Analysis and Extensions of the PCF Secure Two-Party Computation Compiler

Analyse und Erweiterungen des PCF Secure Two-Party Computation Compilers
Bachelor-Thesis von Benedikt Hiemenz
Tag der Einreichung:

1. Gutachten: Dr. Thomas Schneider
2. Gutachten: MSc. Daniel Demmler

TECHNISCHE UNIVERSITÄT DARMSTADT

EC SPRIDE

CASED

Analysis and Extensions of the PCF Secure Two-Party Computation Compiler
Analyse und Erweiterungen des PCF Secure Two-Party Computation Compilers

Vorgelegte Bachelor-Thesis von Benedikt Hiemenz

1. Gutachten: Dr. Thomas Schneider
2. Gutachten: MSc. Daniel Demmler

Tag der Einreichung:

Erklärung zur Bachelor-Thesis

Hiermit versichere ich, die vorliegende Bachelor-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 28. August 2014

(Benedikt Hiemenz)

Abstract - English

In 2013, the Portable Circuit Format (PCF) project was started to improve the way circuits for secure computation can efficiently be built, stored and evaluated. Secure computation is a growing field of cryptography, especially in the last decade and allows two or more parties to jointly evaluate a common function over their inputs without the need to disclose any information but the final result of the calculation. To allow an evaluation in such an oblivious way, the common function is first translated into a circuit, that consists of the combination of many connected Boolean gates. These - usually very big - circuits have to be processed efficiently, which is a known problem of secure computation, but necessary to make it usable for a bigger audience and hence assist in its dissemination.

In the course of this thesis, all components of the PCF project are described as well as their relation to each other. In addition, different improvements for the project have been implemented with regard to recent researches. The existing implementation that is used for the function evaluation based on the known approach of Yao's Garbled Circuits (GC). We add an implementation of the Goldreich, Micali and Wigderson (GMW) protocol, since the GMW approach provides a more efficient function evaluation for certain circuits. With the aim to support different arithmetic operations in addition to the existing Boolean ones, we introduce some new expressions for the circuit evaluation description. The advantages and disadvantages of all implemented project modifications are presented in detail in the course of this thesis.

Abstract - German

Die vorliegende Thesis beschäftigt sich mit verschiedenen Aspekten des Portable Circuit Format (PCF) Projektes, mit dessen Hilfe sich Circuits für Secure Computation effizienter aufbauen, speichern und verarbeiten lassen. Secure Computation bezeichnet ein in den letzten Jahren wachsendes Feld der Kryptographie, bei dem zwei oder mehr Parteien gemeinsam eine Funktion auf ihren Eingaben auswerten, ohne Informationen über ihre Eingaben preisgeben zu müssen. Lediglich die Funktionsausgabe ist am Ende allen Parteien bekannt. Um eine solche Berechnung zu ermöglichen, übersetzen viele bekannte Verfahren die Funktion zuerst in einen Circuit, eine Struktur aufgebaut aus der Verknüpfung Boolescher Gates. Diese - meist riesigen - Circuits effizient zu verarbeiten ist ein bekanntes Problem von Secure Computation, aber auch Voraussetzung, um es für einen größeren Kreis von Entwicklern und deren Anwendungen verfügbar zu machen.

Die einzelnen Komponenten des PCF Projektes werden im Rahmen dieser Thesis vorgestellt sowie deren Zusammenspiel beschrieben. Darüber hinaus wurden verschiedene Verbesserungen für das Projekt realisiert, welche neuere Forschungsergebnisse berücksichtigen. Wir ergänzen die vorhandene Implementation der Funktionsauswertung, welche auf dem Verfahren von Yao's Garbled Circuits (GC) basiert, um das Verfahren von Goldreich, Micali and Wigderson (GMW), einer in vielen Fällen effektiveren Möglichkeit Funktionen nach dem Secure Computation Prinzip auszuwerten. Zudem verbessern wir die Möglichkeiten die Auswertung des Circuits zu beschreiben, indem wir arithmetische Operationen neben den vorhandenen Booleschen einführen. Die Vor- und Nachteile aller Projektmodifikationen werden im Verlauf dieser Thesis ausführlich dargestellt.

Contents

1 Introduction

Privacy protection is an important topic for many technologies and products nowadays. More and more technical devices, which collect and handle our private data continually, are ubiquitous in our everyday life. Most people appreciate the advantages of these devices, like smartphones, but also want to retain control over their data. They want to decide, who is allowed to use their data and for example under what circumstances the data may be used or shared with third parties. Existing practices for dealing with private data are often not transparent for customers and mainly based on confidence in the provider.

Secure computation could play an important role to give people back more control over the processing of their private data. It is a field of cryptography, in which a lot of research has been done during the last decade and deals with a problem Andrew C. Yao has introduced in 1986 [Yao86]. Nowadays it is receiving more and more practical relevance: How can n parties with inputs x_i ($i \in \{1, 2, ..., n\}$) jointly compute a function $f(x_1, x_2, .., x_n)$ without revealing any information but the output of the function, especially no information about other parties' inputs? Additionally, secure computation does not need a trusted third party to be involved, which simply could compute and share the function output.

For a secure process, a secure computation protocol has to guarantee particular properties, including privacy and correctness. No information but the function's output should be leaked during the procedure and as a result of correctness, all parties can be sure to obtain the correct output at the end of the joint evaluation. This kind of evaluation is called Secure Function Evaluation (SFE). However, many existing secure computation projects assume specialized knowledge for their procedure, which prevent their dissemination. Other approaches do not scale very well and hence are not usable for many functions of practical interest.

In 2013, a new approach was presented by a developer team from Virginia and Oregon [KsMB13a]. The project is named after a newly developed format called Portable Circuit Format (PCF), which allows a compact and efficient description of a function used for SFE. The following thesis is based on this project. Compared to prior works, the PCF project tries to enable both, good scalability and an usage without specialized knowledge. Using PCF, it is easy for developers to work in their known language and compile any function to a secure computation protocol. Many optimizations have been included in the compiling process. Once the function is compiled, the SFE works similar to a program that is building and evaluating a Boolean circuit at runtime. This approach provides an promising way to work efficiently with circuits and thereby has a good scalability. Altogether, the PCF project has potential to increase the usage of secure computation.

1.1 Motivation

Although described first in the 1980s, due to the high performance secure computation needs, it has rather been of a theoretical issue for a long time. But for the last years many researchers tried to improve existing protocols or their implementations or create new ones with regard to performance issues, especially for SFE. In addition, computational resources of electronic devices as well as communication bandwidth is growing rapidly every year and as a consequence even mobile devices are in focus of secure computation developers nowadays. Moreover, many people have become more sensitive in dealing with their private data in technical devices in recent years.

The PCF project attempts to provide a practical application, but it is a young project and further improvements are needed. Because of its early stage of development, the components belonging to the project require more tests. Although their conceptual elaboration is described in detail and well-conceived, the practical implementation needs further improvements, too. A set of various instructions is supported by the project to describe any Boolean circuit, but unfortunately, no assistance or reference for the instructions exist so far. That makes it even more difficult to understand and hence improve the procedure.

This thesis describes supplementary improvements within the project's design and implementation. The PCF project applies Yao's Garbled Circuits (GC), one of the most established and efficient approach for secure two-party computation today. As recent researches [CHK+12], [SZ13] have shown that the approach of Goldreich, Micali and Wigderson (GMW) provides a better performance for circuits with low depth, we would like to add GMW as another backend. Another improvement we have implemented concerns the provided expressions. Because PCF only allows Boolean operations within its circuit description, we want to extend the format in order to support arithmetic ones as well. This could facilitate developers and users to get a better understanding of PCF and hence increase its usability.

1.2 Contributions and Outline

This thesis is concerned with the PCF project, especially with its interpreter part and its interaction with different approaches for secure computation. At first, an overview on the fundamental concepts of secure computation, which are significant to this thesis, is given in Chapter 2. Afterwards, a number of related works are presented in Chapter 3.

Chapter 4 introduces and explains the main components of the existing PCF project, its compiler and interpreter. Their relation to each other is shown as well as the advantages of them. In the next chapters we describe the two project versions, that have been published until now.

Chapter 5 describes PCF version 1, its functionality and interpreter part with focus on the Java implementation. Since the PCF developer did not publish a documentation about the meaning of each instruction, we have written a documentation that illustrates the existing PCF 1 instruction set. We explain all instructions and give a short example for each of them. After that, we add our newly written implementation of GMW to the existing PCF backend. This modification is tested and compared to the prior implementation and supplemented by a summary of the advantages and disadvantages of our new backend as well as an overview of future works on this topic.

Chapter 6 focuses on PCF version 2 and is structured similar to Chapter 5. The interpreter part, written by the PCF developers in C and C++, is described first. We show the differences to the previous version, as well. In the next section, we present our documentation about the existing instruction set of PCF 2 including an example for each instruction. Our subsequent modifications for the PCF 2 interpreter are twofold: at first we add our newly written GMW implementation as another backend. Afterwards, we extend the instruction set to enable the evaluation of arithmetic operations in addition to the Boolean ones. We therefore modify several parts of the existing components. The last section gives a discussion about the advantages of our arithmetic operations and its future work.

Finally, the results of this thesis are summarized in Chapter 7.

2 Preliminaries

Generic secure computation protocols can be divided into two main approaches: the concept of Garbled Circuits, developed by Yao in 1986 [Yao86] and GMW, a protocol designed by Goldreich, Micali and Wigderson in 1987 [GMW87]. Both approaches are described in the next section. In addition, the Oblivious Transfer protocol is important for these approaches. It is also explained in the course of the next section as well as other fundamental concepts playing a significant role in this thesis.

2.1 Adversary Model

An adversary model classifies several attacks against a certain protocol design. In most cases, the attacker is a regular participant of the system, who attempts to obtain unauthorized information access during the process. Various models have been developed over the years, not only for cryptographic protocols. Basically, the attacker's capabilities are divided into two classifications, a malicious (or active) and a semi-honest (or passive) behaviour.

Semi-honest A participant is called semi-honest (or passive) adversary, when he complies with the protocol rules but beyond that tries to receive additional (sensitive) information, whether about the system itself or another participant. This kind of attack is often used in secure computation as standard assumption, which each protocol at least has to be secure against.

Malicious A malicious (or active) adversary model describes an attack, where the adversary's behaviour includes conscious deviations from protocol rules. The adversary has no restriction in his attempt or rules of conduct to attain further information, he can send arbitrary messages, abort the protocol and much more. Providing security against these attacks are much more complicated.

2.2 Boolean Circuits

The function representation for secure computation is often described as Boolean circuit. A Boolean circuit is composed of Boolean gates and has n inputs and m outputs. An example is shown in Figure 1, where $n = 5$ and $m = 2$.

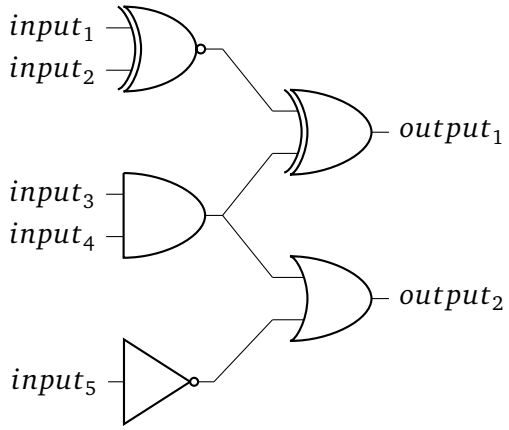

Figure 1: Boolean circuit example

Any function can be used for secure computation, if the function has a corresponding Boolean circuit representation. However, as usually many Boolean gates are required to built a complete function - even for simple ones - computing and evaluating functions in a secure and oblivious way needs a lot of memory and computational power. The architecture of Boolean circuits only enables a sequential evaluation, as each gate depends on another gate's output. Various techniques exist to find gates that could be grouped into layers and thereby allow a parallel evaluation. Another issue applies to the identification of reusable parts within the Boolean circuit, a technique that would reduce the memory consumption. Creating an optimal Boolean circuit for a function, which has the lowest possible depth and thereby enables good prerequisites for a parallel evaluation, is a known optimization problem of secure computation, but not part of this thesis.

2.3 Oblivious Transfer

Oblivious Transfer (OT) is a protocol that allows two parties, a sender and a receiver, to exchange information with minimal exposure. The first description of OT was written by Rabin in 1981 [Rab81]. A few years later, Even, Goldreich and Lempel published their work, describing a different approach of OT and classifying the functionality most of OT researches are based on today [EGL85]. They used public key cryptography for their implementation.

Basically, OT deals with the following issue: let us assume that the sender holds a number of messages, in most cases represented as bit strings, while the receiver wants to obtain one or multiple of the sender's messages. Using an OT protocol for exchange, the sender does not get any information about which messages the receiver has picked, and the receiver learns no information about the messages that he did not choose. OT protocols are usually named after the number of messages and choices, for example if a sender holds n messages of which a receiver wishes to obtain k ($k < n$), we denote the protocol as k-out-of-n OT protocol. OT is very important for secure computation and is used by most SFE implementations during their execution. For Yao's GC (cf. Section 2.4), a 1-out-of-2 OT protocol is needed to exchange the encrypted keys for each parties' input. GMW, which is summarized in Section 2.6 uses a 1-out-of-4 OT protocol during its evaluation to handle AND gates within the circuit.

Due to its relevance, a lot of research has focused on OT optimization in the last years. Naor and Pinkas published several efficient OT protocols in 2001 [NP01], which are one of the best known approaches until now. Their protocol ensures security against a semi-honest adversary and focuses on applications, "which require two or more parties to perform many oblivious transfers of strings"[1], such as Yao's GC. A few years later, Naor and Pinkas described and compared different OT protocols and their improvements [NP05].

Most OT protocols are based on expensive public key cryptographic primitives, which limit their performance greatly. OT extension protocols [Bea96], [IKNP03] allow to overcome this bottleneck by extending few public key based OTs to many OTs using symmetric key cryptography only. More optimizations for [IKNP03] was published by Asharov, Lindell, Schneider and Zohner in 2013 [ALSZ13]. They also provide an open source implementation, which is partially based on [CHK+12] and [SZ13]. Again, the protocol is secure against semi-honest adversaries.

2.4 Yao's Garbled Circuits

Andrew C. Yao was the first researcher, who demonstrated that any polynomial function that can be represented as Boolean circuit can also be evaluated securely by introducing a secure computation protocol

[1] Naor, Pinkas; Page 1, Section 1.2 [NP01]

called Yao's Garbled Circuits (GC) [Yao86]. Yao's GC can only be performed by two parties and is one of the most established and efficient approaches for secure two-party computation until today. The idea behind Yao's GC is based on symmetric cryptography and the OT protocol.

To explain the concept of GC, let us look at an AND gate evaluation. Given a creator (Alice) with input $x \in \{0,1\}$ and an evaluator (Bob) with input $y \in \{0,1\}$, an AND gate with result $x \wedge y = z \in \{0,1\}$ should be calculated. For the first step, Alice generates six keys for any possible wire and a respective encrypted truth table for the AND gate (cf. Table 1).

x	y	z	encrypted results
0	0	0	$E_{k_{x=0}}(E_{k_{y=0}}(k_{z=0}))$
1	0	0	$E_{k_{x=1}}(E_{k_{y=0}}(k_{z=0}))$
0	1	0	$E_{k_{x=0}}(E_{k_{y=1}}(k_{z=0}))$
1	1	1	$E_{k_{x=1}}(E_{k_{y=1}}(k_{z=1}))$

Table 1: Yao's GC: encrypted truth table

Afterwards, she randomly permutes the table entries to make it impossible for Bob to know the entries because of their respective line. Then Alice transmits the encrypted results to Bob. Furthermore, she sends her corresponding key ($k_{x=0}$ or $k_{x=1}$) and both parties conduct a 1-out-of-2 OT protocol. Alice acts as sender (who provides $k_{y=0}$ and $k_{y=1}$ as inputs), and Bob (acting as receiver) picks the corresponding key to his own input ($k_{y=0}$ or $k_{y=1}$). Now Bob can decrypt the correct line of the truth table and obtain the correct key ($k_{z=0}$ or $k_{z=1}$).

Based on this concept, the entire Boolean circuit is evaluated iteratively. Alice encrypts all gates and holds their respective keys, but gets no information which ones were received by Bob. Bob only sees the encrypted results and has no chance to reconstruct Alice's input. To obtain the outputs, Bob can either send the resulting key back to Alice or Alice can provide a mapping from keys to bits. The protocol can be adapted to any circuits. Because all polynomial functions have a corresponding Boolean circuits representation, Yao's GC can be used as SFE technique for any polynomial function.

Several improvements for Yao's GC have been published over time. One of the most important is the technique for free XOR gates, that was explored by Kolesnikov and Schneider in 2011 [KS08]. Their work allows a better evaluation of XOR gates (garbling is no longer required). Another improvement has been published by Huang, Evans, Katz, and Malka [HEKM11], who demonstrate "several techniques for improving the running time and memory requirements of the garbled-circuit technique"[2]. Among others they improve the protocol's sequence. Prior implementations stipulated, that the creator first generates encrypted values for the entire circuit. After that the evaluator starts with his part of the protocol. Huang, Evans, Katz and Malka published an implementation, that allows the protocol to be executed in a pipelined fashion, where the circuit encryption and evaluation are performed in an interleaved fashion. After the first gate is encrypted, the evaluator can start. This allows the parties to evaluate even large circuits with billions of gates. Other improvements, like a garbled row reduction (3 instead of 4 table entries) have been published in 2009 [PSSW09]. In 2013, some developers published their work about fixed-key AES and how it can be used to improve Yao's GC [BHKR13]. Summarized Yao's GC is considered to be very efficient, because it only requires symmetric key primitives and is a constant round protocol. A comparison between Yao's GC and GMW is given in Section 3.1.

[2] Huang, Evans, Katz and Malka; Page 1, Abstract [HEKM11]

2.5 Multiplication Triples

A Multiplication Triple (MT) [Bea92] is a triple of bits (a, b, c) satisfying the following equation for two parties P_i with $i \in \{0, 1\}$:

$$c_0 \oplus c_1 = (a_0 \oplus a_1) \wedge (b_0 \oplus b_1),$$

with $c = c_0 \oplus c_1$, $b = b_0 \oplus b_1$ and $a = a_0 \oplus a_1$. As MTs are independent of their respective party's input, they can be generated at any time. They even can be retained before the process starts and then be delivered during the evaluation later. This strategy is used in many GMW (cf. Section 2.6) implementations, for example in [SZ13], [DSZ14], [ALSZ13]. To hide information about the processed values, it is necessary that each party only has information about their own MT values. P_i holds shares with index i and does not know $a_{1-i}, b_{1-i}, c_{1-i}$. Otherwise, an adversary can draw inferences about a party's secret values. Different approaches how two parties can jointly generate MTs are popular. One option is based on a 1-out-of-4 OT protocol.

MTs play an important role for GMW, specifically for its AND gate evaluation, that requires a MT for a secure mask generation during the process. For each AND gate, a complete new MT has to be calculated. GMW is described in the next section. Another technique for a more efficient MT generation based on random OT extensions and was proposed in 2013 [ALSZ13].

2.6 GMW

Another established approach for Boolean circuit evaluation is the Goldreich, Micali and Wigderson (GMW) protocol, that was described first in 1987 [GMW87]. Like Yao's GC, its security is mathematically proven and it is even usable for more than two parties. Each party has a mutually independent input, that is represented as bit string with certain length. The GMW protocol provides security against semi-honest adversaries. Assuming that a participant complies with the protocol rules, but attempts to obtain more information during the process, he will not be successful.

The GMW protocol can be partitioned into two phases: a setup phase and an online phase. The setup phase is responsible for the pre-calculation of all required components (for examples MTs), while the online phase takes care of the input sharing and the joint function evaluation. Since the setup phase is independent from both, the parties' inputs and the function itself and therefore can be executed at any time before the SFE, most developers decide to calculate as much as possible during this phase. As a consequence, the setup phase takes longer than the evaluation afterwards. The advantage of this practice is that the function evaluation can be done very fast once the function and its inputs are known and so the user gets the result in a short time. Two or more parties can participate in the GMW protocol. The protocol execution between two parties proceeds as follows: let us image two parties P_i with $i \in \{0, 1\}$, their respective inputs $x, y \in \{0, 1\}^n$ with length n bits and an arbitrary function over their inputs $f(x, y)$.

To protect everyone's privacy, all parties mask their inputs before the evaluation starts. Therefore, they generate a bit string randomly, perform an XOR operation on their respective random values and inputs and share the result. This step is shown in Figure 2. For any adversary, it is impossible to recalculate the origin inputs from the shared values. This step requires only one XOR operation, one random bit string generation and one communication step. Therefore it can be neglected in consideration of performance. Now each party knows the shared value and its own random string, which is used locally.

In the next step, the Boolean circuit is evaluated gate by gate. Depending on the gate type, both parties need to execute a number of different steps for each gate. The GMW protocol only deals with three different gate types, XOR, AND and NOT gates, but this is not a big problem, because all the other gate

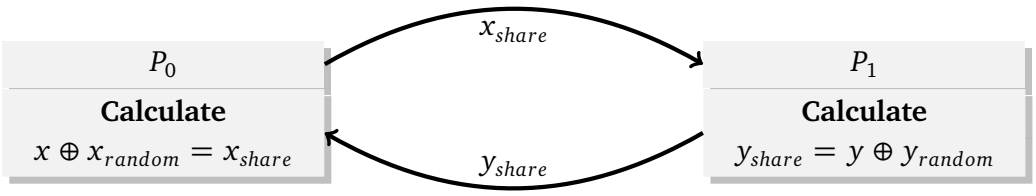

Figure 2: GMW Secret Input Sharing

types can easily be assembled by them. An OR gate for example is composed of: $a \vee b = a \oplus b \oplus (a \wedge b)$, NOR consequently with an additional NOT conjunction.

XOR Gates: XOR gates require only a simple execution, since XOR operations are associative and can be locally evaluated by each party. Because no communication and only one XOR operation is needed, XOR gates are not critical for performance.

NOT Gates: Similar to XOR gates, any NOT gate can be executed effortlessly during the process, because NOT can be described as calculation of XOR and the constant 1. $\neg a = 1 \oplus a$. Thereby its performance is uncritical as well (cf. XOR Gate).

AND Gates: AND gates need a more complex procedure than XOR gates, because they require interaction. Two different variants have been established for a successful evaluation, one of them uses MTs, the other uses a 1-out-of-4 OT. Since the MTs (cf. Section 2.5) are more efficient [SZ13], this approach is described in the following: for each AND gate, a MT is required. First of all, the MTs are used by both parties to calculate two new values (d and e):

$$d_i = x_i \oplus a_i \tag{2.1}$$
$$e_i = y_i \oplus b_i \tag{2.2}$$

d_i and e_i are transmitted to the other party with the result that each party can calculate d and e...

$$d = d_i \oplus d_{1-i} \tag{2.3}$$
$$e = e_i \oplus e_{1-i} \tag{2.4}$$

...and afterwards the output respectively:

$$z_0 = (d \wedge e) \oplus (d \wedge b_0) \oplus (e \wedge a_0) \oplus c_0 \tag{2.5}$$
$$z_1 = (b_1 \wedge d) \oplus (a_1 \wedge e) \oplus c_1 \tag{2.6}$$

AND gates and their evaluation are the bottleneck of GMW's performance. Whereas XOR and NOT gates need no communication step and thereby are uncritical to performance, AND gates require an interaction between both parties.

After all gates are evaluated, one last step has to be accomplished. The existing results z_i are shared and XORed again. Its output is the final function result.

The GMW protocol has been improved continuously in the last decade. For example, the OT execution as well as the MT generation can be performed in parallel to accelerate the procedure [CHK+12], [SZ13]. Parallelism is an important issue in any case, as a well-conceived circuit structure allows a division of all gates into different layers and enables a parallel evaluation within their respective layer. Correspondingly, a low circuit depth is preferable for the GMW approach. Another improvement for the protocol is the number of AND gates, which should be as small as possible. Various techniques can be used for this, most are summarized in [SZ13].

3 Related Work

3.1 Yao's Garbled Circuits vs. GMW

Yao's GC has been the first choice for secure computation for a long time. OT protocols were thought to be costly and GMW requires one round of interaction for each AND layer in the Boolean circuit. But the research of Schneider and Zohner [SZ13] has shown that with the aid of different optimizations, the GMW protocol can outperform Yao's GC under certain conditions.

Yao's GC uses a number of techniques for its procedure. One party has to encrypt the entire function during the evaluation. The larger the function is, the greater the effort. Fortunately, symmetric cryptographic operations can be used for the process, which are executable in an acceptable time nowadays. As a consequence, it is not that much critical for performance than prior Yao's GC implementations. Using other improvements, XOR gates can be evaluated totally "free" [KS08]. Beyond that, an OT protocol is required to exchange keys for the inputs. OT implementations have been improved a lot in the last years (cf. Section 2.3), for example by replacing prior public key procedures with symmetric cryptography. Finally, Yao's GC can be executed in a constant number of rounds.

On the other hand, for each AND gate that is located in the Boolean circuits, GMW requires an interactive OT execution. Using OT pre-computation, most of the workload of the OTs can be pre-computed in a setup phase. The use of MTs instead of OTs even further reduces the number of rounds in the online phase as well as the amount of data sent. As in Yao's GC, XOR gates (and therefore NOT gates as well) are performance uncritical, because they are "free". The main disadvantage of GMW is the number of communication rounds that the parties need to perform when they jointly evaluate the function. For each layer of AND gates, the parties have to perform interaction. As a consequence, the performance of GMW is very dependent on the network latency.

Some projects focus on comparison between both approaches. One of them has been published in 2012 [CHK+12] and describes how a GMW protocol with three or more participants can be implemented much faster than previous techniques. Further research was written by Schneider and Zohner in 2013 [SZ13]. They focus on different methods to improve GMW also for secure two-party computation and conclude that with their improvements "the GMW protocol is a noticeable alternative to previous protocols based on garbled circuits"[1].

3.2 Practical Secure Computation Projects

In the last years many projects were launched, which implement frameworks enabling the development of practical SFE applications that can be used with arbitrary functions.

Fairplay [MNPS04] was one of the first projects allowing to translate any function (written in a high level language) to a suitable circuit. A Yao's GC implementation was responsible for the circuit evaluation afterwards. Another project called FairplayMP supplemented Fairplay by extending the settings to allow more than two parties [BDNP08].

[1] Schneider and Zohner; Page 276, Section 1.1 [SZ13]

Two years later in 2010, a group of researchers introduced TASTY [HKS+10], a compiler to generate efficient secure two-party computation protocols from a high-level description. Compared to prior works, protocols that were generated with TASTY were based on Yao's GC, homomorphic encryption or a combination of both. So TASTY combined the benefits of both approaches. Homomorphic encryption is next to Yao's GC and GMW another approach for secure computation. Besides the generation, TASTY was also able to optimize circuits and execute and benchmark the protocols afterwards.

In 2011, the researchers Huang, Evans, Katz and Malka implemented a generic secure two-party computation framework [HEKM11], which is based on Yas's GC. Their work especially improved the scalability and allowed the evaluation of complex functions in practical applications. Many of their design decisions were based on Fairplay. In the same year, Malka published his work about VMCrypt [Mal11], a Java library supporting new algorithm for secure computation. To make its usage easier for other developers, VMCrypt provided an Application Programming Interface (API) for an easy integration in existing projects as well as a developer manual and different debugging and validation tools.

In 2012, a further modification of Fairplay was developed by Costantino, Martinelli, Sant and Amoruso [CMSA12]. Their project MobileFairPlay focused on the mobile area and provided a feasible security framework, that worked in the Android environment. Mood, Letaw and Butler also published their work in 2012 [MLB12] introducing a compiler for Android, that concentrated on the memory problems mobile phones have due to their limited resources. Their compiler generated circuits, that were compatible with Fairplay, but used further memory-efficient technique for the process. Another project in 2012, started by Kreuter, Shelat and Shen [KsS12], extended prior works about Yao's GC and provided a secure two-party computation system, which even guaranteed security against a malicious party. At the same time, the scalability was not affected much. Choi, Hwang, Katz, Malkin and Rubenstein [CHK+12] were among the first researchers, who developed a practical GMW implementation. Their work was based on Boolean circuits and allowed SFE for multiple parties in the presence of a semi-honest adversary.

One year later in 2013, three researchers introduced PICCO [ZSB13], another compiler for generating secure computation protocols from a high-level description, but with focus on distributed systems, like a cloud or similar environment. Henecka and Schneider published an improved implementation of Yao's GC in 2013 [HS13]. Their work provided protection against the semi-honest adversaries model and supported several optimizations enabling a better performance (10 time faster) and at the same time a lower memory consumption than prior works. Multi party computation is another major challenge for secure computation applications. Keller, Scholl and Smart presented a runtime environment in 2013 for executing secure programs using a multi-party computation protocol [KSS13]. Similar to PCF, they wrote a special compiler receiving a program description and translated it to bytecode. During the compilation, they used several techniques to optimize the bytecode and hence the subsequent evaluation.

In 2014, a group of researchers presented a new version of their prior work called CBMC-GC [FHK+14], an ANSI C compiler for secure computation. Its compilation is divided into various steps including the translation of a high-level description into an intermediate representation, loop unrolling, circuit generation and multiple techniques for the circuit optimization afterwards. CBMC-GC works with Boolean circuits only and supports the Yao's GC approach. Also this year, a different approach was published by Rastogi, Hammer and Hicks called WYSTERIA [RHH14]. Instead of writing the function in a known high level language and then compile it down into its respective circuit representation, WYSTERIA itself is a high-level functional programming language. Using WYSTERIA, developers can write functions that are interpreted dynamically at runtime and can be executed in an oblivious way. WYSTERIA supports secure multi-party computation and uses the GMW approach. An function example is given on their website[2].

[2] https://bitbucket.org/aseemr/wysteria/wiki/Home

4 Portable Circuit Format (PCF)

The Portable Circuit Format (PCF) project has been started by Ben Kreuter, Benjamin Mood, Kevin Butler and abhi shelat a few years ago to improve the way Boolean circuits for secure two-party computation can be created, stored and evaluated. First results were published in 2013 [KsMB13a].

As mentioned previously, many functions of practical interest require a huge Boolean circuit consisting of millions of gates, in order to ensure an oblivious process and thereby a secure way to protect sensitive data during the evaluation. Boolean circuits also have to be built correctly in a data-oblivious manner and as a consequence are usually created with software tools. PCF was started to solve two problems, that have been present in earlier works about secure computation. The first problem is usability: many related works focus on optimization and provide a good scalability, but expect specialized knowledge about their process in order to deal with them [HEKM11], [Mal11]. Developers have to get familiar with their practices before they can use them for their own programs. The second problem is scalability, since other approaches compile the function from a high level language and thereby do not expect any knowledge, but do not scale very well for complex or large functions [MNPS04], [KsS12], [MLB12]. Because of that, in practical applications most approaches are of limited use. To solve both, good scalability and usability, different techniques are applied in the PCF project.

The main task of the project is to develop an optimizing compiler allowing programmers to create and interpret Boolean circuits from a high level language in a compact and effective way, using two components, the PCF compiler and PCF interpreter. The programmer does not need any kind of specialized knowledge to use these components, he can simply write the desired function in his known high level language and afterwards passes it to the PCF compiler, which starts the procedure. Once the compilation is finished and the PCF interpreter has evaluated the Boolean circuit, the result is given back to the programmer. The entire process of PCF is shown in Figure 3 and is described in the following sections. In the course of this chapter, we also show many optimizations, which have been incorporated in both, the compiler and interpreter.

Because PCF itself is independent from its underlying secure computation techniques, the adversary model is determined by its employed protocols (the GMW or Yao's GC protocol for example).

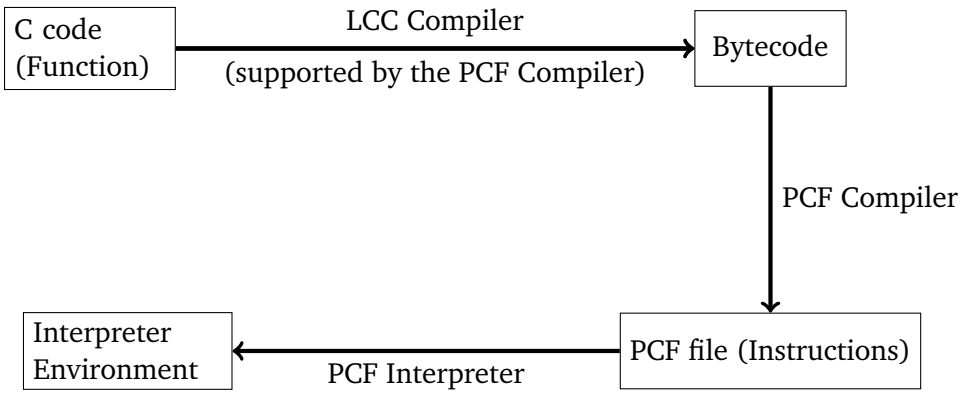

Figure 3: PCF process

4.1 PCF Compiler

The most important part of the PCF project is a special compiler, implemented in Common Lisp. The compiler allows programmers to develop their desired function in a high level language and then compiles it down into a PCF file describing the Boolean circuit representation of the function as a sequence of instructions.

The compilation process is divided into two steps. We assume that an user wants to evaluate a function in an oblivious way and has already written the function in his known program language. In the first step, the source code of his function needs to be translated into bytecode (also called intermediate representation or IR), a machine independent representation of the function, which can be used for multiple target architectures. For testing purpose, the PCF developers have worked with the Little C Compiler (LCC) as front-end to translate any function into bytecode. Written by Chris Fraser and David Hanson during the 90s, LCC is a well-documented C compiler known for its simplicity[1]. But the PCF compiler is not restricted to LCC. In the long term multiple programming language and its respective compilers should be supported as long as the compiled bytecode is compatible. As next step, the PCF compiler starts its compilation process, including not only optimization but also the translation of the bytecode into a Boolean circuits describing format, stored as instructions within a PCF file. These instructions describe the architecture of the Boolean circuit representation and can be jointly evaluated by both parties now. A detailed description of each instruction is given in Section 5.2 for PCF 1 and in Section 6.3 for PCF 2.

4.2 PCF Interpreter

After the function has been compiled, the PCF file can be used for SFE by both parties. An interpreter, the second component of the PCF project is responsible for this part. For every programming language the developer wants to use to read the PCF file, an interpreter has to be implemented, which translates all instructions into commands of the respective secure computation protocol. This happens at runtime. Software programmers only need to link the interpreter in their application, deliver the desired input and the function can be evaluated in a privacy-preserving way.

4.3 Advantages

The PCF project allows language independence, that enables developers to design software dealing with secure computation in their known language. After the function is given in a high level language (such as C), the PCF compiler is able to translate it to a respective secure computation protocol, which the PCF interpreter can execute. No specialized knowledge about the architecture of circuits or its creation process is necessary any more, all these are abstracted. Also PCF supports a more natural programming-like syntax, which is easier to handle than designing a Boolean circuit. To simplify the process further, PCF can easily process non-secret information, which allows interleaving secure computation and the external program. All this could be a chance to make secure computation usable for a bigger audience and assist in its dissemination.

A significant part of the compiling process is optimization in order to accomplish the conduct of a compact and thereby efficient PCF file . PCF instructions can be thought of as a program, that creates Boolean circuits at runtime. All required components are supplied by PCF for this purpose. It allows the creation of methods by using function declaration with return and call statements and supports branches as well as a program pointer. Wires are stored in an array, which can be addressed easily.

[1] https://sites.google.com/site/lccretargetablecompiler/

The Boolean circuit description for a function is evaluated at runtime. The PCF interpreter executes the instructions while only needing a fixed-size and hence constant memory for them. A comparison of different sizes can be found in a presentation, the PCF developer published in 2013 [KsMB13b]. To reduce the required memory for storing the circuit description, PCF uses several techniques. One technique is loop description. As it is unnecessary to unrole loops at compile time, the PCF interpreter takes care of an effective loop evaluation at runtime. Based on the possibility to create subroutines or gadgets (a branch containing instructions) and call them from any placement in the PCF file, recurring circuits (like a loop realization) can be stored in a space-efficient manner. For that reason PCF has a very good scalability and can deal with big and complex functions in acceptable memory resources. Basically loops can be executed under only one condition: their termination condition must not depend on secret values. A secret value (or secret-shared value) depends on the input of one party. Otherwise it involves the risk to leak sensitive information from the control flow. Because PCF can handle the processing of both, secret-shared values and plain values, this is not a problem. Additional restrictions like upper bounds are not imposed by PCF itself. The programmer has to take responsibility for that. Another optimization is address organization. Single gate IDs are addressed in memory and can simply be overwritten if they are no longer needed. All compiler optimizations are based on the bytecode and hence are machine independent. According to the developers, with PCF "the bottleneck in secure computation lies with the cryptographic primitives, not the compilation or storage of circuits."[2].

Another advantage of PCF is that it is independent of the underlying secure computation protocol, as long as it uses Boolean circuits to represent a functionality. Hence, PCF can be evaluated using, for instance, Yao's GC or the GMW protocol by simply writing a respective interpreter for it.

4.4 Versioning

Development about PCF enhances continually and so PCF version 2 has been released meanwhile. The concept has not changed, but both versions differ in many details and thereby the PCF compiler and interpreter are not compatible in any direction. Not only the syntax design changes in PCF 2, which make an execution with the old PCF interpreter impossible, but also some instruction types are altered or completely replaced with different ones. By now for PCF 2 a working interpreter for C++ exists, while PCF 1 supports interpreters in both, C++ and Java high level languages. PCF 1 and PCF 2 compilers are written in Common Lisp and both need bytecode for their compilation.

[2] Kreuter, Shelat, Mood and Butler; Page 1, Abstract [KsMB13a]

5 PCF 1 - Java Interpreter

PCF is a promising work to deal with performance issues due to the processing power, secure computation needs. With regard to Java, the Android environment may be an important area as well. Smartphones and Android become more and more popular and allow everyone to store sensible private data on a device able to exchange information in many different ways, sometimes even out of users' control. Secure computation in a mobile environment is a young subfield, but much progress could be achieved so far. Examples are given in [Dem13], [DSZ14], where the developers provide a practical realization of general secure two-party computation using GMW in a mobile environment and a trusted hardware token to enhance performance. This token is used to pre-generate data to reduce the processing power and memory required during evaluation. Another example is the MightBeEvil[1] project [HCE11], that focuses on Android Apps for secure computation as well.

To enable PCF for an Android application, an interpreter is needed to translate PCF instructions into commands of a secure computation protocol. This chapter introduces the PCF 1 interpreter for Java, its working methods and program structure. The interpreter was written by the PCF developers. In Section 5.2, we present our reference for many PCF 1 instructions. As next step, we add our newly written GMW implementation as additional possibility to evaluate Boolean gates (Section 5.3). At last, we measure the runtime of both approaches (GMW and Yao's GC), compare and interpret them in a benchmark test (Section 5.4).

5.1 Java Interpreter for PCF 1

The PCF developers implemented a working Yao's GC interpreter for the PCF 1 format in Java, which reads and executes PCF instructions at runtime. This interpreter has been written to demonstrate the basic PCF functionality and has never been deployed in a real practical environment so far. Unfortunately, although the mobile area is a significant part to disseminate secure computation, the PCF developers decided to focus on other issues and stopped working on a fully operational Java implementation. The current implementation can only handle PCF 1 instructions. Plans for an update to PCF 2 are missing. Nevertheless, the base code is functional and can be used to experiment with multiple improvements, which we describe in the next sections.

5.1.1 Implementation

The PCF interpreter is written as fully independent application, that takes care not only of all PCF 1 instructions, but also of all required communication issues. Without modification, the consequences being that an extra network channel has to be established, limited to PCF communication only. For the function evaluation the PCF file, including all instructions, is read first. All contained instructions are stored as an object into a HashMap. Then the instructions are executed gradually.

To jointly compute a function, a lot of communication between both parties is required. Java provides many ways for two parties to communicate, most of them are supported by different streams. For the PCF interpreter, an ObjectStream[2] is used to exchange data, mostly BigInteger objects.

[1] `http://www.mightbeevil.com`

[2] `http://docs.oracle.com/javase/tutorial/essential/io/objectstreams.html`

In order to optimize the interpreter, the chosen data type has to be considered, too. The BigInteger class is used to store the parties' input and processes them. BigInteger is a reasonable choice, because it can deal with large numbers on the one hand. On the other hand it supplies the possibility to set a single bit and tests if a single bit is set. Another suitable data type is BitSet that is provided by Java as well. BitSet is highly efficient on bit operations, but unfortunately it does not supply many arithmetic methods by default, which are partly used by the origin PCF interpreter. Furthermore, a mix of both data types is conceivable to benefit from all advantages. But as consequence we have to convert between both data types on several places, which can cause performance issues, too. For the PCF interpreter, BigInteger was chosen as sole data type, especially because bit operations can be optimized manually, if these methods expose as performance bottleneck.

5.1.2 Program Structure

The structure of the PCF program is shown in Figure 4. In its code the interpreter unites both: the server and client implementation. The basic program is constituted by four classes, that initialize and maintain fundamental parts of the interpreter. All these program parts are not responsible for the secure computation protocol itself, they only provide underlying conditions, like network aspects.

- **Program:** abstract class, defines methods that have to be implemented by both, client and server. Also determines program flow.
- **ProgramServer:** implements fundamental server services, like socket creation/listening and OT initialization. Inherits from Program.
- **ProgramClient:** implements fundamental client services, like socket connecting and OT initialization. Inherits from Program.
- **ProgramCommon:** creates common objects for client and server, like ObjectStreams for communication issues or the circuit object.

These four classes implement all required methods for the SFE.

- **InterpreterCommon:** provides methods to manage the common circuits, which should be jointly evaluated by both parties.
- **InterpreterServer:** implements all methods from Programm class. The class is responsible for the server side SFE. Inherits from ProgramServer.
- **InterpreterClient:** implements all methods from Programm class. The class is responsible for the client side SFE. Inherits from ProgramClient.
- **Interpreter:** provides methods to read the PCF file and interprets and executes all instruction types.

Our experiment to use the GMW approach for circuit evaluation instead of Yao‘s GC will be described in the next section. Performance tests, that demonstrate GMW advantages under certain circumstances are included.

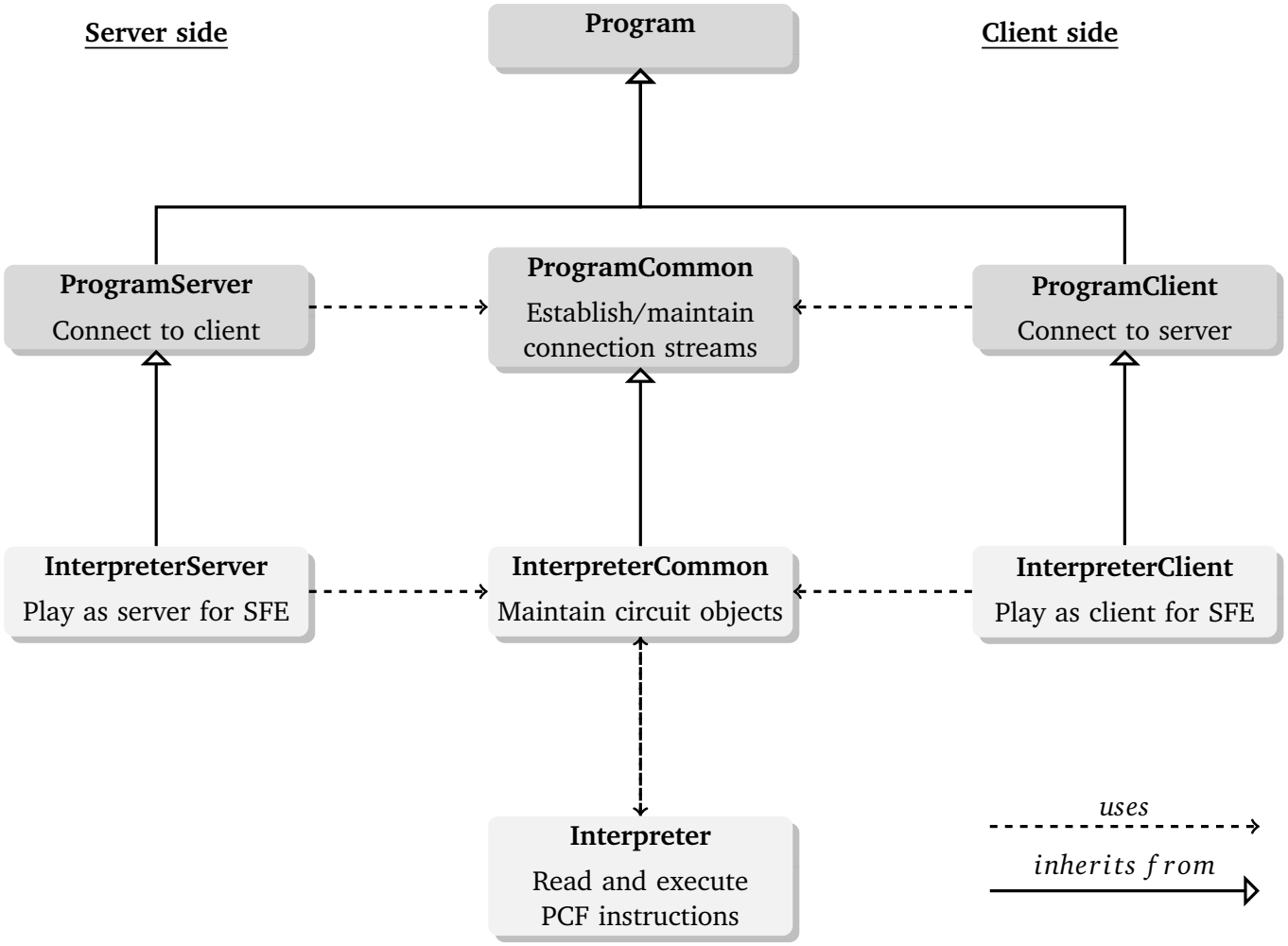

Figure 4: PCF interpreter, program structure

5.2 PCF 1 Instructions

PCF supports different instructions to describe and build a Boolean circuit. All instructions are stored in a file and are read in and interpreted at runtime. This section gives an overview of the most important instructions, all are compatible with any PCF 1 interpreter. They do not work out of the box in a PCF 2 environment. Unfortunately, no reference for all instructions exists yet, as the original paper only gives a short overview [KsMB13a]. Furthermore, instructions of PCF 1 have changed repeatedly during their development. New instructions were introduced or old ones were extended by new parameters. The following list is not complete. Its focus is in the most important PCF instructions, which appear in many PCF files.

Each instruction takes some additional parameters, which are required for a successful execution. Optional parameters do not occur in PCF, all of them have to be set. Some instructions have a label for identification purpose, for example to address the destination for jumps. All these labels have to be unique within the file. To understand the process, it is important to know, that within the circuit each wire is assigned to one Boolean value. Additionally, a flag determines, if the value depends on the inputs. A value that depends on the inputs is called *secret value*. This is important, because some functionality (like a loop condition) can only be used with so called *non-secret values*, which are independent from the inputs and thereby do not leak any information about them.

PCF allows two kind of instruction groups: functions and gadgets. A function can be used to realize a subroutine, which executes instructions and then returns to the caller. The program can call a function at any time without conditions. A gadget defines a branch start, which can also be jumped to at any time, but allows a second parameter as condition, deciding if the jump will be executed or not.

A PCF file example is shown in Figure 5. The function represents a two bit adder that takes two inputs and simply adds them. Line 1-6 define the program header. A constant 0 is set in line 7. Afterwards, the program defines and calls the function `MAIN` to start the calculation in line 9. The function reads Alice's and Bob's input (line 10-15) and uses some XOR and AND operations to calculate the output, which is shifted out in line 24 and 25. The `MAIN` function returns in the last line. A bigger example (a 32 bit adder) is shown in Appendix A. All instructions are described in detail as next step of this section.

```
 1 GADGET: MAIN
 2 CLABEL ALICEINLENGTH 2
 3 CLABEL BOBINLENGTH 2
 4 CLABEL xxx 32
 5 SETLABELC ALICEINLENGTH 2
 6 SETLABELC BOBINLENGTH 2
 7 0000 64 0 0
 8 FUNC main xxx
 9 FUNCTION: main
10 ALICEINPUT32 0 0
11 0110 65 0 64
12 0110 66 1 64
13 BOBINPUT32 32 0
14 0110 67 0 64
15 0110 68 1 64
16 0110 69 65 67
17 0110 70 69 64
18 0110 71 65 67
19 0110 72 65 64
20 0001 73 71 72
21 0110 74 65 73
22 0110 75 66 68
23 0110 76 75 74
24 SHIFT OUT ALICE 70
25 SHIFT OUT ALICE 76
26 RETURN -1 xxx
```

Figure 5: PCF 1 file example

FUNCTION: The instruction FUNCTION defines the starting point of a subroutine. It takes one parameter, the subroutine's label. Unique labels enable a simple way to identify a subroutine making an easy call possible. A RETURN statement denotes the subroutine's end.

Example: `FUNCTION: main`
Defines a subroutine with label "main". Until the RETURN instruction is reached, all following instructions are part of the subroutine.

FUNC Calls a subroutine. Requires two parameters, the label of the called subroutine and a return label that is required for the RETURN statement. After a subroutine is called, the instruction pointer is set to it and executes the following instructions.

Example: `FUNC main xxx`
Calls subroutine "main" and defines the return label "xxx".

RETURN Marks the returning statement of a subroutine. The first parameter is a wire ID, because the condition for a successful returning is that the given wire has to be assigned to 1. Otherwise the return is invalid and will not be executed. This wire has to be independent from the input (non-secret value). An alternative for that parameter is -1, which skips the check. The second parameter defines the label to return to.

Example: `RETURN -1 xxx`
Finishes a subroutine and (thanks to -1) jumps back to the caller (FUNC with return label "xxx") without checking.

GADGET: Denotes the starting point of a new branch. After jumping to a branch, the instruction pointer is set to it and executes the following instructions. Takes one parameter, the branch label.

Example: `GADGET: PC1`
Starts a new branch, called "PC1".

BRANCH Jumps to a branch that is defined by the GADGET instruction. Two parameters are required, the desired branch's label and a wire ID, whose value has to be a non-secret value and determines if the jump is executed. The BRANCH instruction cannot be used to call a subroutine (cf. `FUNCTION`).

Example: `BRANCH PC1 33`
Jumps to branch "PC1", if the wire with ID 33 is not assigned to a secret value (determined by the flag) and its value is 0.

PUSH The PCF interpreter provides some kind of a stack system, that allows to define a copy of a given wire as its successor. The consequence being that independently from any changes of the original wire, the copy is untouched and can be popped later to the same position. The instruction takes only one parameter, the wire ID that should be pushed.

Example: `PUSH 33`
Pushes the wire with ID 33 onto the stack, or more specifically defines (a copy of) the wire 33 as successor of itself.

POP Pops the wire back. The ID remains constant during the process. Takes only one parameter, the wire ID that should be popped from the stack.

Example: `POP 33`
Pops wire with ID 33 back, by overwriting the successor of the given wire to its predecessor. If no successor exist, the program stops.

CLABEL Appears only in the PCF header and is used for general information about the inputs. The instruction creates a label and determines the bit width for each parties' inputs. Takes two parameters, the label name and the number of wires, which should be reserved for the label.

Example: `CLABEL BOBINLENGTH 32`
Creates a new label BOBINLENGTH and allocates 32 wires (and thereby 32 bit) to it.

SETLABELC Another setting for the PCF header. Sets the length of each parties' input. The first parameter defines a label, that has to be created before (cf. CLABEL).

Example: `SETLABELC ALICEINLENGTH 32`
A length of 32 bit is supported for the input by this PCF file.

ALICEINPUT32 and **BOBINPUT32** Fetches 32 bits of one party's input bits and stores them in a given wire. If an input larger than 32 bit is allowed, the first parameter decides, which area of the input is taken.

Example: `ALICEINPUT32 33 0`
Stores 32 bit of Alice's input to 32 wires starting with wire 0. The range of Alice's input reading in is defined by the value of wire 33.

SHIFT OUT Outputs the value of a wire (one bit) for Alice or Bob. The first parameter is hard-coded ("ALICE" or "BOB"), no other values are allowed.

Example: `SHIFT OUT ALICE 33`
Output the content of wire 33 to "ALICE".

STORECONSTPTR The interpreter stores an additional table holding pointers to certain wires. This instruction can be used to set such a pointer.

Example: `STORECONSTPTR 2 33`
Stores a pointer to wire with ID 33 on table index 2

OFFSETPTR Starts at a given wire ID (second parameter) and checks if the next 32 wires are assigned to non-secret values. Only if all of them are independent from the inputs, the instruction calculates a new 32 bit value and copies each bit from one of the 32 wires' values sequentially. Afterwards the result is added to a pointer.

Example: `OFFSETPTR 33 34`
Starts calculation at wire 34 and adds the index to pointer 33.

CPY121 Copies a wire value. The three parameters determine the source wire, an additional offset and the target wire.

Example: `CPY121 2 33 34`
Copies the value of wire 33 to the target wire with ID 34 + the ID of wire on pointer table index 2.

CPY32 Copies a wire value. The three parameters determine the source wire pointer, an additional offset and the target wire.

Example: `CPY32 2 33 34`
Copies the value of wire with ID 34 + the ID of wire on pointer table index 33 to the target wire 2.

5.3 Implementation of a GMW PCF 1 Interpreter

The PCF interpreter for Java can currently evaluate a function in two different ways. The first option supports plaintext for both, each parties' inputs and the gate evaluation. Using plaintext values is for testing and experimental purpose only and should not be used in a practical working environment.

In order to securely process values, the PCF interpreter implements Yao's GC. The approach of Yao's GC is described in Section 2.4. But as mentioned in Section 3.1, in some cases GMW can be a better choice to achieve an efficient and fast SFE. To compare both approaches, the interpreter needs to be extended by a GMW implementation. In Section 2.6 the GMW approach is described, the following section is dealing with our practical implementation in Java.

Different changes have to be made to extend the interpreter. First of all, the program flow has to allow a more flexible execution. Unfortunately, the Yao's GC implementation was an integral part of the interpreter and thereby was firmly attached to it. The responsible code for Yao's GC protocol had to be extracted and provided in an exchangeable way instead. As of now, the program sequence enables an easy way to switch between both approaches. For the next step, all GMW components have to be implemented, including MT generation and the two GMW phases (setup and online phase). The data type does not need any changes, since BigInteger provides everything that is required for the GMW protocol. Also all communication methods are used equally for GMW.

During the setup phase, the interpreter generates enough MTs to provide them later. The OT protocol is used for these calculations, but it is modified by some changes: instead of an usual OT execution, the sender's output are two random messages. This approach is called Random-OT (R-OT) and decreases the communication complexity. More information about this technique can be found in [ALSZ13]. All MT's values (a, b, c) are stored as BigInteger objects, since a_i, b_i and c_i define one MT on position $i \in \{0, 1, ..., n-1\}$ within three BigIntegers with n bit length.

5.3.1 Online Phase - Evaluation

The online phase starts by reading the inputs of each party. In contrast to Yao's GC, GMW is based on masked inputs that have to be generated by each party. The SecureRandom class[3] supplies secure methods to create random values for a BigInteger object with the same length as the respective inputs. After their generation, the masked inputs are shared between the parties.

Afterwards, the gate evaluation routine of PCF needs to be replaced. An opcode is used to differentiate between gate types, which makes it easy to adapt the process for GMW. If an AND, XOR or NOT opcode is detected, the values are evaluated locally (XOR, NOT) or interactively (AND). Every other gate type has to be translated into a combination of XOR and AND gates.

For each AND gate evaluation, a MT is required as well as an interaction step. The circuit has the worst-case depth, as the sequential execution of the instructions impedes the structure of layers, which could be implemented in parallel and hence merge the interaction of various AND gates within the layer. Before the program stops, the outputs are recombined to reconstruct the final result, that can be calculated by an simple XOR operation on both shares.

[3] http://docs.oracle.com/javase/7/docs/api/java/security/SecureRandom.html

5.4 Benchmark GMW vs. Yao

In this section, we perform a detailed benchmark of the plaintext, Yao's GC, and GMW backend of the PCF interpreter. The plaintext and Yao's GC backend were taken from the original PCF implementation and the GMW backend was implemented in the course of this thesis (cf. Section 5.3). Before the benchmark is run, some presumptions about the test results are described.

All backends can be divided into two parts: an initialization and the circuit evaluation. GMW's initialization contains the setup phase, where all required MTs are generated as well as the PCF instructions read in. As mentioned previously (cf. Section 2.6), the calculation for MTs is critical for performance and hence this phase should take much longer than the circuit evaluation.

Yao's GC also uses an initialization phase to prepare some steps that are performance critical. An OT protocol is needed, because the client has to obtain its inputs to the garbled circuit in an oblivious way, before the process starts. The circuit has to be prepared, too. During the evaluation, many symmetric key operations are performed for the oblivious function evaluation.

The process using plaintext is straightforward. Both parties know all values including all inputs. During the initialization phase, the PCF file is read in and the instructions are stored. Afterwards the gates are evaluated using the plaintext values. No communication is needed, as well as no kind of cryptography.

Comparing all backends, it is important to know, that the circuits are not optimized for GMW in any way. Neither the number of AND gates are reduced, nor do we implement any depth-optimized circuit variants (e.g., from [SZ13]). None of the approaches' implementation supports a parallel evaluation. However, as all functions are not very complex, the number of AND gates should be small inherently. More information about the functions are presented in the next section. The GMW initialization phase should take longer than the Yao's GC, because the MT generation needs one OT protocol for each triple. The circuit evaluation is expected to be faster with GMW.

5.4.1 Environment

All benchmarks were performed on a notebook with the following system specifications:

- `Processor:` Intel® Core™ i5-4200U CPU @ 1.60GHz × 4
- `Memory:` 8GB (1x 8192MB) DDR3L-1600
- `Operating System:` Debian GNU/Linux 8 (Jessie) 64-bit
- `Java Runtime:` openjdk-7-jre (Version: 7u51-2.4.6-1)

As both parties were simulated from the same notebook during the tests, possible delays within a network are not part of this test and can not be considered. Especially the GMW backend benefits from this settings.

Unfortunately, only a few PCF files were available to compare. As mentioned previously, a compiler for files compatible with PCF 1 instructions is no longer actively developed. Almost all files that were provided by the PCF developers were defect (most of them irreparable) and thereby could not be used. Even the compiler itself was not working correctly at its last version. So it was unfortunately not possible to create new test examples for this benchmark. As PCF files are composed of many instructions even for small functions, it was also not possible to completely create a file manually, especially for a bigger example. To be able to use some PCF examples anyway, two of the smaller existing PCF files have been

partially reworked. As a consequence, all test results can not be used to compare the PCF project with related works directly. They only make a statement about the performance of GMW and Yao's GC within the PCF interpreter. For the sake of completeness, test results about an (insecure) evaluation using plaintext values are included as well.

Two functions were measured and evaluated for this benchmark test: the first function is a simple adder, that calculates the sum of two 32 bit inputs. In Appendix A, we provide its entire program code. The function is related to our example (cf. Figure 5). The second function is a 32 bit comparator that solves the well known "Yao's Millionaires' Problem" [Yao82] and is often used as an introductory example for secure computation: two millionaires want to know which one is richer without mutually disclosing their wealth. It compares both inputs and returns 1 if the client party enters a higher number than the server party, otherwise 0 is returned. Both functions can handle inputs up to 32 bits, different inputs had no appreciable effects. The adder function contains 31 AND gates and 221 XOR gates, the millionaire function consists of 164 AND gates and 1352 XOR gates. The number of AND gates for the millionaire function could be further reduced by multiple optimizations (for example published at [KSS09]), but for this benchmark we do not focus on circuit optimizations beyond the default PCF ones. Since no functions are optimized for GMW in any way, the circuit depth is equal to the number of gates (252 for the adder function and 1516 for the millionaire function). All instructions are evaluated sequentially and therefore the number of AND gates defines how many communication rounds are required for the GMW backend.

5.4.2 Benchmarks

All measurements were determined by the client and were averaged over 100 executives to avoid measuring errors. They can be found in Table 2 as well as Figure 6.

	Adder 32 bit			**Millionaire 32 bit**		
	Yao's GC	GMW	Plaintext	Yao's GC	GMW	Plaintext
Initialization	2751 ms	3236 ms	1023 ms	2279 ms	3298 ms	508 ms
Circuit evaluation	1246 ms	80 ms	38 ms	1986 ms	104 ms	53 ms
Total	3997 ms	3306 ms	1061 ms	4265 ms	3402 ms	561 ms

Table 2: Benchmark, Adder 32 bit and Millionaire 32 bit

An overview of the time spent in each phase for the three PCF backends is shown in Figure 7. GMW's initialization step is divided into two parts: the PCF instruction reading and the GMW shared phase (including MT generation, cf. Section 2.6). During the circuit evaluation all instructions are executed, the gate evaluation is similar to GMW's online phase. Yao's GC initialization step is also divided: the PCF instructions have to be read and the OT protocol needs to be executed for the inputs. Similar to GMW, the circuit evaluation is shown as single step, a division is not necessary.

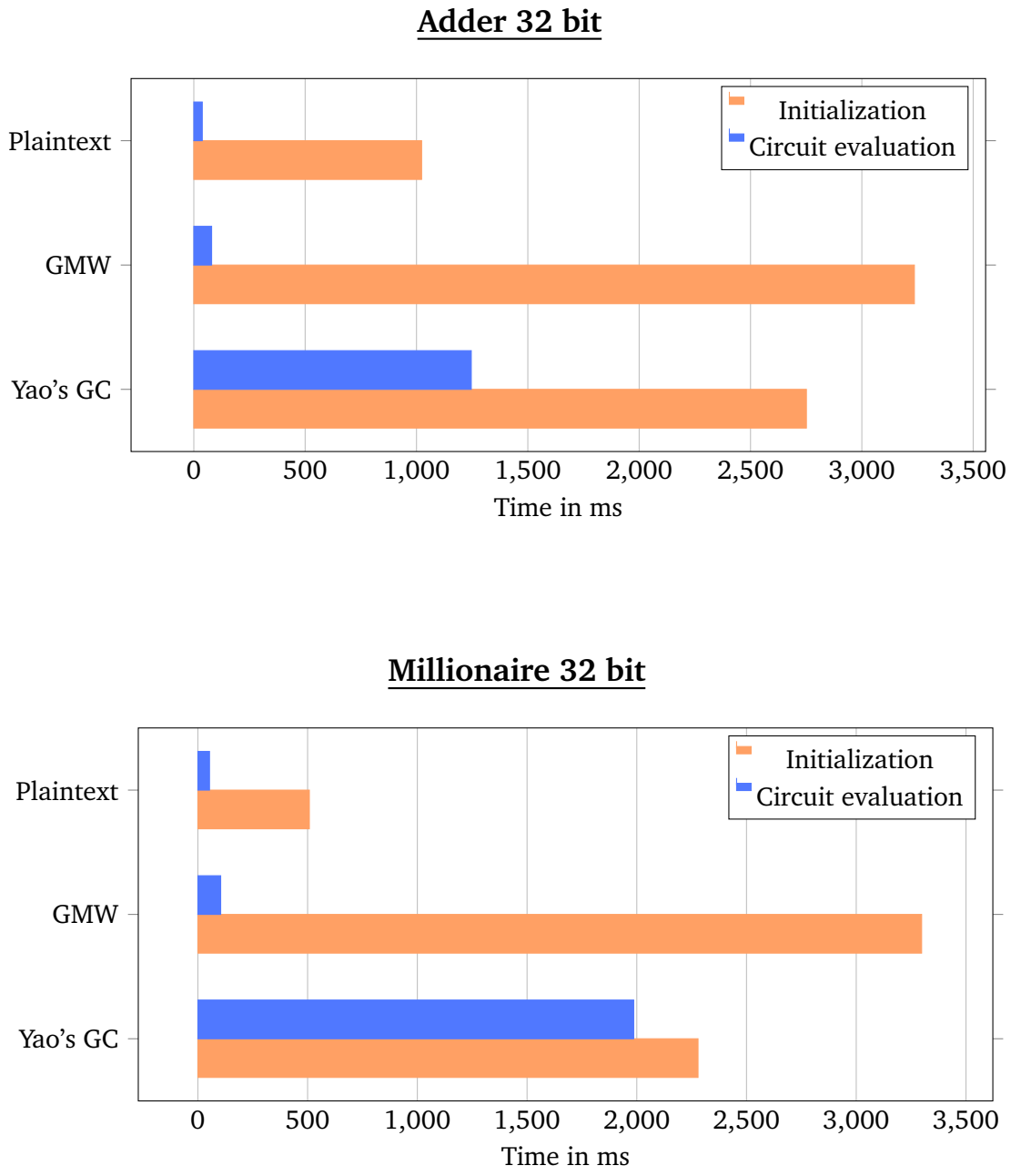

Figure 6: Time distributions for both functions

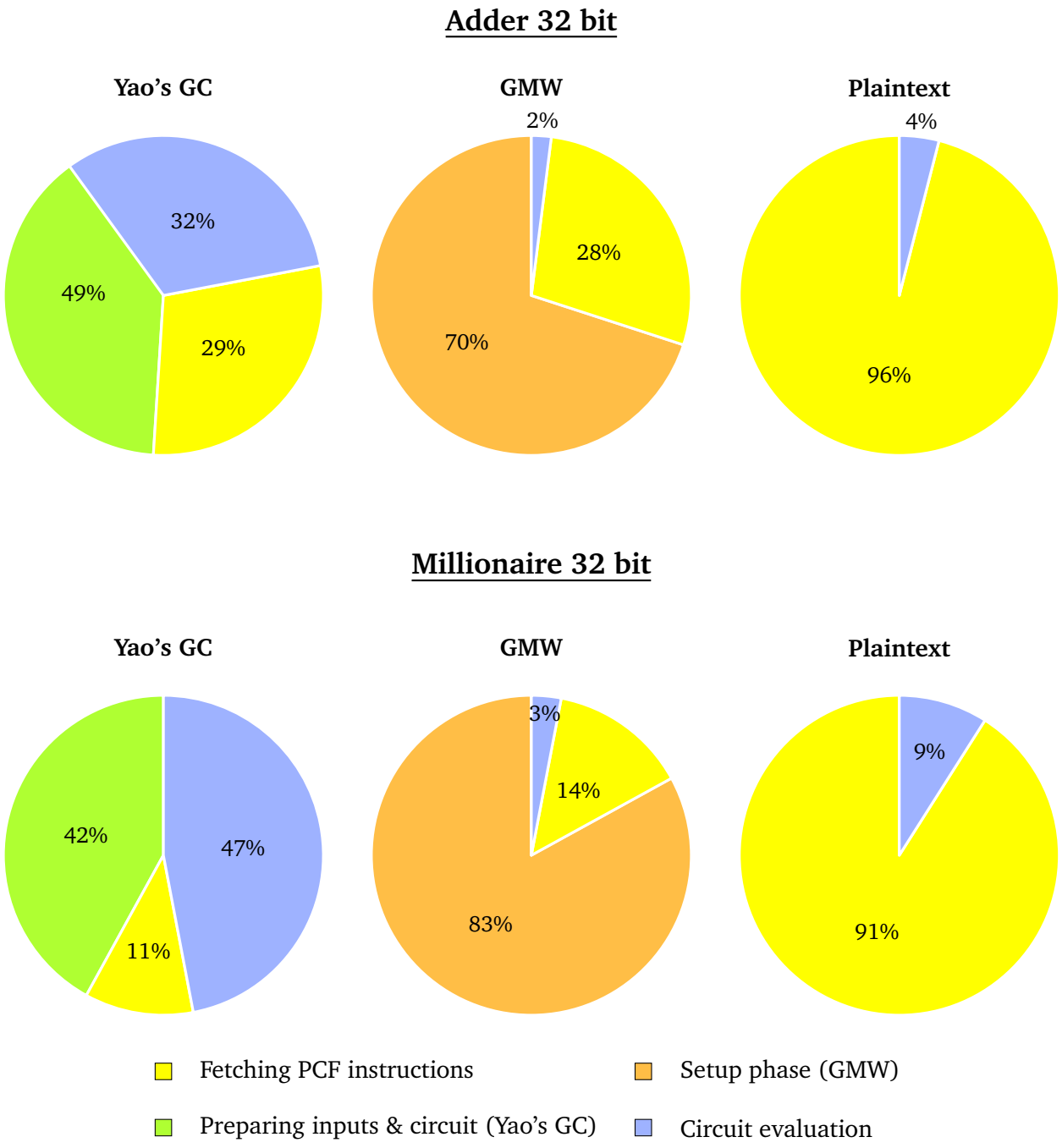

Figure 7: Time distributions for each PCF backend

5.4.3 Evaluation

The benchmark supports the claims of related works, for example [SZ13]. GMW's slowest part contains the initialization, which includes the MT generation and thereby many OT executions. After that, the circuit evaluation can be executed much faster. Yao's GC times for both, initialization and circuit evaluation, are closer together. The procedure with plaintext provides the fastest execution (cf. Table 2).

Initialization The results are shown in detail in Figure 7. For both functions, GMW needs the longest time to execute the initialization phase. That was to be expected and is explained by the MT generation. The Adder function requires only a few AND gates and thereby only a few MTs, since the addition calculation is not very complex and the PCF file was optimized in the following way: during the compilation the PCF compiler adds some `GATES` instructions that copy the input values from one wire to another several times. We assume that the compiler has specific ID areas for specific purposes and uses them independently from any function optimizations. For this benchmark, we remove these instructions, modify the wire IDs that are used during the calculation and start the calculation directly after all inputs are read in without any copying. But although the Adder PCF file is smaller than the Millionaires one (296 against 5877 PCF instructions), the time to read in all instructions takes longer for the Adder function. Further tests showed, that this behavior is independent from the kind of function and can be confirmed even after 100 runs. The reading time takes longer, but less MTs are required, therefore the entire execution time for the initialization phase is similar. Yao's GC initialization phase should be about the same time for the Adder and Millionaire function, because the inputs for the two functions are equal and the functions themselves contain only a simple calculation. The difference of 472 ms could be explained in turn by the time difference between both instruction readings. The execution with plaintext values involves only the instruction reading in the initialization phase. Again, the time difference can be explained by the same reason as with Yao's GC.

Circuit evaluation During the circuit evaluation, GMW's advantages comes to the fore. As a consequence of previous MT generation, the evaluation is almost as fast as a plaintext evaluation. The small difference can be explained by the communication step that every AND gate requires. Nevertheless it is much faster than Yao's GC. A comparison between the Adder and Millionaire evaluation shows in all three backends, that much more instructions have to be executed for the Millionaire function. This is mainly because the Adder file was optimized manually as contrasted with the Millionaire's one.

Altogether GMW is slightly faster than Yao's GC for both functions. Its slower initialization phase compensates completely during the evaluation. But GMW also benefits most from the test environment, as the two parties have been simulated with the same notebook and hence network latency was not an issue. Otherwise, many further optimization for GMW could be implemented in future works, which we outline in the next section.

5.5 PCF 1 and GMW - Summary and Discussion

In this chapter, the PCF interpreter was introduced together with a reference about all important instructions available for PCF 1. Afterwards an extension was presented to provide GMW as new backend for the interpreter. The result of our benchmarks were shown and interpreted in the last section. They confirmed previous works and demonstrated that GMW could be a noticeable alternative to the existing Yao's GC backend. In this section, we discuss general advantages and disadvantages of a GMW integration within the PCF environment.

First of all, the implementation of GMW's gate evaluation is comparatively easy to realize and does not require as much code as the existing Yao's GC part. Due to the replacement, the number of Java classes and lines of code could be decreased a lot. After the Yao's GC part was separated from the interpreter's core, the GMW integration was uncomplicated.

Other advantages can be applied from existing benefits of the GMW approach. Most of the work can be done during the setup phase, which allows a more flexible resource arrangement. This can be useful for various applications and scenarios, the PCF project wants to support. Especially for mobile devices, such as the examples of [Dem13], [DSZ14] it may further improve the performance.

The main disadvantages are based on the fact that GMW has never been in the focus of the PCF developers, who have only focused on the Yao's GC approach. Because of that many things are not implemented in a way, GMW could use them for an optimal performance.

One example is the PCF interpreter and its execution sequence. In its current stage, the interpreter executes all instructions in a sequential order, whereby the circuit construction (which is built from the instructions) only allows a sequential gate evaluation as well. One of GMW's greatest optimization is related to depth-efficient circuits, which enables a parallel evaluation and hence avoids an interaction step for each AND gate. Therefore, a low circuit depth is preferable for the GMW approach. The depth of circuits built with the PCF components is similar to its size and that is the worst-case scenario for GMW. A parallel gate evaluation would allow further improvements. Sending and receiving data for every single AND gate implies not only the required parameters but also network overhead for each connection. In case of a parallel evaluation, the interpreter would have to communicate with the other party only once per layer. Of course more data would be transferred in one connection, but the delay issues - summed up by each connection - could be reduced.

Another issue is about the type of gates, the interpreter has to evaluate. PCF instructions are not limited in their gate depiction. Since GMW can only handle AND, XOR and NOR gates, the interpreter has to transfer all instructions containing any other gate type into a respective presentation of these three types. Also no techniques to decrease the number of AND gates are involved in the compilation process.

But all these issues are a consequence of PCF's design and conception, that are based on its commitment to Yao's GC. The interpreter and compiler could be expanded in order to eliminate these disadvantages. A support for parallel execution might be an improvement for Yao's GC, too.

5.6 PCF 1 - Future Work

Several steps can be done to continue the work on the PCF project and its GMW extension as an additional backend. Most of them can be deduced from the previous section.

First of all, a parallel evaluation should at least partially be supported. A system to mark instructions that could be executed in common has to be introduced for this feature. The PCF compiler has to take care of such a system. Afterwards, the PCF interpreter can perform a parallel evaluation. This improvement requires interpreter and compiler modifications.

Additionally, the PCF compiler should be optimized to reduce the number of AND gates within the Boolean circuit. As opposed to Yao's GC, GMW's performance depends on the number of AND gates and so the compiler should avoid them whenever possible. Schneider published a book about this issue in 2012 [Sch12].

Currently the PCF interpreter is responsible to provide the appropriate gate type for the GMW backend. All gate types except AND, XOR and NOT have to be translated at runtime. This work should be done by the compiler, which could prohibit other gate types in the PCF file a priori by taking this into account during the compilation. Since the compiler only needs to do this once, while the interpreter translates the types at each execution, this would be a good approach for future works.

To reduce the problem of memory consumption for secure computation, the interpreter could load the PCF file only partially into the memory. The current implementation reads the entire file at the beginning and keeps it in memory during the whole execution. The parts could for example be defined by the function instruction or a branch within the file. Especially for devices with low computational resources, such as smartphones is this an important issue. As a result of this strategy, some instructions could be loaded multiple times, due to the fact that the program can jump back to an earlier instruction that is not stored anymore. The compiler should check for frequently used parts and mark them during the compilation. All in all this aspect strongly dependents on the execution device. Henecka and Schneider published their work in 2013 [HS13], which includes circuits and communication caching.

6 PCF 2 - C Interpreter

After PCF 1, the developers released a new project version, called PCF 2. They have introduced several changes with the new version we describe in detail in the course of this chapter. The current stage of development can be found on github[1]. A new project interpreter allowing to read and translate all existing instructions, has already been developed. The core part of the interpreter, responsible for each instruction execution, is written in C. Moreover, similar to the known PCF 1 interpreter in Java (which is described in Section 5.1), the PCF developers applied the Yao's GC protocol as backend for any gate evaluation within the procedure. The Yao's GC implementation is written in C++. The decision to provide a PCF interpreter in C allows a variety of opportunities for existing or future applications. C and C++ are still two of the most widely used and popular languages[2]. The interpreter - provided as a library - benefits from the widespread and versatile usage of the language and can be integrated in many existing C/C++ applications. Also the integration in embedded systems is feasible. In addition, many developers are familiar with the language C and therefore can work with it easily.

This chapter is concerned with the second version of the PCF interpreter in C/C++. A short overview on all alterations to the prior version is presented in the next section. Afterwards, all aspects of the interpreter are described, its construction and working progress (cf. Section 6.2). In Section 6.3, all PCF 2 instructions, that are offered for a successful circuit building, are explained. Finally, two implemented modifications for PCF 2 are introduced: an extension of the evaluation process and the implementation of several new instructions. Comparable to the PCF 1 extension in Section 5.3, we want to extend the C interpreter to support GMW in addition to Yao's GC, since GMW is under certain circumstances more efficient (cf. Section 6.4). Furthermore, the PCF 2 instruction set is enlarged by arithmetic operations to allow further optimizations and to supply a better maintenance by introducing a higher abstraction level. Section 6.5 describes all aspects of arithmetic operations.

6.1 Alterations to PCF 1

The release of a second project version called PCF 2 has introduced various alterations and improvements. The developers specified an entire new syntax for the format and implemented keywords to make the instructions and their respective meaning more readable for users. A keyword, for example `:DE` or `:TRUTH-TABLE` indicates its following value. Furthermore, brackets are used to define the start and end point of each instructions.

The instruction set has also been renewed. A few unnecessary instructions were removed, others were replaced or altered, for example by new parameters and some new ones were added. The stack environment was removed completely, since the PCF developers considered that its memory usage was disproportionate to its benefits. So the `PUSH` and `POP` instructions are no longer offered without any alternative. Therefore, new features are supplied by implementing new instructions. The `BITS` instruction for example allows users to distribute a wire's value on several other wires. Since a wire value is not limited to a single bit, but can also be an integer, it is possible to split a value bit by bit and share it. Each wire also stores additional data, so called *keydata*. They are used by some instructions to save required information. Beyond that, PCF 2 has introduced support for some arithmetic operations, like

[1] `https://github.com/cryptouva/pcf`
[2] `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`

multiplication and addition. But these operations can only be used for plaintext values so far. They do not support the handling of secret shared values, which depend on one of the inputs.

6.2 C Interpreter for PCF 2

The interpreter for PCF 2 is divided into two parts. The first part, written in C, contains the interpreter core including all required methods to read and interpret any PCF 2 instruction. The second part is written in C++. It provides a Yao's GC implementation for the gate evaluation and takes care of communication aspects between both parties during the procedure. The interpreter is in a working state and improved continually[3]. The focus of this chapter is on the interpreter core, since we add GMW as additional backend later. However, both components are presented in the following.

6.2.1 Interpreter Core

The PCF 2 interpreter core is responsible for dealing with multiple features of the PCF 2 instruction set. Methods to read in the PCF file are included as well as a precise implementation of each instruction within the file, their respective parameters and other important informations. Some options for a versatile input system, enabling various ways to read both parties' inputs, are supported in addition. The inputs can be read from a given file or as program arguments. Furthermore, the function result is printed at the end. The interpreter core also defines all required type structures, like the different elements of a wire, consisting of its stored value, a flag that decides if the wire is assigned to a secret value and so called keydata, which provide additional information for some instructions. The working flow is similar to the PCF 1 interpreter (cf. Section 5.1). At the beginning, the PCF file is read in and all contained instructions are stored (with their respective parameters) into an array. Afterwards, each instruction within this array is processed sequentially. So the processing does not happen directly, the complete file is cached by the interpreter first. Each instruction can be figured as a transition between states, which are passed by the program during the execution. A Program Counter (PC) points to the currently running instruction and hence is used to determine or manipulate the next instruction, which is for example important for jump executions. Only if a gate evaluation appears, the program needs to use Yao's GC. Two classes construct this part of the program, `pcflib` and `opdef`. Their interaction is shown in Figure 8. `opdef` stores and maintains all instructions' implementations and provides them to the program at the appropriate time. `pcflib` regulates the working flow that is described above. It maintains all methods to read and execute each instruction in order to build and evaluate the circuit.

6.2.2 Yao's GC Implementation

The second part of the interpreter is written in C++. It deals with both, the communication between the two parties and the gate evaluation using the Yao's GC protocol.

An external library is used for the communication part, called MPICH[4]. Published under a BSD-like license, MPICH is an implementation of the Message Passing Interface (MPI)[5] and supports all current MPI versions (v1.0 till v.3.0). MPI in turn is a specification for a message-passing system, that is well tested and widely used today. For Yao's GC, two different implementations are provided. The first one is secure against semi-honest adversary model, the attacker does not influence the protocol rules, but tries to gain further sensible information (cf. Section 2.1). Beyond that, another implementation of Yao's GC was developed, that requires more steps but provides protection against malicious adversaries.

[3] `https://github.com/cryptouva/pcf/tree/gh-pages/pcflib`
[4] `http://www.mpich.org/`
[5] `http://www.mpi-forum.org/docs/docs.html`

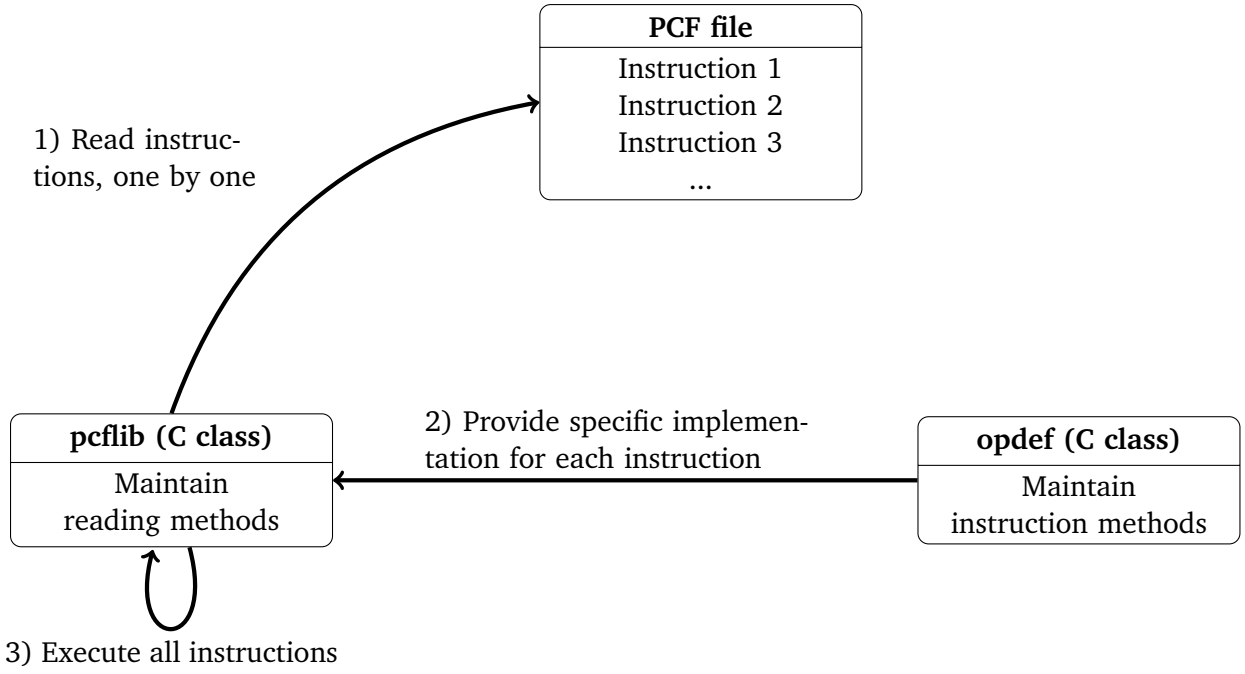

Figure 8: PCF 2 interpreter, program structure

The implementation is based on the work of [sS11]. When the program starts, the user decides with a program argument, which implementation he wants to use.

6.3 PCF 2 Instructions

Similar to PCF 1, several instructions are supported by PCF 2 to describe and build the circuit in a compact and effective way. These instructions, however, are only partly comparable to their predecessor, since many changes have been introduced. A completely new syntax is used for the instructions, that should improve the readability of the PCF program. New instructions have been introduced, a few have been replaced or removed entirely and some of them require new or different parameters. In the following section, all PCF 2 instructions are introduced including examples and a description of their respective meaning. The instruction list has been extracted from the existing C interpreter and is complete so far. It is unknown, whether further changes are planned by the PCF developers. In summer 2014, a few days before this thesis was finished, the PCF developers published a documentation about the meaning of each instruction, too. Their list can be found on github[6].

All instructions require a few parameters, which specify different options and have to be set. Similar to PCF 1, no optional parameters exist. Furthermore, all necessary labels have to be unique within the PCF file. The techniques for conditional and unconditional jumps are adopted from the first project version, but instead of `GADGET` and `FUNCTION` in PCF 1, the instruction `LABEL` defines the target and `BRANCH` (conditional) or `CALL` (unconditional) execute the jump.

We show a simple example for a PCF file in Figure 9. Again, the function is a two bit adder. The program works similar to our prior example in Figure 5. Line 1-4 define the program header. Afterwards, the inputs of both parties are read in (line 5-6) and the calculation starts (line 7). Line 15 outputs the result. Furthermore, we provide the entire program code of a 32 bit adder in Appendix B.

[6] `https://github.com/cryptouva/pcf`

```
 1 (LABEL :STR "pcfentry" )
 2 (INITBASE :BASE 1 )
 3 (LABEL :STR "main" )
 4 (CLEAR :LOCALSIZE 128 )
 5 (CALL :NEWBASE 427 :FNAME "alice" )
 6 (CALL :NEWBASE 65 :FNAME "bob" )
 7 (GATE :DEST 493 :OP1 427 :OP2 65 :TRUTH_TABLE #*0110 )
 8 (GATE :DEST 459 :OP1 493 :OP2 492 :TRUTH_TABLE #*0110 )
 9 (GATE :DEST 494 :OP1 492 :OP2 427 :TRUTH_TABLE #*0110 )
10 (GATE :DEST 495 :OP1 493 :OP2 494 :TRUTH_TABLE #*0001 )
11 (GATE :DEST 492 :OP1 427 :OP2 495 :TRUTH_TABLE #*0110 )
12 (GATE :DEST 493 :OP1 428 :OP2 66 :TRUTH_TABLE #*0110 )
13 (GATE :DEST 460 :OP1 493 :OP2 492 :TRUTH_TABLE #*0110 )
14 (CALL :NEWBASE 491 :FNAME "output_alice" )
15 (RET :VALUE 491 )
```

Figure 9: PCF 2 file example

INITBASE This instruction is called at the beginning of each PCF program to initialize the PC and the base counter. Both are required to address different positions within the program. The PC points to the current instruction and is initialized to the position of the "main" label, that defines the starting point. The base counter is mainly used during the input reading to decide, which area of the input is taken. But it is also added to almost every array position that is used to address the wires. The base counter is initialized to 1.

Example: `(INITBASE :BASE 1 )`
Can be found in the first lines of every PCF file. It sets the base counter to 1 and jumps to "main".

LABEL Defines a new label that address is saved for a call within the program. As mentioned previously, all labels have to be unique. This instruction is used to support branches during the process.

Example: `(LABEL :STR "main" )`
Defines a new label called "main". According to INITBASE, the PC is set to the address of this label on program launch.

BRANCH BRANCH is used to reset the PC, a technique that enables a jump to any desired label. First of all, it checks the wire with an ID specified by the first parameter. If the wire is known (the value has already been calculated and is a non-secret value) and the value is 1, the PC is set to the new address. Otherwise, the PC is increased by one and the next instruction within the PCF file is executed.

Example: `(BRANCH :CND 33 :TARG "PC5" )`
Checks the wire 33 (+ current base position). If the value is 1, the program jumps to label "PC5".

CONST A constant value is set to a wire. Existing entries are overwritten. The first parameter defines the desired target wire, the second specifies the value.

Example: `(CONST :DEST 33 :OP1 65 )`
The value 65 is set to wire 33 (+ current base position).

GATE This instruction is responsible for a single gate evaluation. It receives four parameter: one target wire (:DEST), two source wires (:OP1 and :OP2) and a bit encoded truth table (:TRUTH-TABLE).

The truth table defines the operation, which is applied to the source wires. A four bit value encodes the truth table, its descending sequence describes the table. For example 0001 corresponds to an AND and 0110 to an XOR operation. The calculated result is assigned to the target. Only Boolean values are allowed for this instruction, since only Boolean operations can be evaluated.

Example: `(GATE :DEST 33 :OP1 34 :OP2 35 :TRUTH-TABLE #0110 )`
0110 is the truth table for the XOR operation. Source wires are 34 and 35, the result is assigned to wire 33. The current base position is added to all IDs.

CLEAR Clears all wires in a given range. To clear a wire, its value is set to 0 and its keydata are deleted.

Example: `(CLEAR :LOCALSIZE 160 )`
Clears the 160 following wires that are based on the current base position.

CALL The CALL instruction receives two parameters: a wire ID assigning a new base position and a function name. Four functions are predefined: "alice", "bob", "output_alice" and "output_bob". The first two functions can be called to read in the respective parties' input. The other two output the result. If none of the four function is given as parameter, the program searches for an appropriated label.

Example: `(CALL :NEWBASE 33 :FNAME "alice" )`
Reads Alice's input and stores it to wires starting at ID 33 (+ current base position).

RET Returns after a function call, resets the PC and base counter and frees allocated memory. For a successful return, the value of a given wire has to be set.

Example: `(RET :VALUE 33 )`
Checks wire's value with ID 33, if the value is 1, jumps back, otherwise exits with an error.

BITS Copies the value of a wire to a given set of wires, each wire receives one bit of the value. Keydata and flags are copied, too. Can be used to distribute 32 bit value to 32 wires.

Example: `(BITS :DEST (33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64) :OP1 33 )`
All data of wire 33 are distributed to the wires 33 till 64. It is allowed (and used in some PCF files) that the source wire (:OP1) is also part of the destination wires (:DEST).

MKPTR Creates a pointer by adding the current base position to the value of a given wire.

Example: `(MKPTR :DEST 33 )`
The value of wire 33 is increased by the current base position.

COPY Enables a copy operation, given three parameters, the destination, the source and the width.

Example: `(COPY :DEST 33 :OP1 65 :OP2 32 )`
Wire 33 (+ current base position) defines the destination wire, the source is set by wire 65 (+ current base position). Because of the third parameter, 32 following wires are copied.

COPY-INDIR Another copy operation. It takes three parameters, one for the destination wire, one for the source and one for the desired width. Compared with the copy operation above, the source of this instruction is defined by a given wire's value.

Example: `(COPY-INDIR :DEST 33 :OP1 65 :OP2 32 )`

The value of wire 65 (+ current base position) sets the source wire, wire 33 defines the destination. Because of the third parameter, 32 wires (starting at 33 + current base position) are copied to the 32 destination wires.

INDIR-COPY Similar to COPY-INDIR, except that the destination wire is defined by the given value and not the source.

Example: `(INDIR-COPY :DEST 33 :OP1 65 :OP2 32 )`
The value of wire 33 (+ current base position) sets the destination wire, wire 65 defines the source. Because of the third parameter, 32 wires (starting at 65 + current base position) are copied to the 32 destination wires.

JOIN & ADD & MUL These three instructions should help to support arithmetic circuits in addition to Boolean ones. A wire stores its values as uint32_t, so arithmetic operation, like JOIN, ADD or MUL are possible. But so far they can only be used to process plaintext values and not secret shared values. All three instructions require three parameters, two for the source operators and another for the result.

6.4 Implementation of a GMW PCF 2 Interpreter

The development of a GMW implementation and its establishment as new backend is our first modification for the PCF 2 interpreter, which has been realized in the context of this thesis. The required steps to use our GMW implementation instead of the Yao's GC routine are described in this section.

In addition to the evaluation approach, the entire communication system has been restructured. Instead of MPI, we use standard C libraries, because we have used these libraries before and are more familiar with them. They allow one party (who takes the server role) to create and maintain a socket. The other party takes the client role and connects to the socket. Required information have to be shared or determined before, like the IP address of the server or the used port number for the service. Both parties use the socket to send and receive data during the evaluation process, including the masked inputs (which have to be shared), the parameter d and e for every AND gate evaluation and the respective local output to reconstruct the final result.

In the course of our work for the GMW implementation, we have designed a new instruction for further improvements of the protocol. This instruction is called `MPC-COMM` and it does not require any parameter for its execution. Within the PCF file, it consists of the following syntax:

PCF syntax: `(MPC-COMM )`

Our purpose and motivation for this instruction is to allow the programmer or any other user dealing with PCF to decide manually when both parties should share their required values d and e for the AND gate evaluation. While the interpreter executes this instruction, the number of AND gates waiting for the communication step and hence for the other party's d and e, is checked. If at least one AND gate is ready to exchange the values, the communication starts and the evaluation can be continued.

In the long-term planning this instruction can be used to combine several interaction steps of AND gates in order to reduce the number of required communication rounds. Every time an AND gate does not depend on a gate, that has not been evaluated yet, the gates can be grouped into layers. Afterwards, a communication step is only required for every layer and not every single AND gate. This strategy enables a good optimization for GMW and was mentioned in various sections of this thesis (e.g. Section 2.6 or Section 5.5). But to take full advantage of this feature, the compiler has to be modified as well.

The compiler selects the AND gates, which can be grouped and which depend on each other. By now, the compiler is not optimized to decrease the circuit depth in this way and therefore the communication step has to start for each AND gate or the programmer has to decide it manually and that might be impossible because of the large size of a circuit.

During the evaluation, we only need to modify the `GATE` instructions to use our GMW implementation. Furthermore, methods for additional GMW options have to be implemented, like the secret input sharing or the MT generation within the setup phase.

6.5 Arithmetic Operations

As part of this thesis, the PCF project was extended by one more feature, the support for arithmetic operations including their identification and evaluation. In order to do that, we enhance the current set of instructions by several arithmetic operations: the basic operation addition, a comparison operation and a multiplexer operation. This extension is an useful improvement for PCF 2 because of two reasons: maintenance and optimization. Both will be explained in the following section. To be compatible with current PCF 2 components, our modifications do not change or remove existing sourcecode, since we only add our new instructions and their implementations.

PCF in both, version 1 and 2, can deal with Boolean operations only. All arithmetic operations of a function are converted during the compilation into their respective Boolean representation and then are evaluated by Boolean gates. A truth table helps to determine which operation is needed for each gate, two wires define the source values, another wire is assigned with the result. So basically, an ordinary logic gate is simulated. PCF 2 provides the `GATE` instruction for Boolean operations (cf. Section 6.3). This strategy does not enable the kind of abstraction, programmers are used to deal with in their software. Of course, one advantage of the PCF project is its independence allowing an interaction without any specialized knowledge, also in respect to the function representation and evaluation. A PCF file is ordinarily created by the compiler without the need for manual modifications. The interpreter reads in and handles all instructions within the PCF file automatically, too. So a programmer does not have to know anything about the file, its instructions or the process at all. Nevertheless, a basic understanding about PCF and its instructions may be helpful, for example for debugging reasons. As the project is very young and long-term tests do not exist yet, a programmer may fix potential bugs directly inside the PCF file. Also further optimizations can improve the evaluation process and thereby are a good reason to become more familiar with the format. The more understandable the instructions are, the easier it is for a developer to comprehend the layout of any PCF file. Arithmetic operations are a good example for that. If a function contains an arithmetic operation, a programmer expects a similar operation within the PCF file. This is more readable and intuitive than its respective Boolean representation. All in all with the help of arithmetic operations, the maintenance of a PCF file is easier, because it enables a higher abstraction level.

A support for arithmetic circuits is in the focus of the PCF developer team as well. It should be "used in homomorphic encryption and multiparty computation applications, as well as applications in verifiable computation"[7]. The current implementation does not include this feature by now, because the existing instructions ADD, `MUL` and `JOIN` deal with plaintext values only. Both improvements could be combined in future developments.

[7] Kreuter, Shelat; Page 2, Section 2.4 [Ks13]

6.5.1 Implementation of Arithmetic Operations

All arithmetic operations should be able to calculate values with arbitrary length. However, both lengths are defined by the same parameter with the consequence that the calculation only accepts two values of equal length. This feature requires a new form of addressing, since each wire is assigned to one value, saved as an uint32_t variable. So although it is possible to store an operator for a 32 bit arithmetic operations entirely in a single wire, we chose another way. The current implementation assumes that only a Boolean value (0 or 1) is stored into a single wire to simplify the handling of various instructions, like the Boolean gate evaluation. Values greater than 1 cause a program crash. As a consequence, each affected instruction would need to check the currently used addressing mode and converts it if necessary. Moreover, each bit within the uint32_t variable needs to be addressed individually during the evaluation. And also the length would be limited to maximal 32 bit for each operator, which maybe would not be suitable for some scenarios. These things taken into consideration, it is more practical to retain the current kind of addressing and adapt the arithmetic operations to it. Hence the operators are saved into a number of wires, each of them is assigned to one bit. Similar to an array, the first wire references the starting point and the following wires supply the remaining bits. Each bit can be addressed with its respective wire identification. The `BITS` instruction is one good opportunity to define operators for arithmetic operations, because it allows to distribute a single value bit by bit to an arbitrary number of destination wires.

For now, we support the following three arithmetic operations, which calculations are taken from [KSS09]:

Addition: The addition operation calculates the sum of two values x and y of equal length. A third wire saves the result. Its implementation realizes a simple adder, including carry and parity bits. The carry bit is calculated with (x & y are the two operators, r the result and i works as iterator):

$$r_{i+1} = (x_i \oplus r_i) \wedge (y_i \oplus r_i) \oplus r_i \text{ with } r_0 = 0$$

To support this calculation, a new PCF instruction has to be defined:

```
(MPC-ADD :DEST 65 :OP1 1 :OP2 33 :LEN 32 :BUF 97 )
```

In this example two 32 bit values (one starts at ID 1, the other at ID 33) are added and the result is stored in the wire with ID 65.

Comparator: Using the comparator operation, the interpreter determines, if the first value is greater than the second one. A positive result (the number is greater) is stored as 1 in a given third wire, otherwise the third wire is assigned to 0. Within the operation, the calculation is implemented as follows (again, x & y are the two operators, r's last bit the final result and i works as iterator):

$$r_{i+1} = (x_i \oplus r_i) \wedge (y_i \oplus r_i) \oplus x_i \text{ with } r_0 = 0$$

We define a new PCF instruction to support this feature:

```
(MPC-CMP :DEST 65 :OP1 1 :OP2 33 :LEN 32 :BUF 66 )
```

In this example two 32 bit values (the first starts at ID 1, the second at ID 33) are compared and the result (1 if the first value is greater than the second one, else 0) is stored in the wire with ID 65.

Multiplexer: Our multiplexer needs three operators for its execution. The third one decides if the first or the second is passed on to the destination. The internal implementation translates the multiplexer into (with x and y as input, r is the result and s decides which one is passed and again i works as iterator):

$$r_i = x_i \oplus (s \wedge (x_i \oplus y_i))$$

Once again, we have defined a new PCF instruction to handle such an operation:
`(MPC-MUX :DEST 66 :OP1 1 :OP2 33 :OP3 65 :LEN 32 :BUF 98 )`

In this example the value of wire 65 is checked and if it is assigned to 1, the wires 1-32 are copied to the wires 66-97 or (if wire 65 is assigned to 0) the wires 33-64 are copied to the wires 66-97.

During the execution, our new instructions are read in together with all others and cached by the program. Their syntax is shown above. All of them define the destination wire first, following the standard practices of PCF. As already mentioned at the beginning of this section, the given wire does not store the entire result, but sets the starting point describing an array of wires, which store one bit of the result in each case. Afterwards all operators are read in. The addition and comparison operations supply two operators, the multiplexer requires one more for its calculation. At last, two parameter specify the wires, which are responsible for the length and a buffer. The former parameter sets the fixed length for all operators by defining the number of wires, which store the value. Besides, a buffer wire is used to cache values during the calculation. This wire applies the same array technique as both operators.

After all instructions are fetched, the evaluation starts. If one of our instructions occur during the process, its calculation is carried out according to the equation described above. No further steps are necessary.

Some of the calculations require an interaction between both parties, since they use an AND gate that inputs could depend on secret shared values. These gates are already evaluated with the aid of the GMW technique. Our extension to support GMW in addition was described in Section 6.4.

6.6 PCF 2 and Arithmetic Operations - Summary and Discussion

The topic of this chapter was the PCF 2 interpreter, written in C/C++, including a description of its structure, working procedure and functionality (cf. Section 6.2). We worked out the alterations between the previous PCF project version and the current release (cf. Section 6.1). We further described the improvements, which were incorporated into the new version. A number of supported instructions were introduced and exemplified in Section 6.3.

Finally two modifications for the interpreter were implemented. At first, we added a new PCF backend in order to evaluate gates with the approach of GMW instead of Yao's GC. Afterwards, the instruction set was enlarged to support arithmetic operations in addition to the existing Boolean ones. Currently, our operations include an adder, a comparison and a multiplexer.

The introduction of arithmetic operations shall improve the PCF project by creating a higher abstraction level. It simplifies the PCF file and therefore improves its maintenance and for example in case of an error, its debugging possibility. But beyond that, more advantages are obvious. Since arithmetic instructions are a much more compact way to describe arithmetic operations, we reduced the size of the PCF file. As an example, the addition of two 32 bit values requires 192 gates collectively and therefore 192 `GATES` instructions within the PCF file. As a result of our extension, only one instruction is needed for the addition description. The arithmetic comparison of two numbers or a multiplexer also requires a lot of instructions, which can also be reduced to one now (the PCF 2 millionaire function consists of 1065 `GATES` instructions within the PCF file).

All in all, the compactness ensures a better maintenance for the PCF files. Additionally, electronic devices with limited resources will benefit from that, due to the fact that they can handle functions more easily. To save the PCF file, less storage is needed and the PCF interpreter has to load and keep less

instructions during the execution, which relieves the main memory.

A disadvantage of our arithmetic operations could be that users can not implement their own improvements or adaptations for these operations. If users want to execute an addition of two values in a different way than the PCF interpreter provides using its arithmetic operations, they have to use the usual `GATES` instructions again. But since we have not replaced any instruction, this is still possible. As a further consequence of our extension, the PCF components became more complex, because the number of code lines were increased and the arithmetic instructions require additional logic. The maintenance costs increase as well as the susceptibility to errors, due to the complexity.

Instead of arithmetic operation instructions, it is also possible to calculate each arithmetic operation in a PCF function and call them respectively on demand with the desired operators as arguments. However, these special functions could then use ordinary Boolean instructions to calculate the result and return it. This strategy would avoid multiple placements of the same arithmetic operation and thereby reduce the PCF file size as well. But the size would not be as small as it would be with our new instructions, as the function itself is saved within the PCF file. Also the maintenance would be improved by applying reusable functions. But again, the effect would not be as big as our approach provides. The PCF file is the best possibility for developers to implement their own optimizations beyond the default PCF ones, since the PCF compiler and interpreter should not specially adapted to each program of PCF file. This offers many opportunities but also error sources. Our new instructions provide a better protection against mistakes than special functions do. As the function is a part of the entire program, a developer could for example accidentally overwrite important wires during his modifications. A single instruction is easier to handle than an entire function, especially because with our instructions the PCF interpreter ensures the functionality of any arithmetic operation. Finally, if a developer really wants to use functions for arithmetic operations, he is free to do so in his programs. But since it is not the most intuitive way, it should not be the default strategy in a system aiming to be usable for people without specialized knowledge. The biggest advantage of such functions is that the PCF interpreter does not require any modifications, because only the compiler needs to detect arithmetic operations and translate them into functions.

The advantages and disadvantages of our GMW implementation related to the backend replacement were discussed in Section 6.6 and can be fully applied to PCF 2.

6.7 PCF 2 - Future Work

To continue the work around the PCF 2 project, some further improvements can be done. As we only realized three arithmetic operations, future works could focus on further ones (e.g. multiplication).

Moreover, the expression of a single instruction can be further abstracted by representing an entire calculation. If a general known algorithm appears within the evaluated function, it could be described as a distinct instruction. An example would be the greatest common divisor (gcd) of two numbers, which is based on arithmetic operations but would provide an even higher abstraction level than primitive operations, like the addition of two values. Many other algorithms could be supplied in that way. However, the PCF interpreter then needs to decide, how any algorithm is implemented and therefore calculated. If more approaches exist for a successful calculation (the gcd for example can be determined with different methods), somebody has to define the most suitable one for the interpreter. Since each calculation method has certain advantages this could be difficult to decide. Also the complexity of both, the PCF compiler and interpreter, increases with the number of supplied instructions. To avoid a disproportionate disadvantage for the maintenance of any PCF components, the introduction of new instructions should be limited to the frequently used and well-known algorithms only.

Although the approach of functions, that are used for arithmetic operations within the PCF file does not replace our new instructions, it could be extended, too. The PCF compiler could support the inclusion of further extern PCF files, similar to a library system. That way, functions that are used often could be outsourced and imported on demand. Other programs could then import these files, too. This is not limited to arithmetic operations, but could also include entire algorithms or program parts. During the compilation, the compiler checks and adapts accordingly what imports the environment supplied. However, such a library system would require bigger changes in each project component but might be interesting for future works.

As another topic for potential future works is the analysis of the function to recognize and tag procedures, that are independent from each other and hence can be parallel evaluated. Similar to a parallel evaluation of AND gates for GMW, arithmetic operations or even entire subfunctions could be parallel evaluated, too. This would increase the complexity of all PCF components again, but it would also improve the performance. During the compilation, the function could be tested for possible parallelism. The OpenMP[8] project for example provides different techniques to analyze a function for such a purpose. The interpreter has to recognize and execute the parallel parts afterwards.

The future work for our GMW implementation as new backend was described in Section 5.6 and can be applied to PCF 2 as well.

[8] http://openmp.org/wp/

7 Conclusion

This last chapter focuses on a summary on the thesis topic and gives an overview about our results, their meaning and possible future works.

7.1 Summary and Discussion

In the course of this thesis, we have worked on different aspects around the PCF project. The project was started in 2013 by a developer team from Virginia and Oregon aiming to improve the way circuits for secure computation can be built, stored and processed in comparison to existing procedures. It actually affords new possibilities in the field of secure computation by reducing the required performance and thereby enables the process even for devices with limited resources. Both published versions were main topics for this thesis and we have especially addressed two issues. At first, we add our implementation of GMW to the existing PCF backends. The interpreter has only supported the approach of Yao's GC so far, but we wanted to demonstrate the advantage of GMW. Secondly, we designed some new instructions, which enable PCF 2 to handle arithmetic operations in addition to Boolean ones. Both extensions can be seemed as improvements for the PCF project.

The usage of GMW as evaluation approach has shown that a performance faster than the existing approach of Yao's GC can be reached, although we have only realized a small part of all possible optimizations for GMW. Many further improvements are therefore conceivable for the GMW backend, which could be the topic for future works.

Our new instructions improve the usability of PCF files by increasing their readability and hence improving their maintenance. Furthermore, we could decrease the file size, as arithmetic operations allow a more compact circuit description.

7.2 Future Work

Some enhancements can be done to continue the work of this thesis and the PCF project at all. They are introduced in the following section.

Section 5.6 already lists some future work to extend and improves the GMW backend within the PCF compiler and interpreter. The most important work might be the depth optimization for circuits in order to enable a grouping of AND gates, which can be parallel evaluated. Other improvements are for example the reduction and avoidance of AND gates during the circuit building or the prevention of any other `GATE` types except AND, XOR and NOT gates as part of the compilation. The PCF interpreter could also cache the reading instructions partially instead of loading and storing them all at once.

Our implementation about arithmetic operations can be continued with several works as well. We have described them in detail in Section 6.7. Among others, further improvements include the introduction of the following features: more arithmetic operations could be implemented to expand our set of supported operations. Furthermore, instructions could even describe entire algorithms as a new abstraction level. A parallel evaluation of instructions could improve the performance and might be realized by future projects, too.

In addition to our work, other areas of the PCF project could be in the focus of future works:

7.2.1 Secure Multi-Party Computation

To support more use cases, the PCF project should enable secure multi-party computation. By now, only two parties can participate in the process. Since GMW in general allows secure multi-party computation, the compiler and interpreter could be improved accordingly.

7.2.2 Communication

Any interaction between both parties requires some kind of communication. To transfer data, it might be useful for some scenarios to extend the interpreter by further communication techniques, like Bluetooth or NFC. By now, one party (the server) creates a TCP socket and the other party (the client) connects to it.

7.2.3 Work in Progress

Other improvements are already in progress. Currently, the PCF developers are working on some enhancements around the front-end compiler [Ks13]. They try to support LLVM [LA04] bytecode next to the currently used LCC compiler. LLVM allows them "to support a larger variety of languages"[1].

Several improvements for the PCF compiler have already been implemented [Ks13], including support for homomorphic encryption (not complete yet), linking functions (like AES S-Box) and compiler optimizations. Unfortunately, no schedule for these works is known, but all improvements ensure that the PCF project supplies more and more important functionalities and therefore is still an interesting project for all users, who want to work in this field of cryptography.

[1] Kreuter, Shelat; Page 2, Section 2.1 [Ks13]

Abbreviations

API	Application Programming Interface
GC	Garbled Circuits
gcd	greatest common divisor
GMW	Goldreich, Micali and Wigderson
LCC	Little C Compiler
MPI	Message Passing Interface
MT	Multiplication Triple
OT	Oblivious Transfer
PC	Program Counter
PCF	Portable Circuit Format
R-OT	Random-OT
SFE	Secure Function Evaluation

Bibliography

[ALSZ13] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. "More Efficient Oblivious Transfer and Extensions for Faster Secure Computation". In: *ACM Conference on Computer and Communications Security (ACM CCS '13)*. Berlin, Germany: ACM, 2013, pp. 535–548. URL: http://doi.acm.org/10.1145/2508859.2516738 (cit. on pp. 8, 10, 23).

[BDNP08] Assaf Ben-David, Noam Nisan, and Benny Pinkas. "FairplayMP: A System for Secure Multi-party Computation". In: *ACM Conference on Computer and Communications Security (ACM CCS '08)*. Alexandria, Virginia, USA: ACM, 2008, pp. 257–266. URL: http://doi.acm.org/10.1145/1455770.1455804 (cit. on p. 12).

[Bea92] Donald Beaver. "Efficient Multiparty Protocols Using Circuit Randomization". English. In: *Advances in Cryptology - EUROCRYPT*. Vol. 576. LNCS. Springer, 1992, pp. 420–432 (cit. on p. 10).

[Bea96] Donald Beaver. "Equivocable Oblivious Transfer". English. In: *Advances in Cryptology - EUROCRYPT*. Vol. 1070. LNCS. Springer, 1996, pp. 119–130 (cit. on p. 8).

[BHKR13] Mihir Bellare, Viet Tung Hoang, Sriram Keelveedhi, and Phillip Rogaway. "Efficient Garbling from a Fixed-Key Blockcipher". In: IEEE Computer Society, 2013, pp. 478–492 (cit. on p. 9).

[CHK+12] SeungGeol Choi, Kyung-Wook Hwang, Jonathan Katz, Tal Malkin, and Dan Rubenstein. "Secure Multi-Party Computation of Boolean Circuits with Applications to Privacy in On-Line Marketplaces". English. In: *Topics in Cryptology*. Vol. 7178. LMCS. Springer, 2012, pp. 416–432 (cit. on pp. 6, 8, 11 sqq.).

[CMSA12] Gianpiero Costantino, Fabio Martinelli, Paolo Santi, and Dario Amoruso. "An Implementation of Secure Two-party Computation for Smartphones with Application to Privacy-preserving Interest-cast". In: *Mobile Computing and Networking (Mobicom '12)*. Istanbul, Turkey: ACM, 2012, pp. 447–450. URL: http://doi.acm.org/10.1145/2348543.2348607 (cit. on p. 13).

[Dem13] Daniel Demmler. "Hardware-Assisted Two-Party Secure Computation on Mobile Devices". In: Master thesis TU Darmstadt. 2013 (cit. on pp. 17, 29).

[DSZ14] Daniel Demmler, Thomas Schneider, and Michael Zohner. "Ad-Hoc Secure Two-Party Computation on Mobile Devices using Hardware Tokens". In: *USENIX Security Symposium (USENIX Sec '14)*. Full version: http://eprint.iacr.org/2014/467. USENIX, 2014, pp. 893–908. URL: http://thomaschneider.de/papers/DSZ14.pdf (cit. on pp. 10, 17, 29).

[EGL85] Shimon Even, Oded Goldreich, and Abraham Lempel. "A Randomized Protocol for Signing Contracts". In: *ACM* 28.6 (1985), pp. 637–647 (cit. on p. 8).

[FHK+14] Martin Franz, Andreas Holzer, Stefan Katzenbeisser, Christian Schallhart, and Helmut Veith. "CBMC-GC: An ANSI C Compiler for Secure Two-Party Computations". In: *Compiler Construction (CC '14)*. Ed. by Albert Cohen. Vol. 8409. Springer, 2014, pp. 244–249. URL: http://dx.doi.org/10.1007/978-3-642-54807-9_15 (cit. on p. 13).

[GMW87] Oded Goldreich, Siltnb Micali, and Avi Wigderson. “How to Play ANY Mental Game”. In: *ACM Symposium on Theory of Computing (ACM STOC ’87)*. New York, New York, USA: ACM, 1987, pp. 218–229. URL: http://doi.acm.org/10.1145/28395.28420 (cit. on pp. 7, 10).

[HCE11] Yan Huang, Peter Chapman, and David Evans. “Privacy-preserving Applications on Smartphones”. In: *USENIX Conference on Hot Topics in Security (USENIX HotSec ’11)*. San Francisco, CA: USENIX, 2011, pp. 4–4. URL: http://dl.acm.org/citation.cfm?id=2028040.2028044 (cit. on p. 17).

[HEKM11] Yan Huang, David Evans, Jonathan Katz, and Lior Malka. “Faster Secure Two-party Computation Using Garbled Circuits”. In: *USENIX Security Symposium (USENIX Sec ’11)*. San Francisco, CA: USENIX, 2011, pp. 35–35. URL: http://dl.acm.org/citation.cfm?id=2028067.2028102 (cit. on pp. 9, 13 sq.).

[HKS+10] Wilko Henecka, Stefan Kögl, Ahmad-Reza Sadeghi, Thomas Schneider, and Immo Wehrenberg. “TASTY: Tool for Automating Secure Two-partY computations”. In: *ACM Conference on Computer and Communications Security (ACM CCS ’10)*. Code: http://encrypto.de/code/TASTY. ACM, 2010, pp. 451–462. URL: http://thomaschneider.de/papers/HKSSW10.pdf (cit. on p. 13).

[HS13] Wilko Henecka and Thomas Schneider. “Faster Secure Two-Party Computation with Less Memory”. In: *ACM Symposium on Information, Computer and Communications Security (ASIACCS ’13)*. Code: http://encrypto.de/code/me-sfe. ACM, 2013, pp. 437–446. URL: http://thomaschneider.de/papers/HS13.pdf (cit. on pp. 13, 30).

[IKNP03] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. “Extending Oblivious Transfers Efficiently”. In: *Advances in Cryptology - EUROCRYPTO*. Vol. 2729. LNCS. Springer, 2003, pp. 145–161. URL: http://www.iacr.org/cryptodb/archive/2003/CRYPTO/1432/1432.pdf (cit. on p. 8).

[KS08] Vladimir Kolesnikov and Thomas Schneider. “Improved Garbled Circuit: Free XOR Gates and Applications”. In: *International Colloquium on Automata, Languages and Programming (ICALP ’08)*. Vol. 5126. Springer, 2008, pp. 486–498. URL: http://thomaschneider.de/papers/KS08XOR.pdf (cit. on pp. 9, 12).

[Ks13] Benjamin Kreuter and abhi shelat. “Lessons Learned with PCF: Scaling Secure Computation”. In: *ACM Workshop on Language Support for Privacy-enhancing Technologies (ACM PETShop ’13)*. Berlin, Germany: ACM, 2013, pp. 7–10. URL: http://doi.acm.org/10.1145/2517872.2517877 (cit. on pp. 37, 43).

[KsMB13a] Benjamin Kreuter, abhi shelat, Benjamin Mood, and Kevin Butler. “PCF: A Portable Circuit Format for Scalable Two-Party Secure Computation”. In: *USENIX Security Symposium (USENIX Sec ’13)*. USENIX, 2013, pp. 321–336. URL: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/kreuter (cit. on pp. 5, 14, 16, 19).

[KsMB13b] Benjamin Kreuter, abhi shelat, Benjamin Mood, and Kevin Butler. *PCF: Scaling Secure Computation*. Presentation, https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/kreuter. 2013 (cit. on p. 16).

[KSS09] Vladimir Kolesnikov, Ahmad-Reza Sadeghi, and Thomas Schneider. “Improved Garbled Circuit Building Blocks and Applications to Auctions and Computing Minima”. In: *International Conference on Cryptology and Network Security (CANS ’09)*. Vol. 5888. Full version: http://eprint.iacr.org/2009/411. Springer, 2009, pp. 1–20. URL: http://thomaschneider.de/papers/KSS09.pdf (cit. on pp. 25, 38).

[KsS12] Benjamin Kreuter, abhi shelat, and Chih-Hao Shen. "Billion-gate Secure Computation with Malicious Adversaries". In: *USENIX Security Symposium (USENIX Sec '12)*. Bellevue, WA: USENIX, 2012, pp. 14–14. URL: http://dl.acm.org/citation.cfm?id=2362793.2362807 (cit. on pp. 13 sq.).

[KSS13] Marcel Keller, Peter Scholl, and Nigel P. Smart. "An Architecture for Practical Actively Secure MPC with Dishonest Majority". In: *ACM Conference on Computer & Communications Security (ACM CCS '13)*. Berlin, Germany: ACM, 2013, pp. 549–560. URL: http://doi.acm.org/10.1145/2508859.2516744 (cit. on p. 13).

[LA04] Chris Lattner and Vikram Adve. "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In: *Code Generation and Optimization (CGO '04)*. Palo Alto, California: IEEE Computer Society, 2004, pp. 75–. URL: http://dl.acm.org/citation.cfm?id=977395.977673 (cit. on p. 43).

[Mal11] Lior Malka. "VMCrypt: Modular Software Architecture for Scalable Secure Computation". In: *ACM Conference on Computer and Communications Security (ACM CCS '11)*. Chicago, Illinois, USA: ACM, 2011, pp. 715–724. URL: http://doi.acm.org/10.1145/2046707.2046787 (cit. on pp. 13 sq.).

[MLB12] Benjamin Mood, Lara Letaw, and Kevin Butler. "Memory-Efficient Garbled Circuit Generation for Mobile Devices". English. In: *Financial Cryptography and Data Security*. Vol. 7397. LNCS. Springer, 2012, pp. 254–268 (cit. on pp. 13 sq.).

[MNPS04] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. "Fairplay - a Secure Two-party Computation System". In: *USENIX Security Symposium (USENIX Sec '04)*. San Diego, CA: USENIX, 2004, pp. 20–20. URL: http://dl.acm.org/citation.cfm?id=1251375.1251395 (cit. on pp. 12, 14).

[NP01] Moni Naor and Benny Pinkas. "Efficient Oblivious Transfer Protocols". In: *ACM Symposium on Discrete Algorithms (ACM SODA '01)*. Washington, D.C., USA: Society for Industrial and Applied Mathematics, 2001, pp. 448–457. URL: http://dl.acm.org/citation.cfm?id=365411.365502 (cit. on p. 8).

[NP05] Moni Naor and Benny Pinkas. "Computationally Secure Oblivious Transfer". English. In: *Journal of Cryptology* 18.1 (2005), pp. 1–35 (cit. on p. 8).

[PSSW09] Benny Pinkas, Thomas Schneider, Nigel P. Smart, and Stephen C. Williams. "Secure Two-Party Computation is Practical". In: *Advances in Cryptology - ASIACRYPT*. Vol. 5912. LNCS. Full version at http://eprint.iacr.org/2009/314. Springer, 2009, pp. 250–267. URL: http://thomaschneider.de/papers/PSSW09.pdf (cit. on p. 9).

[Rab81] Michael O. Rabin. "How To Exchange Secrets with Oblivious Transfer". In: (1981). Harvard University Technical Report. URL: http://eprint.iacr.org/2005/187 (cit. on p. 8).

[RHH14] Aseem Rastogi, Matthew A. Hammer, and Michael Hicks. "Wysteria: A Programming Language for Generic, Mixed-Mode Multiparty Computations". In: *IEEE Symposium on Security and Privacy*. Website: https://bitbucket.org/aseemr/wysteria/wiki/Home. 2014 (cit. on p. 13).

[Sch12] Thomas Schneider. *Engineering Secure Two-Party Computation Protocols: Design, Optimization, and Applications of Efficient Secure Function Evaluation*. Springer, 2012, p. 138 (cit. on p. 29).

[sS11] abhi shelat and Chih-hao Shen. "Two-Output Secure Computation with Malicious Adversaries". English. In: *Advances in Cryptology - EUROCRYPT*. Vol. 6632. LNCS. Springer, 2011, pp. 386–405. URL: http://dx.doi.org/10.1007/978-3-642-20465-4_22 (cit. on p. 33).

[SZ13] Thomas Schneider and Michael Zohner. “GMW vs. Yao? Efficient Secure Two-Party Computation with Low Depth Circuits”. In: *Financial Cryptography and Data Security (FC ’13)*. Vol. 7859. Springer, 2013, pp. 275–292. URL: http://thomaschneider.de/papers/SZ13.pdf (cit. on pp. 6, 8, 10 sqq., 24, 28).

[Yao82] Andrew C. Yao. “Protocols for Secure Computations”. In: *Foundations of Computer Science (FOCS ’82)*. IEEE Computer Society, 1982, pp. 160–164. URL: http://dx.doi.org/10.1109/SFCS.1982.88 (cit. on p. 25).

[Yao86] Andrew C. Yao. “How to Generate and Exchange Secrets”. In: *Foundations of Computer Science (FOCS ’86)*. IEEE Computer Society, 1986, pp. 162–167. URL: http://dx.doi.org/10.1109/SFCS.1986.25 (cit. on pp. 5, 7, 9).

[ZSB13] Yihua Zhang, Aaron Steele, and Marina Blanton. “PICCO: a general-purpose compiler for private distributed computation”. In: *ACM Conference on Computer & Communications Security (ACM CCS ’13)*. Berlin, Germany: ACM, 2013, pp. 813–826. URL: http://doi.acm.org/10.1145/2508859.2516752 (cit. on p. 13).

Appendix

Appendix A - PCF 1 file with adder function

```
GADGET: MAIN
CLABEL ALICEINLENGTH 32
CLABEL BOBINLENGTH 32
CLABEL xxx 64
SETLABELC ALICEINLENGTH 32
SETLABELC BOBINLENGTH 32
0000 33 0 0
1111 34 0 0
FUNC main xxx
FUNCTION: main
ALICEINPUT32 0 0
0110 100 0 33
0110 101 1 33
0110 102 2 33
0110 103 3 33
0110 104 4 33
0110 105 5 33
0110 106 6 33
0110 107 7 33
0110 108 8 33
0110 109 9 33
0110 110 10 33
0110 111 11 33
0110 112 12 33
0110 113 13 33
0110 114 14 33
0110 115 15 33
0110 116 16 33
0110 117 17 33
0110 118 18 33
0110 119 19 33
0110 120 20 33
0110 121 21 33
0110 122 22 33
0110 123 23 33
0110 124 24 33
0110 125 25 33
0110 126 26 33
0110 127 27 33
0110 128 28 33
0110 129 29 33
```

```
42 | 0110  130  30  33
43 | 0110  131  31  33
44 | BOBINPUT32  329  0
45 | 0110  165  0  33
46 | 0110  166  1  33
47 | 0110  167  2  33
48 | 0110  168  3  33
49 | 0110  169  4  33
50 | 0110  170  5  33
51 | 0110  171  6  33
52 | 0110  172  7  33
53 | 0110  173  8  33
54 | 0110  174  9  33
55 | 0110  175  10  33
56 | 0110  176  11  33
57 | 0110  177  12  33
58 | 0110  178  13  33
59 | 0110  179  14  33
60 | 0110  180  15  33
61 | 0110  181  16  33
62 | 0110  182  17  33
63 | 0110  183  18  33
64 | 0110  184  19  33
65 | 0110  185  20  33
66 | 0110  186  21  33
67 | 0110  187  22  33
68 | 0110  188  23  33
69 | 0110  189  24  33
70 | 0110  190  25  33
71 | 0110  191  26  33
72 | 0110  192  27  33
73 | 0110  193  28  33
74 | 0110  194  29  33
75 | 0110  195  30  33
76 | 0110  196  31  33
77 | 0110  1034  100  165
78 | 0110  1035  1034  33
79 | 0110  1036  100  165
80 | 0110  1037  100  33
81 | 0001  1038  1036  1037
82 | 0110  1039  100  1038
83 | 0110  1040  101  166
84 | 0110  1041  1040  1039
85 | 0110  1042  101  166
86 | 0110  1043  101  1039
87 | 0001  1044  1042  1043
88 | 0110  1045  101  1044
89 | 0110  1046  102  167
90 | 0110  1047  1046  1045
91 | 0110  1048  102  167
```

92	0110	1049	102	1045
93	0001	1050	1048	1049
94	0110	1051	102	1050
95	0110	1052	103	168
96	0110	1053	1052	1051
97	0110	1054	103	168
98	0110	1055	103	1051
99	0001	1056	1054	1055
100	0110	1057	103	1056
101	0110	1058	104	169
102	0110	1059	1058	1057
103	0110	1060	104	169
104	0110	1061	104	1057
105	0001	1062	1060	1061
106	0110	1063	104	1062
107	0110	1064	105	170
108	0110	1065	1064	1063
109	0110	1066	105	170
110	0110	1067	105	1063
111	0001	1068	1066	1067
112	0110	1069	105	1068
113	0110	1070	106	171
114	0110	1071	1070	1069
115	0110	1072	106	171
116	0110	1073	106	1069
117	0001	1074	1072	1073
118	0110	1075	106	1074
119	0110	1076	107	172
120	0110	1077	1076	1075
121	0110	1078	107	172
122	0110	1079	107	1075
123	0001	1080	1078	1079
124	0110	1081	107	1080
125	0110	1082	108	173
126	0110	1083	1082	1081
127	0110	1084	108	173
128	0110	1085	108	1081
129	0001	1086	1084	1085
130	0110	1087	108	1086
131	0110	1088	109	174
132	0110	1089	1088	1087
133	0110	1090	109	174
134	0110	1091	109	1087
135	0001	1092	1090	1091
136	0110	1093	109	1092
137	0110	1094	110	175
138	0110	1095	1094	1093
139	0110	1096	110	175
140	0110	1097	110	1093
141	0001	1098	1096	1097

142	0110 1099 110 1098
143	0110 1100 111 176
144	0110 1101 1100 1099
145	0110 1102 111 176
146	0110 1103 111 1099
147	0001 1104 1102 1103
148	0110 1105 111 1104
149	0110 1106 112 177
150	0110 1107 1106 1105
151	0110 1108 112 177
152	0110 1109 112 1105
153	0001 1110 1108 1109
154	0110 1111 112 1110
155	0110 1112 113 178
156	0110 1113 1112 1111
157	0110 1114 113 178
158	0110 1115 113 1111
159	0001 1116 1114 1115
160	0110 1117 113 1116
161	0110 1118 114 179
162	0110 1119 1118 1117
163	0110 1120 114 179
164	0110 1121 114 1117
165	0001 1122 1120 1121
166	0110 1123 114 1122
167	0110 1124 115 180
168	0110 1125 1124 1123
169	0110 1126 115 180
170	0110 1127 115 1123
171	0001 1128 1126 1127
172	0110 1129 115 1128
173	0110 1130 116 181
174	0110 1131 1130 1129
175	0110 1132 116 181
176	0110 1133 116 1129
177	0001 1134 1132 1133
178	0110 1135 116 1134
179	0110 1136 117 182
180	0110 1137 1136 1135
181	0110 1138 117 182
182	0110 1139 117 1135
183	0001 1140 1138 1139
184	0110 1141 117 1140
185	0110 1142 118 183
186	0110 1143 1142 1141
187	0110 1144 118 183
188	0110 1145 118 1141
189	0001 1146 1144 1145
190	0110 1147 118 1146
191	0110 1148 119 184

192	0110	1149	1148	1147
193	0110	1150	119	184
194	0110	1151	119	1147
195	0001	1152	1150	1151
196	0110	1153	119	1152
197	0110	1154	120	185
198	0110	1155	1154	1153
199	0110	1156	120	185
200	0110	1157	120	1153
201	0001	1158	1156	1157
202	0110	1159	120	1158
203	0110	1160	121	186
204	0110	1161	1160	1159
205	0110	1162	121	186
206	0110	1163	121	1159
207	0001	1164	1162	1163
208	0110	1165	121	1164
209	0110	1166	122	187
210	0110	1167	1166	1165
211	0110	1168	122	187
212	0110	1169	122	1165
213	0001	1170	1168	1169
214	0110	1171	122	1170
215	0110	1172	123	188
216	0110	1173	1172	1171
217	0110	1174	123	188
218	0110	1175	123	1171
219	0001	1176	1174	1175
220	0110	1177	123	1176
221	0110	1178	124	189
222	0110	1179	1178	1177
223	0110	1180	124	189
224	0110	1181	124	1177
225	0001	1182	1180	1181
226	0110	1183	124	1182
227	0110	1184	125	190
228	0110	1185	1184	1183
229	0110	1186	125	190
230	0110	1187	125	1183
231	0001	1188	1186	1187
232	0110	1189	125	1188
233	0110	1190	126	191
234	0110	1191	1190	1189
235	0110	1192	126	191
236	0110	1193	126	1189
237	0001	1194	1192	1193
238	0110	1195	126	1194
239	0110	1196	127	192
240	0110	1197	1196	1195
241	0110	1198	127	192

```
242 | 0110 1199 127 1195
243 | 0001 1200 1198 1199
244 | 0110 1201 127 1200
245 | 0110 1202 128 193
246 | 0110 1203 1202 1201
247 | 0110 1204 128 193
248 | 0110 1205 128 1201
249 | 0001 1206 1204 1205
250 | 0110 1207 128 1206
251 | 0110 1208 129 194
252 | 0110 1209 1208 1207
253 | 0110 1210 129 194
254 | 0110 1211 129 1207
255 | 0001 1212 1210 1211
256 | 0110 1213 129 1212
257 | 0110 1214 130 195
258 | 0110 1215 1214 1213
259 | 0110 1216 130 195
260 | 0110 1217 130 1213
261 | 0001 1218 1216 1217
262 | 0110 1219 130 1218
263 | 0110 1220 131 196
264 | 0110 1221 1220 1219
265 | SHIFT OUT ALICE 1035
266 | SHIFT OUT ALICE 1041
267 | SHIFT OUT ALICE 1047
268 | SHIFT OUT ALICE 1053
269 | SHIFT OUT ALICE 1059
270 | SHIFT OUT ALICE 1065
271 | SHIFT OUT ALICE 1071
272 | SHIFT OUT ALICE 1077
273 | SHIFT OUT ALICE 1083
274 | SHIFT OUT ALICE 1089
275 | SHIFT OUT ALICE 1095
276 | SHIFT OUT ALICE 1101
277 | SHIFT OUT ALICE 1107
278 | SHIFT OUT ALICE 1113
279 | SHIFT OUT ALICE 1119
280 | SHIFT OUT ALICE 1125
281 | SHIFT OUT ALICE 1131
282 | SHIFT OUT ALICE 1137
283 | SHIFT OUT ALICE 1143
284 | SHIFT OUT ALICE 1149
285 | SHIFT OUT ALICE 1155
286 | SHIFT OUT ALICE 1161
287 | SHIFT OUT ALICE 1167
288 | SHIFT OUT ALICE 1173
289 | SHIFT OUT ALICE 1179
290 | SHIFT OUT ALICE 1185
291 | SHIFT OUT ALICE 1191
```

```
292 SHIFT OUT ALICE 1197
293 SHIFT OUT ALICE 1203
294 SHIFT OUT ALICE 1209
295 SHIFT OUT ALICE 1215
296 SHIFT OUT ALICE 1221
297 RETURN -1 xxx
```

Appendix B - PCF 2 file with adder function

```
 1 (LABEL :STR "pcfentry" )
 2 (INITBASE :BASE 1 )
 3 (LABEL :STR "main" )
 4 (CLEAR :LOCALSIZE 128 )
 5 (CONST :DEST 129 :OP1 0 )
 6 (BITS :DEST (129 130 131 132 133 134 135 136 137 138 139 140 141 142
      143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159
      160) :OP1 129 )
 7 (CONST :DEST 161 :OP1 65 )
 8 (MKPTR :DEST 161 )
 9 (COPY :DEST 162 :OP1 129 :OP2 32 )
10 (CALL :NEWBASE 194 :FNAME "alice" )
11 (INDIR-COPY :DEST 161 :OP1 194 :OP2 32 )
12 (CONST :DEST 227 :OP1 1 )
13 (MKPTR :DEST 227 )
14 (CONST :DEST 228 :OP1 65 )
15 (MKPTR :DEST 228 )
16 (COPY-INDIR :DEST 229 :OP1 228 :OP2 32 )
17 (INDIR-COPY :DEST 227 :OP1 229 :OP2 32 )
18 (CONST :DEST 261 :OP1 0 )
19 (BITS :DEST (261 262 263 264 265 266 267 268 269 270 271 272 273 274
      275 276 277 278 279 280 281 282 283 284 285 286 287 288 289 290 291
      292) :OP1 261 )
20 (CONST :DEST 293 :OP1 97 )
21 (MKPTR :DEST 293 )
22 (COPY :DEST 294 :OP1 261 :OP2 32 )
23 (CALL :NEWBASE 326 :FNAME "bob" )
24 (INDIR-COPY :DEST 293 :OP1 326 :OP2 32 )
25 (CONST :DEST 359 :OP1 33 )
26 (MKPTR :DEST 359 )
27 (CONST :DEST 360 :OP1 97 )
28 (MKPTR :DEST 360 )
29 (COPY-INDIR :DEST 361 :OP1 360 :OP2 32 )
30 (INDIR-COPY :DEST 359 :OP1 361 :OP2 32 )
31 (CONST :DEST 393 :OP1 1 )
32 (MKPTR :DEST 393 )
33 (COPY-INDIR :DEST 394 :OP1 393 :OP2 32 )
34 (CONST :DEST 426 :OP1 33 )
35 (MKPTR :DEST 426 )
36 (COPY-INDIR :DEST 427 :OP1 426 :OP2 32 )
```

```
37 | (CONST :DEST 492 :OP1 0 )
38 | (GATE :DEST 493 :OP1 427 :OP2 394 :TRUTH-TABLE #*0110 )
39 | (GATE :DEST 459 :OP1 493 :OP2 492 :TRUTH-TABLE #*0110 )
40 | (GATE :DEST 493 :OP1 427 :OP2 394 :TRUTH-TABLE #*0110 )
41 | (GATE :DEST 494 :OP1 492 :OP2 427 :TRUTH-TABLE #*0110 )
42 | (GATE :DEST 495 :OP1 493 :OP2 494 :TRUTH-TABLE #*0001 )
43 | (GATE :DEST 492 :OP1 427 :OP2 495 :TRUTH-TABLE #*0110 )
44 | (GATE :DEST 493 :OP1 428 :OP2 395 :TRUTH-TABLE #*0110 )
45 | (GATE :DEST 460 :OP1 493 :OP2 492 :TRUTH-TABLE #*0110 )
46 | (GATE :DEST 493 :OP1 428 :OP2 395 :TRUTH-TABLE #*0110 )
47 | (GATE :DEST 494 :OP1 492 :OP2 428 :TRUTH-TABLE #*0110 )
48 | (GATE :DEST 495 :OP1 493 :OP2 494 :TRUTH-TABLE #*0001 )
49 | (GATE :DEST 492 :OP1 428 :OP2 495 :TRUTH-TABLE #*0110 )
50 | (GATE :DEST 493 :OP1 429 :OP2 396 :TRUTH-TABLE #*0110 )
51 | (GATE :DEST 461 :OP1 493 :OP2 492 :TRUTH-TABLE #*0110 )
52 | (GATE :DEST 493 :OP1 429 :OP2 396 :TRUTH-TABLE #*0110 )
53 | (GATE :DEST 494 :OP1 492 :OP2 429 :TRUTH-TABLE #*0110 )
54 | (GATE :DEST 495 :OP1 493 :OP2 494 :TRUTH-TABLE #*0001 )
55 | (GATE :DEST 492 :OP1 429 :OP2 495 :TRUTH-TABLE #*0110 )
56 | (GATE :DEST 493 :OP1 430 :OP2 397 :TRUTH-TABLE #*0110 )
57 | (GATE :DEST 462 :OP1 493 :OP2 492 :TRUTH-TABLE #*0110 )
58 | (GATE :DEST 493 :OP1 430 :OP2 397 :TRUTH-TABLE #*0110 )
59 | (GATE :DEST 494 :OP1 492 :OP2 430 :TRUTH-TABLE #*0110 )
60 | (GATE :DEST 495 :OP1 493 :OP2 494 :TRUTH-TABLE #*0001 )
61 | (GATE :DEST 492 :OP1 430 :OP2 495 :TRUTH-TABLE #*0110 )
62 | (GATE :DEST 493 :OP1 431 :OP2 398 :TRUTH-TABLE #*0110 )
63 | (GATE :DEST 463 :OP1 493 :OP2 492 :TRUTH-TABLE #*0110 )
64 | (GATE :DEST 493 :OP1 431 :OP2 398 :TRUTH-TABLE #*0110 )
65 | (GATE :DEST 494 :OP1 492 :OP2 431 :TRUTH-TABLE #*0110 )
66 | (GATE :DEST 495 :OP1 493 :OP2 494 :TRUTH-TABLE #*0001 )
67 | (GATE :DEST 492 :OP1 431 :OP2 495 :TRUTH-TABLE #*0110 )
68 | (GATE :DEST 493 :OP1 432 :OP2 399 :TRUTH-TABLE #*0110 )
69 | (GATE :DEST 464 :OP1 493 :OP2 492 :TRUTH-TABLE #*0110 )
70 | (GATE :DEST 493 :OP1 432 :OP2 399 :TRUTH-TABLE #*0110 )
71 | (GATE :DEST 494 :OP1 492 :OP2 432 :TRUTH-TABLE #*0110 )
72 | (GATE :DEST 495 :OP1 493 :OP2 494 :TRUTH-TABLE #*0001 )
73 | (GATE :DEST 492 :OP1 432 :OP2 495 :TRUTH-TABLE #*0110 )
74 | (GATE :DEST 493 :OP1 433 :OP2 400 :TRUTH-TABLE #*0110 )
75 | (GATE :DEST 465 :OP1 493 :OP2 492 :TRUTH-TABLE #*0110 )
76 | (GATE :DEST 493 :OP1 433 :OP2 400 :TRUTH-TABLE #*0110 )
77 | (GATE :DEST 494 :OP1 492 :OP2 433 :TRUTH-TABLE #*0110 )
78 | (GATE :DEST 495 :OP1 493 :OP2 494 :TRUTH-TABLE #*0001 )
79 | (GATE :DEST 492 :OP1 433 :OP2 495 :TRUTH-TABLE #*0110 )
80 | (GATE :DEST 493 :OP1 434 :OP2 401 :TRUTH-TABLE #*0110 )
81 | (GATE :DEST 466 :OP1 493 :OP2 492 :TRUTH-TABLE #*0110 )
82 | (GATE :DEST 493 :OP1 434 :OP2 401 :TRUTH-TABLE #*0110 )
83 | (GATE :DEST 494 :OP1 492 :OP2 434 :TRUTH-TABLE #*0110 )
84 | (GATE :DEST 495 :OP1 493 :OP2 494 :TRUTH-TABLE #*0001 )
85 | (GATE :DEST 492 :OP1 434 :OP2 495 :TRUTH-TABLE #*0110 )
86 | (GATE :DEST 493 :OP1 435 :OP2 402 :TRUTH-TABLE #*0110 )
```

```
 87 (GATE :DEST 467 :OP1 493 :OP2 492 :TRUTH_TABLE #*0110 )
 88 (GATE :DEST 493 :OP1 435 :OP2 402 :TRUTH_TABLE #*0110 )
 89 (GATE :DEST 494 :OP1 492 :OP2 435 :TRUTH_TABLE #*0110 )
 90 (GATE :DEST 495 :OP1 493 :OP2 494 :TRUTH_TABLE #*0001 )
 91 (GATE :DEST 492 :OP1 435 :OP2 495 :TRUTH_TABLE #*0110 )
 92 (GATE :DEST 493 :OP1 436 :OP2 403 :TRUTH_TABLE #*0110 )
 93 (GATE :DEST 468 :OP1 493 :OP2 492 :TRUTH_TABLE #*0110 )
 94 (GATE :DEST 493 :OP1 436 :OP2 403 :TRUTH_TABLE #*0110 )
 95 (GATE :DEST 494 :OP1 492 :OP2 436 :TRUTH_TABLE #*0110 )
 96 (GATE :DEST 495 :OP1 493 :OP2 494 :TRUTH_TABLE #*0001 )
 97 (GATE :DEST 492 :OP1 436 :OP2 495 :TRUTH_TABLE #*0110 )
 98 (GATE :DEST 493 :OP1 437 :OP2 404 :TRUTH_TABLE #*0110 )
 99 (GATE :DEST 469 :OP1 493 :OP2 492 :TRUTH_TABLE #*0110 )
100 (GATE :DEST 493 :OP1 437 :OP2 404 :TRUTH_TABLE #*0110 )
101 (GATE :DEST 494 :OP1 492 :OP2 437 :TRUTH_TABLE #*0110 )
102 (GATE :DEST 495 :OP1 493 :OP2 494 :TRUTH_TABLE #*0001 )
103 (GATE :DEST 492 :OP1 437 :OP2 495 :TRUTH_TABLE #*0110 )
104 (GATE :DEST 493 :OP1 438 :OP2 405 :TRUTH_TABLE #*0110 )
105 (GATE :DEST 470 :OP1 493 :OP2 492 :TRUTH_TABLE #*0110 )
106 (GATE :DEST 493 :OP1 438 :OP2 405 :TRUTH_TABLE #*0110 )
107 (GATE :DEST 494 :OP1 492 :OP2 438 :TRUTH_TABLE #*0110 )
108 (GATE :DEST 495 :OP1 493 :OP2 494 :TRUTH_TABLE #*0001 )
109 (GATE :DEST 492 :OP1 438 :OP2 495 :TRUTH_TABLE #*0110 )
110 (GATE :DEST 493 :OP1 439 :OP2 406 :TRUTH_TABLE #*0110 )
111 (GATE :DEST 471 :OP1 493 :OP2 492 :TRUTH_TABLE #*0110 )
112 (GATE :DEST 493 :OP1 439 :OP2 406 :TRUTH_TABLE #*0110 )
113 (GATE :DEST 494 :OP1 492 :OP2 439 :TRUTH_TABLE #*0110 )
114 (GATE :DEST 495 :OP1 493 :OP2 494 :TRUTH_TABLE #*0001 )
115 (GATE :DEST 492 :OP1 439 :OP2 495 :TRUTH_TABLE #*0110 )
116 (GATE :DEST 493 :OP1 440 :OP2 407 :TRUTH_TABLE #*0110 )
117 (GATE :DEST 472 :OP1 493 :OP2 492 :TRUTH_TABLE #*0110 )
118 (GATE :DEST 493 :OP1 440 :OP2 407 :TRUTH_TABLE #*0110 )
119 (GATE :DEST 494 :OP1 492 :OP2 440 :TRUTH_TABLE #*0110 )
120 (GATE :DEST 495 :OP1 493 :OP2 494 :TRUTH_TABLE #*0001 )
121 (GATE :DEST 492 :OP1 440 :OP2 495 :TRUTH_TABLE #*0110 )
122 (GATE :DEST 493 :OP1 441 :OP2 408 :TRUTH_TABLE #*0110 )
123 (GATE :DEST 473 :OP1 493 :OP2 492 :TRUTH_TABLE #*0110 )
124 (GATE :DEST 493 :OP1 441 :OP2 408 :TRUTH_TABLE #*0110 )
125 (GATE :DEST 494 :OP1 492 :OP2 441 :TRUTH_TABLE #*0110 )
126 (GATE :DEST 495 :OP1 493 :OP2 494 :TRUTH_TABLE #*0001 )
127 (GATE :DEST 492 :OP1 441 :OP2 495 :TRUTH_TABLE #*0110 )
128 (GATE :DEST 493 :OP1 442 :OP2 409 :TRUTH_TABLE #*0110 )
129 (GATE :DEST 474 :OP1 493 :OP2 492 :TRUTH_TABLE #*0110 )
130 (GATE :DEST 493 :OP1 442 :OP2 409 :TRUTH_TABLE #*0110 )
131 (GATE :DEST 494 :OP1 492 :OP2 442 :TRUTH_TABLE #*0110 )
132 (GATE :DEST 495 :OP1 493 :OP2 494 :TRUTH_TABLE #*0001 )
133 (GATE :DEST 492 :OP1 442 :OP2 495 :TRUTH_TABLE #*0110 )
134 (GATE :DEST 493 :OP1 443 :OP2 410 :TRUTH_TABLE #*0110 )
135 (GATE :DEST 475 :OP1 493 :OP2 492 :TRUTH_TABLE #*0110 )
136 (GATE :DEST 493 :OP1 443 :OP2 410 :TRUTH_TABLE #*0110 )
```

```
137 (GATE :DEST 494 :OP1 492 :OP2 443 :TRUTH-TABLE #*0110 )
138 (GATE :DEST 495 :OP1 493 :OP2 494 :TRUTH-TABLE #*0001 )
139 (GATE :DEST 492 :OP1 443 :OP2 495 :TRUTH-TABLE #*0110 )
140 (GATE :DEST 493 :OP1 444 :OP2 411 :TRUTH-TABLE #*0110 )
141 (GATE :DEST 476 :OP1 493 :OP2 492 :TRUTH-TABLE #*0110 )
142 (GATE :DEST 493 :OP1 444 :OP2 411 :TRUTH-TABLE #*0110 )
143 (GATE :DEST 494 :OP1 492 :OP2 444 :TRUTH-TABLE #*0110 )
144 (GATE :DEST 495 :OP1 493 :OP2 494 :TRUTH-TABLE #*0001 )
145 (GATE :DEST 492 :OP1 444 :OP2 495 :TRUTH-TABLE #*0110 )
146 (GATE :DEST 493 :OP1 445 :OP2 412 :TRUTH-TABLE #*0110 )
147 (GATE :DEST 477 :OP1 493 :OP2 492 :TRUTH-TABLE #*0110 )
148 (GATE :DEST 493 :OP1 445 :OP2 412 :TRUTH-TABLE #*0110 )
149 (GATE :DEST 494 :OP1 492 :OP2 445 :TRUTH-TABLE #*0110 )
150 (GATE :DEST 495 :OP1 493 :OP2 494 :TRUTH-TABLE #*0001 )
151 (GATE :DEST 492 :OP1 445 :OP2 495 :TRUTH-TABLE #*0110 )
152 (GATE :DEST 493 :OP1 446 :OP2 413 :TRUTH-TABLE #*0110 )
153 (GATE :DEST 478 :OP1 493 :OP2 492 :TRUTH-TABLE #*0110 )
154 (GATE :DEST 493 :OP1 446 :OP2 413 :TRUTH-TABLE #*0110 )
155 (GATE :DEST 494 :OP1 492 :OP2 446 :TRUTH-TABLE #*0110 )
156 (GATE :DEST 495 :OP1 493 :OP2 494 :TRUTH-TABLE #*0001 )
157 (GATE :DEST 492 :OP1 446 :OP2 495 :TRUTH-TABLE #*0110 )
158 (GATE :DEST 493 :OP1 447 :OP2 414 :TRUTH-TABLE #*0110 )
159 (GATE :DEST 479 :OP1 493 :OP2 492 :TRUTH-TABLE #*0110 )
160 (GATE :DEST 493 :OP1 447 :OP2 414 :TRUTH-TABLE #*0110 )
161 (GATE :DEST 494 :OP1 492 :OP2 447 :TRUTH-TABLE #*0110 )
162 (GATE :DEST 495 :OP1 493 :OP2 494 :TRUTH-TABLE #*0001 )
163 (GATE :DEST 492 :OP1 447 :OP2 495 :TRUTH-TABLE #*0110 )
164 (GATE :DEST 493 :OP1 448 :OP2 415 :TRUTH-TABLE #*0110 )
165 (GATE :DEST 480 :OP1 493 :OP2 492 :TRUTH-TABLE #*0110 )
166 (GATE :DEST 493 :OP1 448 :OP2 415 :TRUTH-TABLE #*0110 )
167 (GATE :DEST 494 :OP1 492 :OP2 448 :TRUTH-TABLE #*0110 )
168 (GATE :DEST 495 :OP1 493 :OP2 494 :TRUTH-TABLE #*0001 )
169 (GATE :DEST 492 :OP1 448 :OP2 495 :TRUTH-TABLE #*0110 )
170 (GATE :DEST 493 :OP1 449 :OP2 416 :TRUTH-TABLE #*0110 )
171 (GATE :DEST 481 :OP1 493 :OP2 492 :TRUTH-TABLE #*0110 )
172 (GATE :DEST 493 :OP1 449 :OP2 416 :TRUTH-TABLE #*0110 )
173 (GATE :DEST 494 :OP1 492 :OP2 449 :TRUTH-TABLE #*0110 )
174 (GATE :DEST 495 :OP1 493 :OP2 494 :TRUTH-TABLE #*0001 )
175 (GATE :DEST 492 :OP1 449 :OP2 495 :TRUTH-TABLE #*0110 )
176 (GATE :DEST 493 :OP1 450 :OP2 417 :TRUTH-TABLE #*0110 )
177 (GATE :DEST 482 :OP1 493 :OP2 492 :TRUTH-TABLE #*0110 )
178 (GATE :DEST 493 :OP1 450 :OP2 417 :TRUTH-TABLE #*0110 )
179 (GATE :DEST 494 :OP1 492 :OP2 450 :TRUTH-TABLE #*0110 )
180 (GATE :DEST 495 :OP1 493 :OP2 494 :TRUTH-TABLE #*0001 )
181 (GATE :DEST 492 :OP1 450 :OP2 495 :TRUTH-TABLE #*0110 )
182 (GATE :DEST 493 :OP1 451 :OP2 418 :TRUTH-TABLE #*0110 )
183 (GATE :DEST 483 :OP1 493 :OP2 492 :TRUTH-TABLE #*0110 )
184 (GATE :DEST 493 :OP1 451 :OP2 418 :TRUTH-TABLE #*0110 )
185 (GATE :DEST 494 :OP1 492 :OP2 451 :TRUTH-TABLE #*0110 )
186 (GATE :DEST 495 :OP1 493 :OP2 494 :TRUTH-TABLE #*0001 )
```

```
187 (GATE :DEST 492 :OP1 451 :OP2 495 :TRUTH_TABLE #*0110 )
188 (GATE :DEST 493 :OP1 452 :OP2 419 :TRUTH_TABLE #*0110 )
189 (GATE :DEST 484 :OP1 493 :OP2 492 :TRUTH_TABLE #*0110 )
190 (GATE :DEST 493 :OP1 452 :OP2 419 :TRUTH_TABLE #*0110 )
191 (GATE :DEST 494 :OP1 492 :OP2 452 :TRUTH_TABLE #*0110 )
192 (GATE :DEST 495 :OP1 493 :OP2 494 :TRUTH_TABLE #*0001 )
193 (GATE :DEST 492 :OP1 452 :OP2 495 :TRUTH_TABLE #*0110 )
194 (GATE :DEST 493 :OP1 453 :OP2 420 :TRUTH_TABLE #*0110 )
195 (GATE :DEST 485 :OP1 493 :OP2 492 :TRUTH_TABLE #*0110 )
196 (GATE :DEST 493 :OP1 453 :OP2 420 :TRUTH_TABLE #*0110 )
197 (GATE :DEST 494 :OP1 492 :OP2 453 :TRUTH_TABLE #*0110 )
198 (GATE :DEST 495 :OP1 493 :OP2 494 :TRUTH_TABLE #*0001 )
199 (GATE :DEST 492 :OP1 453 :OP2 495 :TRUTH_TABLE #*0110 )
200 (GATE :DEST 493 :OP1 454 :OP2 421 :TRUTH_TABLE #*0110 )
201 (GATE :DEST 486 :OP1 493 :OP2 492 :TRUTH_TABLE #*0110 )
202 (GATE :DEST 493 :OP1 454 :OP2 421 :TRUTH_TABLE #*0110 )
203 (GATE :DEST 494 :OP1 492 :OP2 454 :TRUTH_TABLE #*0110 )
204 (GATE :DEST 495 :OP1 493 :OP2 494 :TRUTH_TABLE #*0001 )
205 (GATE :DEST 492 :OP1 454 :OP2 495 :TRUTH_TABLE #*0110 )
206 (GATE :DEST 493 :OP1 455 :OP2 422 :TRUTH_TABLE #*0110 )
207 (GATE :DEST 487 :OP1 493 :OP2 492 :TRUTH_TABLE #*0110 )
208 (GATE :DEST 493 :OP1 455 :OP2 422 :TRUTH_TABLE #*0110 )
209 (GATE :DEST 494 :OP1 492 :OP2 455 :TRUTH_TABLE #*0110 )
210 (GATE :DEST 495 :OP1 493 :OP2 494 :TRUTH_TABLE #*0001 )
211 (GATE :DEST 492 :OP1 455 :OP2 495 :TRUTH_TABLE #*0110 )
212 (GATE :DEST 493 :OP1 456 :OP2 423 :TRUTH_TABLE #*0110 )
213 (GATE :DEST 488 :OP1 493 :OP2 492 :TRUTH_TABLE #*0110 )
214 (GATE :DEST 493 :OP1 456 :OP2 423 :TRUTH_TABLE #*0110 )
215 (GATE :DEST 494 :OP1 492 :OP2 456 :TRUTH_TABLE #*0110 )
216 (GATE :DEST 495 :OP1 493 :OP2 494 :TRUTH_TABLE #*0001 )
217 (GATE :DEST 492 :OP1 456 :OP2 495 :TRUTH_TABLE #*0110 )
218 (GATE :DEST 493 :OP1 457 :OP2 424 :TRUTH_TABLE #*0110 )
219 (GATE :DEST 489 :OP1 493 :OP2 492 :TRUTH_TABLE #*0110 )
220 (GATE :DEST 493 :OP1 457 :OP2 424 :TRUTH_TABLE #*0110 )
221 (GATE :DEST 494 :OP1 492 :OP2 457 :TRUTH_TABLE #*0110 )
222 (GATE :DEST 495 :OP1 493 :OP2 494 :TRUTH_TABLE #*0001 )
223 (GATE :DEST 492 :OP1 457 :OP2 495 :TRUTH_TABLE #*0110 )
224 (GATE :DEST 493 :OP1 458 :OP2 425 :TRUTH_TABLE #*0110 )
225 (GATE :DEST 490 :OP1 493 :OP2 492 :TRUTH_TABLE #*0110 )
226 (GATE :DEST 493 :OP1 458 :OP2 425 :TRUTH_TABLE #*0110 )
227 (GATE :DEST 494 :OP1 492 :OP2 458 :TRUTH_TABLE #*0110 )
228 (GATE :DEST 495 :OP1 493 :OP2 494 :TRUTH_TABLE #*0001 )
229 (GATE :DEST 492 :OP1 458 :OP2 495 :TRUTH_TABLE #*0110 )
230 (COPY :DEST 491 :OP1 459 :OP2 32 )
231 (CALL :NEWBASE 523 :FNAME "output_alice" )
232 (LABEL :STR "$1" )
233 (RET :VALUE 523 )
```