il/project/Fairplay/fp1.html`.

[MNPS04] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay — a secure two-party computation system. In *USENIX*, 2004.

[NIS01] NIST, U.S. National Institute of Standards and Technology. Federal information processing standards publication (FIPS 197). Advanced Encryption Standard (AES), November 2001. `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`.

[NIS02] NIST, U.S. National Institute of Standards and Technology. Federal information processing standards publication (FIPS 180-2). Announcing the Secure Hash Standard, August 2002. `http://csrc.nist.gov/publications/fips/fips180-2/fips-180-2.pdf`.

[NP01] Moni Naor and Benny Pinkas. Efficient oblivious transfer protocols. In *SODA '01: Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 448–457, Philadelphia, PA, USA, 2001. Society for Industrial and Applied Mathematics.

[NPS99] Moni Naor, Benny Pinkas, and Reuben Sumner. Privacy preserving auctions and mechanism design. In *1st ACM Conf. on Electronic Commerce*, pages 129–139, 1999.

[NSA03] NSA, U.S. National Security Agency. CNSS policy no. 15, fact sheet no. 1. National policy on the use of the Advanced Encryption Standard (AES) to protect national security systems and national security information, June 2003. `http://www.cnss.gov/Assets/pdf/cnssp_15_fs.pdf`.

[OHC] OHCHR, United Nations Office of the High Commisioner for Human Rights. Universal declaration of human rights. `http://www.unhchr.ch/udhr/lang/eng.htm`.

[OS05] Rafail Ostrovsky and William E. Skeith III. Private searching on streaming data. In *Advances in Cryptology – CRYPTO 2005*, volume 3621 of *LNCS*, pages 223–240, 2005.

[Pin02] Benny Pinkas. Cryptographic techniques for privacy-preserving data mining. *SIGKDD Explor. Newsl.*, 4(2):12–19, 2002.

[PW92] Birgit Pfitzmann and Michael Waidner. How to break and repair a "provably secure" untraceable payment system. In *Advances in Cryptology – CRYPTO 91*, pages 338–350, London, UK, 1992. Springer-Verlag.

[Rud93] Richard Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *ICCAD '93: Proceedings of the 1993 IEEE/ACM international conference on Computer-aided design*, pages 42–47, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[Sch06] Bruce Schneier. The eternal value of privacy. *Wired News*, May 18 2006. `http://www.wired.com/politics/security/commentary/securitymatters/2006/05/70886`.

[Sha49] Claude E. Shannon. The synthesis of two-terminal switching circuits. *Bell Systems Technical Journal*, 28(1):59–98, 1949.

[SS71] Arnold Schönhage and Volker Strassen. Schnelle Multiplikation großer Zahlen (Fast multiplication of large numbers). *Computing*, 7(3):281–292, 1971.

[SYY99] Tomas Sander, Adam Young, and Moti Yung. Non-interactive cryptocomputing for NC^1. In *Proc. 40th IEEE Symp. on Foundations of Comp. Science*, pages 554–566, New York, 1999. IEEE.

[Val76] Leslie G. Valiant. Universal circuits (preliminary report). In *Proc. 8th ACM Symp. on Theory of Computing*, pages 196–203, New York, NY, USA, 1976. ACM Press.

[Vol99] Heribert Vollmer. *Introduction to Circuit Complexity: A Uniform Approach*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[Wak68] Abraham Waksman. A permutation network. *J. ACM*, 15(1):159–163, 1968.

[Weg87] Ingo Wegener. *The complexity of Boolean functions*. John Wiley & Sons, Inc., New York, NY, USA, 1987.

[Woe05] Philipp Woelfel. Bounds on the OBDD-size of integer multiplication via universal hashing. *J. Comput. Syst. Sci.*, 71(4):520–534, 2005.

[Yao82] Andrew C. Yao. Protocols for secure computations. In *Proc. 23rd IEEE Symp. on Foundations of Comp. Science*, pages 160–164, Chicago, 1982. IEEE.

[Yao86] Andrew C. Yao. How to generate and exchange secrets. In *Proc. 27th IEEE Symp. on Foundations of Comp. Science*, pages 162–167, Toronto, 1986. IEEE.