Karlsruhe Institute of Technology

Master thesis

Using Predicated Instructions in Oblivious Data Structures

Jakob Nedlin

Date: 28. June 2020

Supervisors: Prof. Dr. Dennis Hofheinz
Prof. Dr. Peter Sanders

Institute of Theoretical Informatics, Algorithmics
Department of Informatics
Karlsruhe Institute of Technology

Abstract

Multi Party Computation (MPC) deals with the problem which arises if several parties that do not trust each other want to work together in order to jointly execute a program, without the input of the individual parties becomes known to each other. Algorithms which are calculated between these parties must usually be available as a boolean circuit. It is disadvantageous here that converting programs for RAM machines into boolean circuits is associated with a significant increase in program runtime. By means of predicated instructions, algorithms and programs written for regular RAM machines can be translated into efficient MPC programs. For this purpose, this thesis introduces a model in order to discuss certain optimization possibilities of the individual program fragments. Using a practical example, the minimal spanning tree problem, the developed approach is implemented, performance is determined and the result is compared with another known method for solving the problem. It is shown that the program runtimes can be reduced. In practice, the effectiveness strongly depends on the control-flow of the input program.

Bei sicherer Mehrparteienberechnung (multi party computation, MPC) arbeiten mehrere sich gegenseitig nicht vertrauende Parteien zusammen, um gemeinsam die Ausführung eines Programmes zu berechnen, ohne dass die Eingabe der einzelnen Parteien für die anderen bekannt wird. Damit Algorithmen zwischen diesen Parteien berechnet werden können, müssen die Algorithmen in der Regel als boolescher Schaltkreis vorliegen. Nachteilig ist hierbei, dass das Umwandeln von Programmen für RAM-Maschinen in boolesche Schaltkreise mit einer deutlichen Steigerung der Programmlaufzeit verbunden ist. Mit Hilfe von prädizierten Instruktionen lassen sich Algorithmen und Programme, die für reguläre RAM Maschinen geschrieben wurden, in effiziente MPC Programme übersetzen. Dazu wird in der vorliegenden Arbeit ein Modell eingeführt, um einzelne Optimierungsmöglichkeiten der Programmfragmente zu betrachten. An einem ausgewählten praktischen Beispiel, dem minimalen Spannbaum-Problem, wird dieser Ansatz umgesetzt, die Performance ermittelt und das Ergebnis mit einer weiteren bekannten Methode zum Lösen des Problems verglichen. Es kann gezeigt werden, dass sich durch dieses Vorgehen die Programmlaufzeiten reduzieren lassen. In der Praxis hängt die Effektivität jedoch stark vom Kontrollfluss des Eingabeprogramms ab.

Hiermit versichere ich, dass ich diese Arbeit selbständig verfasst und keine anderen, als die angegebenen Quellen und Hilfsmittel benutzt, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des Karlsruher Instituts für Technologie zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet habe.

Karlsruhe, den 28.6.2020

Contents

1 Introduction

1.1 Motivation

In secure multiparty computation (MPC) a set of parties not trusting each other are computing a function together with every party using a secret private input hidden from the other parties. These computation are called secure because only the process does not revealed any information about input data except for the result of the calculated function. Performing such computations without a trusted third party has been a topic in cryptography since 1982 and the introduction of the millionaires' problem by Yao [23].

In general computer science algorithms are often given in the form of a random access memory (RAM)-program. In cryptography there are two general approaches to transform such a RAM-program into an MPC protocol. Firstly, a dedicated MPC protocol can be developed for a specific application based on cryptographic primitives. Secondly, a compiler can be used which can turn any program into a MPC protocol. While the first usually offers better performance the security for the latter only has to be proven once for the compiler instead of every program individually.

A common approach for MPC compilers is to translate the input program into a circuit. Although optimizations have been proposed to reduce the runtime of these circuits there is still a blowup of $\mathcal{O}(T^3 \log T)$ in time when transforming T-time RAM programs into circuits [15].

The introduction of oblivious RAM for MPC which allows random access to a memory without leaking information about the access pattern enables RAM programs to be translated into MPC protocols directly. However, branching in programs that depend on secret input data is usually not allowed, since it could leak information about the private inputs [12]. So far this problem can be circumvented by performing an oblivious computation (implemented in [8]) which additionally hides the CPU state and the program that is being executed. The runtime overhead by using this technique is high because every possible CPU instruction will be calculated in every step.

1.2 Contribution

In this thesis, we will combine existing MPC tools with instruction predication in order to translate regular RAM programs into an MPC protocol while considering efficiency concerns. Compared to existing solutions we can allow branches depending on private inputs

within the program while achieving a better performance than using oblivious computation. As a result this approach is suitable in particular for algorithms with such branches which impact the runtime of the algorithm. We will also discuss possible optimization techniques for the resulting MPC programs.

As an example, we will apply our approach to create a MPC version of Kruskal's minimum spanning-tree algorithm. Then we will create an algorithm using an oblivious version of the union find data structure. We compared both of these approaches in practice by using a MPC framework implementing the SPDZ2k protocol [2].

1.3 Structure of Thesis

In the next chapter we will first introduce some basic definitions and notations used throughout the thesis. Then we will talk about related work. The following chapter will introduce the machine model and formal definition for our MPC compiler. Then we will discuss multiple possible optimization methods for improving the runtime of the MPC programs. Finally, in the following chapter a practical application of the compiler and optimization methods will be applied. In the last chapter we will discuss our results and given an outlook about this kind of approach for MPC algorithms.

2 Preliminaries

The following section will describe notations and concepts used in this thesis.

2.1 Oblivious data structures

In some situations we want to use data stored in a random access memory on a remote server, but at the same time can't trust this server and want to protect our data. To solve this problem the concept of oblivious RAM (ORAM) was introduced by Goldreich in 1987 [4]. In order to protect our data we have to address two problems:

(a) The content of the memory has to be hidden from the server.

(b) The access pattern to the RAM can be used by an adversary to gain information about our data.

A solution to problem (a) can be encrypting the data stored on the server. In order to hide information about the access pattern in problem (b) an ORAM must use an access pattern independent from the actual read and write operations on the RAM.

Different definitions for ORAM have been used in other works [4] [1] [18]. We will use the definition introduced by Shi et al. [18] since it fits our purpose best and generalizes this definition for any algorithm.

Definition 1 (Oblivious RAM)**.** An oblivious RAM (ORAM) is a set of interactive protocols between a server and a client. Let N be the capacity as the number of blocks which can be stored in the ORAM and B is the block size in number of bit in each block. Every block can be addressed by a unique global identifier $u \in \mathcal{U}$ with $\mathcal{U}$ being the set of identifiers. The ORAM has to support the following protocols:

- $\mathbf{ReadAndRemove}(u)$ The client can perform this interactive protocol using a private block identifier $u \in \mathcal{U}$ as input. The server receives and removes the block identified by u from the ORAM and returns the block content to the client. If the ORAM doesn't contain such a block a special symbol $\perp$ will be returned instead.
- $\mathbf{Add}(u, data)$ This protocol requires two private inputs from the client: an identifier $u \in \mathcal{U}$ and block content $data \in \{0,1\}^B$. The client performs this interactive protocol with the server in order to write the content $data$ to the block identified by u. This protocol always has to be preceded by the $\mathbf{ReadAndRemove}(u)$ protocol since we never want to have more than one content block for each identifier in our ORAM.

To match the definitions of a typical random access memory we define a $\mathbf{Get}$ and $\mathbf{Set}$ function based on the previous two protocols which we will use later. $\mathbf{Get}(u)$ invokes the $\mathbf{ReadAndRemove}(u)$ protocol followed by $\mathbf{Add}(u, data)$ where $data$ is the result of the first operation. Similar $\mathbf{Set}(u, data)$ invokes $\mathbf{ReadAndRemove}(u)$ and then $\mathbf{Add}(u, data)$ with the given input.

Additionally we want an ORAM to satisfy the security definition given below based on our definition for computational indistinguishability.

Definition 2 (Computational indistinguishability)**.** Two ensembles of random variables $X = \{x_1, x_2, \ldots, x_n\}$, $Y = \{y_1, y_2, \ldots, y_n\}$ are computationally indistinguishable if for every probabilistic polynomial time algorithm $\mathcal{D}$ a negligible function μ exists with the following property:

$$|Pr[\mathcal{D}(1^n, X) = 1] - Pr[\mathcal{D}(1^n, Y) = 1]| \leq \mu(n) \tag{2.1.1}$$

We call a function $\mu : \mathbb{N} \to \mathbb{R}$ negligible if $\exists c \in \mathbb{N}, n_0 \in \mathbb{N} : \forall n > n_0 : |\mu(n)| < n^{-c}$.

Definition 3 (ORAM Security definition)**.** Let $y := ((op_1, arg_1), (op_2, arg_2), \ldots, (op_M, arg_M))$ be a sequence of data request with length M. With each op_i denoting either a $\mathbf{ReadAndRemove}$ or an $\mathbf{Add}$ operation, arg_i denotes the argument of the operation with

$$\begin{aligned} arg_i &= u_i, && \text{if } op_i = \mathbf{ReadAndRemove} \\ arg_i &= (u_i, data_i), && \text{if } op_i = \mathbf{Add} \end{aligned}$$

As in definition 1 we define that an $\mathbf{Add}$ operation is always preceded by a $\mathbf{ReadAndRemove}$ operation. That means if $op_i = \mathbf{Add}$ then $op_{i-1} = \mathbf{ReadAndRemove}$ and $u_{i-1} = u_i$.

Let $\mathrm{ops}(y)$ be the sequence of operations $(op_1, op_2, \ldots, op_M)$ and $A(y)$ be the access sequence to the remote memory when performing a sequence of data requests y. We define an ORAM as secure if for any two data request sequences y, z with $|y| = |z|$ and $\mathrm{ops}(y) = \mathrm{ops}(z)$, their respective access patterns $A(y)$ and $A(z)$ are computationally indistinguishable by anyone but the client.

Goldreich and Ostrovsky [5] (extended by Larsen et al. [13]) proved $\Omega(\log(n))$ is a lower bound for the number accesses to the memory in each read and write operation. In practice, an ORAM using a binary-tree with a runtime in $\mathcal{O}(\log(n)^3)$ per RAM access with constant clientside storage has been proposed by Shi et al. [18].

We define oblivious algorithms similar to the ORAM definition: For any two inputs with equal length we want the memory accesses performed by our algorithm to be computationally indistinguishable.

Definition 4 (Oblivious algorithm)**.** Let $\mathcal{A}$ be an algorithm using a remote memory and $\tilde{\mathcal{A}}(x)$ be sequence of accesses to this memory performed by $\mathcal{A}$ with input x. We call $\mathcal{A}$ an oblivious algorithm if for any input i_1, i_2, $\tilde{\mathcal{A}}(i_1)$ and $\tilde{\mathcal{A}}(i_2)$ are computationally indistinguishable.

2.2 Secure multi-party computation

The ORAM model described in the previous section assumes a client is storing data on an untrusted server. In some use-cases we want to perform computation with more than two parties not trusting each other, keeping their inputs private. This is called secure multi-party computation (MPC). To perform algorithms between multiple untrusted parties the data shared between the parties has to be encrypted in a way that every participant has a share from the data. We will call such an encrypted data value *share*. In order to read the actual content from a share every party has to reveal its share to the other parties. This technique is called secret sharing.

There are different methods to implement secret-sharing most notably the additive scheme and Shamir's scheme [17]. These schemes use offline-phases where every parties is performing calculations on its on data and online-phases that requires communication between those parties. As a consequence some operation which can be performed offline like addition are fast while other operations like multiplication are significantly slower.

MPC primitives

When describing joint algorithms as MPC protocol we will use the following notation and primitives, summarized below in table 2.1.

Notation	Description
$[a]$	A shared secret (share) for the value of a
$[a] + [b]$ $[a] - [b]$ $[a] \cdot [b]$	Performs an arithmetic operation for the value of both shares and returns a share containing the result.
$[a] < [b]$	Returns a share $[c]$ with $c = 1$ if the value of a is smaller than the value of b and $c = 0$ otherwise.
$\mathbf{EQZ}([a])$	Performs an equal zero (EQZ) comparison returning a share $[c]$ with $c = 1$ if the value of a equals 0 and $c = 0$ otherwise.
$\mathbf{IfElse}([c], [a], [b])$	Shorthand for $[c] \cdot [a] + ([1] - [c]) \cdot [b]$

Table 2.1: Notation and primitives for MPC protocols

3 Related Work

To our knowledge this is the first work using predicated instruction to find efficient MPC programs. As such, there are no directly related works. For the general idea of using RAM programs in MPC there have been practical implementation such as by Marcel Keller in [8]. In this work a subset of the C program language is being introduced as a programming language for a multiparty RAM machine. The machine is based on secret-sharing and tree-based ORAM. It can execute programs by running every possible instruction on every step which also hides the program which is being executed from the adversary.

Our contribution is based off the ORAM implementations for MPC as given by Marcel Keller and Peter Scholl in [11]. These implementations are MPC versions of the ORAM introduced by Shi et al. in [18] and the Path ORAM by Stefanov et al. in [19]. Keller and Scholl further use this ORAM to implement a MPC version of Dijkstra's algorithms where the graph and source vertex are shared secrets across all parties. They also implemented their algorithms using SPDZ which is a nickname for the MPC protocol introduced by Damgard et al. in [3] and performed some experiments in their work.

In the experimental part of this thesis we use an implementation of the SPDZ$_{2^k}$ MPC protocol proposed by Cramer et al. in [2]. This protocol operates over the field of integers modulo 2^k and allows up to $n-1$ of n participating parties to be corrupted. Another notable MPC protocol with similar runtime cost using the same security model is the MASCOT protocol by Keller, Orsini and Scholl [10]. The MASCOT protocol (which stands for faster malicious arithmetic secure computation with oblivious transfer) relies on oblivious transfer in the preprocessing phase of the protocol in order to use symmetric cryptography which is faster than public-key cryptography used in other MPC protocols. The implementation of the SPDZ$_{2^k}$ protocol had slightly better runtime in a few practical tests for the online phase of the algorithm compared to MASCOT which has a faster offline phase. We decided to use SPDZ$_{2^k}$ over the MASCOT protocol since we compared the runtime of the online phases for our implementations.

Regarding the lower bound for oblivious data structures Riko Jacob et al. proofed in [7] $\Omega(\log n)$ as a lower bound for the expected amortized runtime of oblivious stacks, queues, deques, priority queues and search trees with n items.

For the particular problem of solving the minimum spanning tree problem in a parallel MPC setting a solution has been proposed by Peeter Laud in [14]. This approach uses a variation of Borůvka's minimum spanning tree (MST) algorithm as well as a batched oblivious array with read and write function which performs multiple read and write operations in amortized constant time.

4 Machine and Compiler Model

In this section we will introduce a generic abstract machine model which can execute programs. Then we will use this model to define a compiler that can translate programs into efficient MPC programs using predicated instructions which will be described later. We will also introduce a corresponding compiler optimization problem which consists of finding a MPC program with the fastest average runtime for a given input program using predicated instructions.

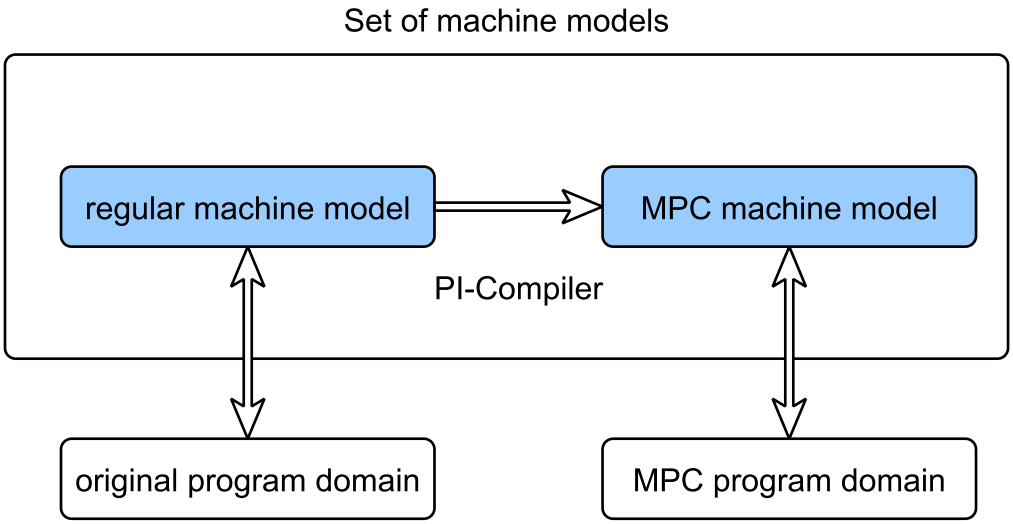

Figure 4.1: This diagram shows how we use the different representations of a program to transform a program from its original program domain into an MPC program domain. The double ended arrows indicate which practical program domains are represented by which theoretical machine model. The one sided arrow represents the predicated instruction compiler (PI-Compiler) described in definition 13.

In practice our input is a program in some arbitrary program domain (for example a programming language like C) and we want to translate our program into a MPC program domain such as the MP-SPDZ framework. This framework implements well known MPC protocols in a high level script scripting and we will late use it for performing runtime comparisons in the experimental part. Our primary goal is to describe a general approach

to efficiently translate programs not limited to a specific program domain. In order to do this we abstract from these specific practical implementations by representing both the original program domain and the MPC program domain as a machine model as described later in definition 7. The predicated instruction compiler describes how a program can be translated from the regular to the MPC machine model. This connection between the theoretical models and the practical program domains is also depicted in figure 4.1.

4.1 Terms

The machine model used for the definition of the compiler is an abstract model similar to the RAM machine for executing programs. We choose an abstract model which allows us to consider optimizations for this model representative for any specific model without the loss of generality. In particular a specific model in this context can be an abstract machine model itself for example the common RAM machine model.

There are two important parts in the abstract machine model: machine states and machine instructions. A program execution in our model is represented by successively applying machine instructions to an initial machine which contains the input data.

Definition 5 (Machine states)**.** A machine state (or program state) describes the current state of the whole memory which can be used by the machine while executing programs. This represents all memory cells used by the machine such as CPU registers, tapes or random access memory.

For the sake of simplicity and without loss of generality we define the set of possible machine states S to consist of four parts: input In, working memory Mem, program counter PC and output Out.

$$S = In \times Mem \times PC \times Out$$

The program counter PC should be a subset of the natural numbers. The definitions for other individual part are not limited by the abstract model. Although in general we assume that the machine state is a representation for the memory cells of the machine. For example, a working memory consisting of a RAM with 100 bytes of memory would be represented as $S = \mathbb{Z}_{256} \times \mathbb{Z}_{256} \times \cdots \times \mathbb{Z}_{256}$ (100 times) with $\mathbb{Z}_{256}$ denoting the set of integers modulo 256.

The program's input data will be stored within the input In of the machine state when the execution for a program begins. After the execution is finished the program's output will be stored in the Out part of the machine state.

We denote the access to the input In of a machine state m as m_{In}. Similar m_{Mem}, m_{PC} and m_{Out} denotes the access to the working memory, program counter and output of m, respectively.

Definition 6 (Machine instructions)**.** Let S be a set of machine states. We define a machine instruction $mi : S \mapsto S$ as a mathematical function mapping every previous machine state

to a new state. We call a machine instruction mi with $\forall m \in S : m_{PC} + 1 = mi(m)_{PC}$ a non-jump or regular machine instruction. All other instruction are called jump instructions.

Definition 7 (Machine model)**.** We define a (specific) machine model M representing a set of actual machines as a 3-tuple $M = (S, I, c)$. With S being the set of machine states, I being the set of machine instructions on the set of S which can be performed the represented machines and c being a cost function $c : I \mapsto \mathbb{N}$ which assigns costs to every machine instruction. This costs represent the runtime of an instruction for this model.

In general such a machine model is an abstraction for a program domain. A program domain denotes any environment which allows the execution of program which can be represented by our previous definitions of machine state and machine instructions. An example would be any imperative programming language: the commands used in such a language can be described as machine instructions and the memory of the programs at runtime can be described as machine state in our abstract model.

In our previous definitions specific machine models with very similar properties are treated as distinct models. We want to group similar models to find optimizations which work for such models. For example, we can consider different machine models using random access memory with different memory sizes. This way, we can create specific algorithms for various input sizes.

Definition 8 (Family of machine models)**.** We define a group of machine models $\mathcal{M}$ as family of machine models if all models provide the same set of instructions and there exist a surjective function $f : \mathbb{N} \mapsto \mathcal{M}$ which enumerates all models in the family. Usually we have additional information about the runtime behavior. For example if our function $f(n)$ enumerates machine models by different RAM sizes we know that the runtime of read and write operations to this RAM have polylogarithmic asymptotic growth with the RAM size n in a family representing MPC.

We will late use the input parameter for the enumerating function of a family of machine models to represent a public input parameter for MPC programs. For example, the input size of a program can be represented by this parameter.

As mentioned in definition 7 in order to describe a specific machine model for our abstract model we have to define these three components:

(i) The set of machine states S which represents the state of a program execution for our machine. This set describes the current state of the whole memory used by a machine between machine instructions.

(ii) The set of machine instructions I which describe all possible transition between machine states

(iii) A cost function c assigning a cost value for all machine instructions

Definition 9 (Program)**.** A program $\mathcal{P}$ in our model is defined as a list of machine instructions which is equivalent to a finite sequence of functions. The number of machine instruction in the sequence $\mathcal{P}$ is denoted as $\mathcal{P}_{max}$. The expected number of machine instructions which have to be executed in order to run a program $\mathcal{P}$ for an average input is denoted by $\#MI(\mathcal{P})$.

The execution of such a program in our model is represented by successively applying machine instructions to an input program state until an output state is reached. A machine instruction is a mathematical function which maps a previous program state to a new one.

A program $\mathcal{P}$ with a number of $\mathcal{P}_{max}$ instructions can be executed by applying every machine instruction consecutively to the current machine state. This is shown in algorithm 1. The applied machine instruction is determinated by the program counter which is part of the machine state. The initial state contains the input parameters for the program and the program counter is set to one (other values are chosen arbitrary). When the program counter is greater than the number of instructions in the program $\mathcal{P}_{max}$ the execution ends and the program is finished. The final machine state contains the algorithm's output.

Algorithm 1 $out \leftarrow \mathbf{RegularExecution}(\mathcal{P}, i)$
Executes program $\mathcal{P}$ for a given input i.

```
currentState ← create initial state for input i
pc ← 1
while pc <= P_max do
    mi ← P(pc)                          ▷ Fetch the next machine instruction mi from P
    currentState ← mi(currentState)     ▷ Apply mi to the current state
    pc ← currentState_PC                ▷ Get program counter from currentState
end while
return currentState_Out
```

In order to translate an input program into a MPC program we have to define the requirements for a machine model representing a MPC domain and we have to define the relation between MPC and regular machine models.

Definition 10 (MPC machine model)**.** We call a machine model $\hat{M}$ a MPC model if the program domain represented by $\hat{M}$ can be use to perform secure multiparty computations as described in section 2.2. In particular we want this MPC program domain to satisfy these properties:

(i) The data stored within the machine state represents shares which cannot be read by any strict subset of the involved parties except when all parties decide to unveil the value.

(ii) Every function represented by a machine instruction does not reveal any information about the value stored within the machine state.

This implies we can't use the program execution described in algorithm 1 in practice since the parties cannot read the program counter from the current machine state. The algorithm for executing programs in such a MPC model will be described in section 4.2.

Definition 11 (MPC convertable program)**.** We define a program in our model as MPC convertable if for the corresponding set of machine states S and set of used machine instructions I exists a MPC model $\hat{M} = (\hat{S}, \hat{I}, \hat{c})$ with $\hat{S}$ being isomorphic to S with an isomorphism $\varphi_s : S \mapsto \hat{S}$. In addition we want the set of machine instructions I to be isomorphic to $\hat{I}$ with an isomorphism $\varphi_i : (S \mapsto S) \mapsto (\hat{S} \mapsto \hat{S})$ which maps every machine instruction from the original model to an equivalent instruction in the MPC model by fulfilling the following property: $\forall ins \in I : \forall s \in S : \varphi_s\big(ins(s)\big) = \big(\varphi_i(ins)\big)(s)$.

The previous definition applies to programs for families of machine models if it holds for all machine models in this family and there is a family of machine models containing all corresponding MPC models.

In order to describe the oblivious program execution for our MPC model we define a formal $\mathbf{conditionalSet}$-function which takes the previous and the next machine state and a secret share of a bit b and return the previous state if $b = 0$ and the next state otherwise.

$$conditionalSet(b, prev, next) = \begin{cases} prev & b = 0 \\ next & b = 1 \end{cases} \tag{4.1.1}$$

Regarding the definition of $\mathbf{conditionalSet}$ you would expect a runtime asymptotic to the size of the underlying machine states. But we'll only use this function after a number of machine instructions were performed which are known in advance of the actual program's execution. And by doing this, we can implement this function in practice by conditionally applying the part of the machine state which can be changed by the preceding machine instructions. Thus we can assume an asymptotic runtime equal to the machine instructions mentioned.

4.2 Problem definition

Given an input program which fulfills our definition for a MPC convertable program the predicated instruction-compiler should be able to find the fastest MPC protocol implementing the input program. We can easily create a secure (but inefficient) output for any input program by using our required isomorphism φ_i and $\mathbf{conditionalSet}$. Instead of executing only one instruction at a time, we have to execute every instruction used by the input program in every step and combine it with our $\mathbf{conditionSet}$-function. Using this function allows us to only apply changes to the machine state if the current program counter points to the current machine instruction. This oblivious execution of a program is shown in algorithm 2 using the notation for MPC primitives described in table 2.1. Following this naive

approach we have to execute every machine instruction for every step of our program. The runtime T of the resulting program for an input program $\mathcal{P}$ with a set of machine instructions I is equal to the accumulated runtime of all machine instructions multiplied by the number of executed instructions. Denoting the cost function of the MPC machine model as $\hat{c}$ the runtime T of our MPC program execution is given by the following equation:

$$T = \#MI(\mathcal{P}) \cdot \sum_{i \in I} \hat{c}(i) \tag{4.2.1}$$

Depending on the size of the program it can be more efficient to execute all machine instructions given by the machine model. This idea was already implemented by Marcel Keller in [8] and also allows to hide the program being executed from the adversaries.

Since the value of the program counter in a MPC model is a shared secret between the involved parties we cannot read its value to determine the termination of a program without leaking additional information about the inputs. Instead we have to use an upper bound value which gives us the number of instructions executed before the program execution is stopped. In algorithm 2 we consider a program $\mathcal{P}$ for a family of machine models. For this reason we use an $upperBound$-function which gives us an upper bound for every machine model given by a public parameter n known to all parties which denotes what machine from the family of the machine models will be used.

Algorithm 2 $[out] \leftarrow \mathbf{ObliviousExecution}(\mathcal{P}, upperBound, i)$
Performs an oblivious execution of program $\mathcal{P}$ with $\mathcal{P}_{max}$ instructions with for a secret-shared input i.

```
currentState ← create MPC machine state for input i
pc ← [1]
for j ← 1 to upperBound(n) do
    for k ← 1 to P_max do                    ▷ Loop through all machine instructions
        mi ← P(k)
        nextState ← mi(currentState)
                ▷ Only apply mi to the machine state if the program counter point to k
        b ← EQZ(pc − [k])
        currentState ← conditionalSet(b, currentState, nextState)
        pc ← currentState_Out
    end for
end for
return currentState_Out
```

We can easily improve the runtime of the oblivious program execution by executing multiple successive non-jump machine instructions in a single step. This can be achieved by grouping these machine instructions together which would be part of the same vertex when constructing a control-flow graph of the program. If we do this, we still have to run

every machine instruction in our program on every step, but instead of one step for every single machine instruction we have one step per group. We call these groups of machine instructions predicated instruction.

Definition 12 (Predicated instruction program)**.** We define a predicated instruction program $\hat{\mathcal{P}}$ as finite sequence of finite sequences of machine instructions. Every sequence of machine instructions in $\hat{\mathcal{P}}$ is called a predicated instruction. The total number of sequences in $\hat{\mathcal{P}}$ is denoted as $\hat{\mathcal{P}}_{max}$.

Let $\mathcal{PI}$ be the set of predicated instructions used in our compiled program $\hat{\mathcal{P}}$ and $\#PI(\hat{\mathcal{P}})$ be the number of predicated instructions required until the execution of $\hat{\mathcal{P}}$ has finished. The total runtime T of the output program is given by

$$T = \#PI(\hat{\mathcal{P}}) \cdot \sum_{ins \in \mathcal{PI}} \sum_{i=1}^{ins_{max}} \hat{c}(ins(i)) \tag{4.2.2}$$

with ins_{max} being the number of machine instructions from the predicated instruction ins and $\hat{c}$ being the cost function of the MPC machine model.

An optimal compiler has to find a predicated instruction program $\hat{\mathcal{P}}$ which results in the fastest average-case runtime. We will first introduce two kinds of definitions for a predicated instruction compiler and then define the corresponding optimization problem.

Definition 13 (Predicated instruction compiler)**.** We define a predicated instruction compiler C between a machine model $\mathcal{M}$ and a MPC machine model $\hat{\mathcal{M}}$ (or two families of such models $\mathcal{M}$ and $\hat{\mathcal{M}}$) as a probabilistic polynomial time algorithm. Given a MPC convertable input program $\mathcal{P}$ for $\mathcal{M}$ C has to calculate a predicated instruction program $\hat{\mathcal{P}}$ as well as an $upperBound$ function in the MPC machine model $\hat{\mathcal{M}}$ (or all MPC machine models in the family $\hat{\mathcal{M}}$) using the isomorphisms φ_s, φ_i as described by definition 11. The output of compiler C has to fulfill the following requirements:

For all valid inputs i for $\mathcal{P}$ holds: If we run algorithm 1 with $\mathcal{P}$ and i as input and remember the last value of $currentState$ we can run the compilers output and successively apply the predicated instructions as shown in algorithm 3. Then we compare the output of the last state from the regular algorithm with the output from the last state of the oblivious algorithm. If we apply our ismorphism for machine states φ_s to the former the result has to be equal to the latter. This can be described by the following equations:

$$\begin{aligned} C_{\mathcal{M},\hat{\mathcal{M}}}(\mathcal{P}) &= (\hat{\mathcal{P}}, upperBound) \\ \forall i \in \text{input}(\mathcal{P}) : ls(\mathbf{PredicatedInstrExecution}, \hat{\mathcal{P}}, upperBound, i)_{Out} &= \varphi_s(ls(\mathbf{RegularExecution}, \mathcal{P}, i))_{Out} \end{aligned} \tag{4.2.3}$$

With input($\mathcal{P}$) denoting all possible inputs for program $\mathcal{P}$ and $ls(x, p1, p2, \dots)$ denoting the last machine state in the execution of algorithm x with parameters $p1, p2, \dots$. If the predicate instruction compiler is defined on a family of machine models the variable n denotes a public integer indicating which model of the family to use, otherwise n is omitted and $upperBound$ is a value instead of a function.

Algorithm 3 $[out] \leftarrow \mathbf{PredicatedInstrExecution}(\hat{\mathcal{P}}, upperBound, i)$
Performs an oblivious execution of a predicated instruction program $\hat{\mathcal{P}}$ with $\hat{\mathcal{P}}_{max}$ instructions with a given for a secret-shared input i.

$currentState \leftarrow$ create MPC machine state for input i
$pc \leftarrow [1]$
for $j \leftarrow 1$ **to** $upperBound(n)$ **do**
    ▷ In every step execute every machine instruction of every predicated instruction
  for $k \leftarrow 1$ **to** $\hat{\mathcal{P}}_{max}$ **do**
    $pi \leftarrow \hat{\mathcal{P}}(k)$
    $nextState \leftarrow currentState$
    for $l \leftarrow 1$ **to** pi_{max} **do**
      $mi \leftarrow pi(l)$
      $nextState \leftarrow mi(currentState)$
    end for
      ▷ Only apply pi to the machine state if the program counter point to k
    $b \leftarrow \mathbf{EQZ}(pc - [k])$
    $currentState \leftarrow \mathbf{conditionalSet}(b, currentState, nextState)$
    $pc \leftarrow currentState_{PC}$
  end for
end for
return $currentState_{Out}$

When we want to transfer a predicated instruction program $\hat{\mathcal{P}}$ into a program of the MPC program domain represented by the MPC machine model we will unroll the inner loops shown in algorithm 3 for $\hat{\mathcal{P}}$. This means in practice our program in the MPC program domain only contains a single loop we'll call execution loop which contains all machine instructions from our program $\hat{\mathcal{P}}$ in addition to some instructions to conditionally apply the machine instructions.

The extent of definition 13 is very wide since the output program $\hat{\mathcal{P}}$ has nothing related to the input expect producing the same result when using the same input. Since we expect the input program to have near optimal runtime we can state a more narrow definition.

Definition 14 (Simplified predicated instruction compiler)**.** We define an probabilistic polynomial time algorithm C as simplified PI-Compiler if it holds the definition of a PI-Compiler, but also fulfills additional constraints to the output predicated instruction program $\hat{\mathcal{P}}$. Let $\mathcal{P}$ be the input program and $\mathcal{P}_{max}$ the number of sequences in this program.

We require every sequence in $\hat{\mathcal{P}}$ to be a part of the input program $\mathcal{P}$ mapped by the isomorphism for machine instructions φ_i:

$$\forall ins \in \hat{\mathcal{P}} : \exists i \in [0..\mathcal{P}_{max}] : \forall j \in [0..ins_{max}] : ins(j) = \varphi_i(\mathcal{P}(i+j)) \tag{4.2.4}$$

Definition 15 (Compiler Optimization Problem)**.** We define the (simplified) predicated instruction compiler optimization problem based on the definition of the predicated instruction compiler. Given an MPC convertable program $\mathcal{P}$ for a single machine model or a family of models we want to find a predicated instruction program $\hat{\mathcal{P}}$ which holds the same requirements as the output of a predicated instruction compiler from definition 13 (and definition 14 respectively) with the minimal runtime across all possible outputs. The runtime of such an output is given in equation 4.2.2.

If we assume an input program with optimal runtime for this optimization problem the runtime of the solution from the simplified problem is at most a constant factor slower than the solution of the full problem. This is true since we can always create a solution that executes every machine instruction in each step and thus achieve a runtime equal to the runtime in equation 4.2.1.

4.3 Proof for non computability of the compiler optimization problem

Our generic model introduced in section 4.1 is able to simulate a given number of steps of a Turing machine. Using this property, we show that given an optimal compiler we can create an algorithm solving the halting-problem for a given Turing machine.

Theorem. *The compiler optimization problem as described in definition 15 is non computable.*

Proof. We want to proof the non computability of the compiler optimization problem by reducing it to the non computable halting problem [21]. We assume that the compiler optimization problem is computable and therefore a probabilistic polynomial time algorithm C exists which can solve this problem. Given such an optimal compiler C we can create a probabilistic polynomial time algorithm solving the halting-problem for any given Turing machine M and input x. We do this by running C for a family of programs with an integer parameter n enumerating this family. The program consisting of the following steps:

(i) Simulate n steps of Turing machine M with input x.

(ii) If m has halted return 1 otherwise return 0.

Our compiler C will always return a program with the best average-case runtime. As a consequence the machine instruction for "return 1" cannot appear in the output program if the Turing machine M will never halt with input x. Otherwise, we could construct a faster program by removing said machine instruction since it will never be executed.

The resulting program either contains a machine instruction for "return 1" or not. If this particular machine instruction exists in the output of C, our algorithm returns "M does eventually halt with input x", otherwise our algorithm returns "M will run indefinitely". This implies we can construct a Turing machine solving the halting problem. This is a contradiction to the fact that the halting problem is non computable. Therefore our assumption is false. □

The algorithm can be performed with a compiler for the full as well as the simplified problem. This concludes both problems are non computable. Notably, we can add more instructions in program described just before the "return 1" instruction. We can use this to show that even a compiler that is able to optimize the runtime to a fixed multiple of the optimal runtime is non computable.

5 Optimizations

In this section, we will discuss possible optimizations which can be applied by a predicated instruction compiler. We assume that we are given an existing predicated instruction program we want to optimize. This is always feasible since we can simply create predicated instruction from the control-flow graph of our input program as described in section 4.2. In general we want to minimize the time of the program execution by minimizing the time spend running predicated instructions which are not applied in the execution loop (the outer loop in algorithm 3). First. we do this by reducing the number of predicated instruction in the same loop. Then. we try to reduce the total runtime by removing costly machine instructions. Finally. we decrease the share spent running unnecessary predicated instructions by reordering the instructions.

5.1 Multiple instruction loops

Our first approach for improving the runtime of a program is splitting the single predicated instruction loop containing all instructions as shown in algorithm 3 into multiple smaller loops with fewer instructions. Since every loop contains less unnecessary instructions this results in a faster total runtime.

This approach is straight forward since we can only improve the runtime by doing this. But it can only be used if the program can be splitted, into multiple sequential phases with the requirement that the number of executed predicated instruction is known in advance. Every phase then results in a loop containing only the predicated instruction actually used within this phase.

5.2 Duplicated instruction removal

Another straight forward optimization approach is the removal of duplicated machine instructions. The execution of a compiled program requires us to run every instruction in every step whether or not this instruction is needed in this step of the program execution. Therefore we want to reduce the total program size as much as possible. Before we can remove a duplicated instruction we have to add a new common predicated instruction containing said instruction. However as a drawback this change increases the total number of

predicated instruction calls. An example how the instructions can be replaced is displayed in the following figure 5.1.

```
Predicated Instruction 1
    Instruction A
    Instruction C

Predicated Instruction 2
    Instruction B
    Instruction C
```

```
Predicated Instruction 1
    Instruction A
    pc <- Predicated
          Instruction 3

Predicated Instruction 2
    Instruction B
    pc <- Predicated
          Instruction 3

Predicated Instruction 3
    Instruction C
```

Figure 5.1: An example for replacing machine instructions with multiple occurrences. On the left side the original instructions are displayed. The right side shows the predicated instructions with machine instruction C replaced. An assignment to the program counter in the program state is denoted with *pc* $\leftarrow$.

Instead of performing an expensive machine instruction we create a new predicated instruction PI_{new} containing said machine instruction once and replace all other occurrences with a jump instruction to the new predicated instruction. In general, unless the replaced machine instruction is at beginning or end of the predicated instruction we also have to split the original predicated instruction after the replacement into two parts. In this case we need PI_{new} to include a write to the program counter pointing to second part of the original predicated instruction. This can be done by expending the MPC machine model to contain a dedicated register that contains the target program counter after instruction PI_{new} was executed. In practice this results in a constant memory overhead per replaced instruction.

Notably in the displayed example since we replace only two occurrences of instruction C the substitution shown in this example can never result in a faster runtime. This is obvious when we use the equation 4.2.2 for the required runtime. The number of predicated instruction calls is doubled since for each call two predicated instructions have to be executed. Let T_x the runtime of an instruction x (T_{pc} is the runtime for the assignment of the program counter) and $\#PI_y$ the average number of calls for the predicated instruction y. This example can be expressed though the following inequality 5.2.1.

$$(\#PI_1 + \#PI_2)(T_A + T_B + 2T_C) < 2(\#PI_1 + \#PI_2)(T_A + T_B + 2T_{pc} + T_C) \quad (5.2.1)$$

Generalization

In general we can check the impact of this optimization on the average runtime by using the following equation: Let $\#PI$ the average number of predicated instruction calls in the whole program and T_1 the runtime of a single iteration of our predicated instruction loop. We assume that in every iteration of the loop only a single predicated instruction can be executed which results in the total runtime T of the program being given by $T = \#PI \cdot T_1$. Let further $\widetilde{\#PI}$ be the average number of predicated instruction calls after our optimization with O_C be the overhead $\widetilde{\#PI} = \#PI + O_C$. Similar $\widetilde{T_1}$ denotes the runtime of a single loop iteration after the optimization was applied with O_T being the overhead such as $\widetilde{T_1} = T_1 + O_T$. Since we want to reduce the runtime of the loop we want the overhead O_T to be negative in order to be useful. Our optimization results in a shorter average runtime if it fulfills the following inequality:

$$\begin{aligned} \frac{\widetilde{\#PI} \cdot \widetilde{T_1}}{\#PI \cdot T_1} < 1 \Leftrightarrow \frac{(\#PI + O_C)(T_1 + O_T)}{\#PI \cdot T_1} < 1 \\ \Leftrightarrow 1 + \frac{O_C}{\#PI} + \frac{O_T}{T_1} + \frac{O_C O_T}{\#PI \cdot T_1} < 1 \end{aligned} \tag{5.2.2}$$

This can be simplified to the following condition:

$$O_C O_T + O_T \#PI + O_C T_1 < 0 \tag{5.2.3}$$

In order to determine the value of O_C we denote the set of predicated instructions that contain the duplicated instruction as I. We also denote with c_i the number of predicated instructions calls required to replace a single instruction call of instruction i after the replacement of machine instructions. This number c_i is depending on the number and position of the performed replacements. We can describe c_i using the following equation:

$$c_i = 2c_i^b + 3c^i + 2c^e \tag{5.2.4}$$

With $c_i^b = 1$ if the duplicated instruction is the first instruction in i and $c_i^b = 0$ otherwise. Similar $c_i^e = 1$ if the duplicated instruction appears at the end of i, otherwise $c_i^e = 0$. c^i denotes the number of occurrences of the replaced instruction which are neither at the beginning nor at the end of i. This results in the following formula:

$$O_C = \sum_{i \in I} c_i \#PI_i \tag{5.2.5}$$

Additionally in order to describe the value of O_T we denote the runtime of our replaced machine instruction as T_s. The total runtime overhead for writing the program counter and

the callback register is denoted as T_r. The value of T_r is depending on the position of the replaced machine instruction within the predicated instructions.

$$O_T = (1 - |I|)T_s + T_r \tag{5.2.6}$$

As mentioned before we need at least three occurrences with the same instruction to achieve a reduction of runtime in total. If the machine instruction that will be replaced is not at the beginning or the end of the predicated instruction the original predicated instruction has to be split into two parts which is executed before and after the new added instruction. Thus the number of predicated instruction calls for such an instruction triples.

This overhead limits the number of situation where the application of this optimization can successfully improve the runtime. On the other hand we can further improve the number of replaced instructions by not only looking for exact duplicates but instead also look for similar instruction.

For example, as mentioned in section 2.1 a reading or writing access to an oblivious RAM often has a polylogarithmic runtime to RAM size and takes in most cases noticeably more time than other machine instruction. We can imagine we are given an input set of predicated instruction where each instruction containing a RAM write instruction at the end we can group these together. This is possible even if these instructions are not exactly the same as they might use different registers for parameters of the write operation. In this case we can add additional registers to the working memory of MPC machine model which contain the parameters of the operation.

To further generalize this optimization we cannot only look at single instructions for replacement but also instruction sequences which are duplicated. Finding the optimal program using this optimization becomes vastly more difficult: In the single instruction case we can check every single machine instruction and performance a replacement regardless of the order of checking since the number of instruction calls doesn't change between replacements[1]. When we consider instruction sequences a replacement influences the number of occurrences for other instruction sequences. As a consequence an algorithm finding the optimal replacements for single instructions has linear runtime whereas such an algorithm for instruction sequences has exponential runtime. Although since the number of replacements that result in a better runtime is usually small it should be feasible to perform a complete search in order to find the best replacements especially for small programs.

5.3 Switch instruction order

The order of the predicated instructions has a great impact on the total runtime of the program. As an example we can assume four predicated instruction A, B, C, D with instruction

[1] Except for the number of calls to register write that are usually cheap

A ending with a jump to instruction D and D ending with a jump to either A or B (on average equally distributed) depending on the program state. We denote the runtime of our program with T. If we execute our program with the instruction order A-B-C-D it is easy to see that the instruction order A-D-B-C has a runtime of $\frac{1}{2}T$.

Since the search space of this problem is exponential in size to the number of predicated instruction it is unlikely that there is a polynomial time algorithm which finds the optimal order given a set of predicated instructions. Nevertheless there have been an optimization approach for a similar situation with SIMD (single instruction, multiple data) computations [16]. In the execution of a predicated program a single step whenever a predicated instruction has been executed is either useful and the result is applied to the machine state or the result is discarded. We want to construct a function W which takes an assignment for the order of predicated instructions as an input and calculates the average share of useful instructions. The idea is to maximize such a function W.

We construct this function W by looking at probabilities for one predicated instruction being followed another and creating a Markov chain model depending on this data. A Markov chain model describes a stochastic process with discrete states and time steps. Every state has probability for each state assigned to change to this state in the next time step. This is called transitions.

Let n denote the number of predicated instruction and m be the number of instructions in our execution loop. Our Markov model contains $n \cdot m$ states $S_{i,j}$. Such a state represents the program counter pointing to instruction i while the program execution is at position j within the execution loop.

We denote the assignment of instruction within the execution loop as a $n \times n$ matrix O with elements from $\{0, 1\}$ with $O_{ij} = 1$ denoting instruction i is placed at position j within the execution loop. Furthermore the probability of a jump from instruction i to instruction k is denoted as p_{ij}. $\vec{u}_i$ denotes the unit vector in the i-th dimension. These values can be estimated by sampling typical inputs for the program. The transition probabilities $T(a, b)$ from a state a to a state b are given by:

$$\forall i \in [1..n] : \forall j \in [1..m-1] : \forall k \in \{x \in [1..n] | x \neq i\} : T(S_{i,j}, S_{k,j+1}) = O_{ij} p_{ik} \tag{5.3.1}$$

$$\forall i \in [1..n] : \forall j \in [1..m-1] : T(S_{i,j}, S_{i,j+1}) = O_{ij} p_{ii} + (1 - O_{ij}) \tag{5.3.2}$$

$$\forall i \in [1..n] : \forall k \in \{x \in [1..n] | x \neq i\} : T(S_{i,m}, S_{k,1}) = O_{im} p_{ik} \tag{5.3.3}$$

$$\forall i \in [1..n] : T(S_{i,m}, S_{i,1}) = O_{im} p_{ii} + (1 - O_{im}) \tag{5.3.4}$$

All other transition probabilities are zero. This results in a sparse transition matrix M for our model which will be described using a number of n^2 submatrices M_{ij} with $m \times m$ elements.

$$
\begin{aligned}
M &= \begin{bmatrix}
T(S_{1,1}, S_{1,1}) & T(S_{1,2}, S_{1,1}) & \dots & T(S_{n,m}, S_{1,1}) \\
T(S_{1,1}, S_{1,2}) & T(S_{1,2}, S_{1,2}) & & \\
\vdots & & \ddots & \\
T(S_{1,1}, S_{n,m}) & & & T(S_{n,m}, S_{n,m})
\end{bmatrix} \\
&= \begin{bmatrix}
M_{11} & M_{21} & \dots & M_{n1} \\
M_{12} & M_{22} & & \\
\vdots & & \ddots & \\
M_{1n} & & & M_{nn}
\end{bmatrix}
\end{aligned}
\tag{5.3.5}
$$

$$
\begin{aligned}
\forall i \in [1..n] : \forall j \in \{x \in [1..n] | x \neq i\} : M_{ij} \\
= p_{ij} \begin{bmatrix}
0 & 0 & \dots & 0 & O_{im} \\
O_{i1} & 0 & & & \\
0 & O_{i2} & & & \\
\vdots & & \ddots & & \\
0 & & & O_{i,m-1} & 0
\end{bmatrix}
\end{aligned}
\tag{5.3.6}
$$

$$
\begin{aligned}
\forall i \in [1..n] : M_{ii} = p_{ii} & \begin{bmatrix}
0 & 0 & \dots & 0 & O_{im} \\
O_{i1} & 0 & & & \\
0 & O_{i2} & & & \\
\vdots & & \ddots & & \\
0 & & & O_{i,m-1} & 0
\end{bmatrix} \\
+ & \begin{bmatrix}
0 & 0 & \dots & 0 & 1 - O_{im} \\
1 - O_{i1} & 0 & & & \\
0 & 1 - O_{i2} & & & \\
\vdots & & \ddots & & \\
0 & & & 1 - O_{i,m-1} & 0
\end{bmatrix}
\end{aligned}
\tag{5.3.7}
$$

We can calculate a stationary distribution $t \in \mathbb{R}^{mn}$ for the presented model by solving the equation $Mt = t$ using the following help functions $h(x) = 1 + (m + x - 2) \bmod m$, $g(x, y) = h(x) + (y - 1)m$:

$$
\forall i \in [1..nm] : \left(\sum_{j=1}^{n} p_{j\lceil i/m \rceil} O_{j,h(i)} t_{g(i,j)} \right) + (1 - O_{\lceil i/m \rceil, h(i)}) t_{g(i, \lceil i/m \rceil)} = t_i \tag{5.3.8}
$$

We can calculate a share W in this distribution gives us the amount of work our algorithm actually spends performing useful instructions.

$$W = \sum_{i=1}^{n} \sum_{j=1}^{m} O_{ij} t_{g(i,j)} \tag{5.3.9}$$

Our goal is to find a matrix O which maximizes this share W while fulfilling the constraints given by equation 5.3.8. Unfortunately, calculating W for an input O involves solving a system of equations with nm equations. Thus, we have to rely on generic solutions for optimization problems such as hill climbing, simulated annealing or genetic algorithms.

Duplicating instructions

In contrast to the removal of duplicated machine instructions it can also be useful to duplicate a often used instruction with a short runtime on purpose. As mentioned before we want to maximize the time spend executing useful predicated instructions which are actually required in the current step of the programs execution. There are two possibilities to find good candidates for duplicated predicated instructions:

- An analytical analysis of the predicated instruction program can be performed.
- By doing an empirically analysis for typical program inputs

Compared to the execution of a regular program the runtime of an execution for a predicated instruction program slows down when we increase the number of instructions. Hence, the duplication of instructions must result in a reduction of the upper bound given by predicated instruction compiler in order to be useful. Looking at the model mentioned in the previous section we can expand it to include instruction duplication by using n as a target parameter to maximize W rather than a fixed value.

6 Application

We will apply our different MPC compiler techniques to the practical problem of finding the minimum-spanning tree (MST) of a graph in this section. We define minimum-spanning tree problem with the following definition.

Definition 16 (Minimum-Spanning Tree)**.** Given a connected graph $G = (V, E)$ and an edge weight function $w : E \mapsto \mathbb{R}$ we define a subset $E' \subseteq E$ as minimum-spanning tree if the graph $G' = (V, E')$ is connected and the total edge weight $w_T(E') = \sum_{i \in E'} w(i)$ of E' is minimal across all subset of E which form a connected graph.

Definition 17 (Minimum-Spanning Tree Problem)**.** Given a connected graph $G = (V, E)$ and an edge weight function $w : E \mapsto \mathbb{R}$ find a subset $E' \subseteq E$ which is a minimum-spanning tree for G and w.

6.1 Baseline algorithm

For a comparison to our predicated instruction based approach we will also introduce a solution similar to the contribution by Wang et al. [22]. The authors published an idea for creating an oblivious MPC data structure based on the functionality of the usual data structures and use these to translate an existing algorithm into an oblivious MPC algorithm. In this case we want to use an oblivious union-find data structure to build an oblivious version of Kruskal's algorithm.

Definition 18 (Union-Find data structure)**.** A union-find data structure represents a number of n sets and provides two functions.

(i) A $\mathbf{find}$ function which is given an element (an integer) and returns a representative for this set. Two elements are in the same set if the representatives returned by the $\mathbf{find}$ function are equal.

(ii) A $\mathbf{union}$ function which is given two representatives returned by $\mathbf{find}$ will join the two sets represented by the input.

Compared to common definition the number of sets is fixed and has to be known from the beginning. It doesn't provide a $\mathbf{makeSet}$-function to create new sets.

6.1.1 Oblivious Union-Find

The naive way of implementing a MPC union-find data structure is using an array arr of shares which holds the following invariant: $\forall i < n : \mathrm{arr}[i]$ is a share of the representative of i.

For every **find** and **union** operation we have to iterate through the whole array. The **find** operation uses an equal zero function (EQZ) as described in section 2.2 to select the correct element from the array without revealing any information about the requested element. This is displayed in algorithm 4. In a similar way the **Union** function shown in algorithm 5 replaces the corresponding representatives in the array. This results in a runtime of $\mathcal{O}(n)$ for each operation.

Algorithm 4 $[a] \leftarrow \mathbf{Find}([i])$ Find represent $[a]$ for a given element $[i]$

Require: $i \geq 0 \vee i < n$
  $[a] \leftarrow [0]$
  for $j \leftarrow 0$ **to** $n - 1$ **do**
    $[b] \leftarrow \mathbf{EQZ}([i] - [j])$
    $[a] \leftarrow [a] + [b] \cdot \mathrm{arr}[j]$
  end for
  return $[a]$

Algorithm 5 $\mathbf{Union}([a], [b])$ Joins the sets containing element a and b

  $[r_0] \leftarrow \mathbf{Find}([a])$
  $[r_1] \leftarrow \mathbf{Find}([b])$
  $[t] \leftarrow [r_0] < [r_1]$
  $([s_0], [s_1]) \leftarrow \mathbf{CondSwap}([t], [r_0], [r_1])$
  $[s] = [s_0] - [s_1]$
  for $i \leftarrow 0$ **to** $n - 1$ **do**
    $[u] \leftarrow \mathbf{EQZ}(\mathrm{arr}[i] - [s_0])$
    $\mathrm{arr}[i] \leftarrow \mathrm{arr}[i] + [u] \cdot [s]$
  end for

In order to improve the asymptotic runtime of this data structure we can replace the array of secret shares with an ORAM with size n. We assume such an ORAM provides a function $\mathbf{Get}$ and $\mathbf{Set}$ as described in definition 3. $\mathbf{Get}(p)$ with returns a share of the value at position given by an input share p. $\mathbf{Set}(p, v)$ changes the value at position p to a given value v. Both of these inputs are given as shares.

Using an ORAM arr allows us to directly random access any element in polylogarithmic runtime instead of iterating through the whole array. Using this property we can improve the asymptotic runtime by changing the invariant: Compared to the naive approach we want

the ORAM to represents a graph where every vertex v has a single successor $\mathrm{arr}(v)$. The representative of a vertex is found by following the successor until r is found that satisfies $\mathrm{arr}(r) = ([r], [c])$ with $[c]$ as height of the tree starting at root r. The number of successor to follow has an upper limit of $log_2(n)$. We can achieve the last property by using a union by rank operation. This means we will always attach the smaller trees to the larger ones.

The **Find**-operation is shown in algorithm 6. Using our invariant we have to follow the successors given by our ORAM $\lceil log_2(n) \rceil$-times. The **Union**-operation displayed in algorithm 7 replaces the representative of the smaller tree with a reference to the larger one. If the height of both trees is equal the selected representative for both subtree is selected arbitrary but the height of the chosen representative is increased by one.

Algorithm 6 $[a] \leftarrow \mathbf{Find}([i])$ Find represent $[a]$ for a given element $[i]$

Require: $i \geq 0 \vee i < n$
  $([r], [c]) \leftarrow \mathbf{Get}([i])$
  for $j \leftarrow 0$ **to** $\lceil log_2(n) \rceil$ **do**
    $([r], [c]) \leftarrow \mathbf{Get}([r])$
  end for
  return $([r], [c])$

Algorithm 7 $\mathbf{Union}([a], [b])$ Joins the sets containing element a and b

$([r_0], [c_0]) \leftarrow \mathbf{Find}([a])$
$([r_1], [c_1]) \leftarrow \mathbf{Find}([b])$
$[t] \leftarrow [c_0] < [c_1]$
$[r] \leftarrow \mathbf{IfElse}([t], [r_1], [r_0])$
$[c] \leftarrow \mathbf{IfElse}([t], [c_1], [c_0]) + \mathbf{EQZ}([c_1] - [c_0]) \cdot \mathbf{EQZ}([r_1] - [r_0])$
$\mathbf{Set}([r_0], ([r], [c]))$
$\mathbf{Set}([r_1], ([r], [c]))$

The runtime of these **Union** and **Find** operation now depend on the underlying ORAM's runtime. We denote the runtime of the ORAM as T_{Get} and T_{Set} for the **Get** and **Set** function respectively. The runtime of the **Find** operation is $\mathcal{O}(log(n)T_{Get})$ and $\mathcal{O}(log(n)T_{Set})$ for **Union**. When using a path ORAM proposed by Wang et al. [22] this results in a runtime of $\mathcal{O}(log(n)^4)$ for both **Union** and **Find**. Although the asymptotic runtime is better than the naive approach the constant factor is much larger so for small input the naive implementation should outperform the ORAM approach.

6.1.2 Blackbox Kruskal

Using one of the oblivious data structures described above we can write an oblivious version of Kruskal's algorithm using the union-find data structure as a black box. Kruskal's algorithm is a greedy algorithm. It finds the minimum spanning tree in two phases:

(i) The input edge array will be sorted by weight.

(ii) A union-find data structure will be used to successively create the minimum spanning tree.

The first phase can be solved by using known sorting algorithms which compare and swap elements within an array independent from the actual data. These algorithms are called sorting networks. This means as long as we are given a compare and swap operation which fulfills our definition for MPC we can easily construct a sorting program as MPC protocol. We tried if we can use our proposed predicated instruction approach in order to implement well-known RAM sorting algorithms as MPC programs. But preliminary experiments showed that an implementation of the quicksort algorithm using predicated instructions has longer runtime in practice than an implementation of the bitonic sorting network. This was not unexpected since the asymptotic runtime of a n-element bitonic sorting network is $\mathcal{O}(n \log^2 n)$ when executing the comparison sequentially and the expected asymptotic runtime of quicksort is $\mathcal{O}(n \log n)$ which is slowed down by a faction of $\log^3 n$ because of the runtime overhead for the ORAM access. Although in theory it could be still possible to find an ORAM algorithm with similar asymptotic runtime since so far only a lower bound of $\mathcal{O}(\log n)$ per ORAM access has been proven.

We will now assume our input array is already sorted and focus on the second phase on the algorithm.

In the second phase a union-find data structure with a new set for every vertex in the graph will be created. Then the algorithm will build the minimum spanning tree by beginning with an empty result array and walking through every sorted list of edges starting with the lowest weight edge. The algorithm will use the find-operation for the two vertices connected by the edge to check whether or not the vertices are within the same set in the union-find data structure. If both vertices are not within the same set the union-function will be called with both sets and the edge will be added to result set.

The second phase of Kruskal's algorithm is shown in algorithm 8. Here $\mathbf{UnionFind}(n)$ denotes the creation of a union-find data structure with n elements and $\mathbf{ObliviousArray}(n)$ denotes the creation of an ORAM with capacity n.

As described before the algorithm consists of a loop which will be executed for every index i of the edge array. In every step we perform two $\mathbf{Find}$ calls to get the representatives of both connected vertices connected by the current edge. Then we check whether or not they belong to the same set and store the result within share c. We can also call the $\mathbf{union}$ operation in every step because calling $\mathbf{union}$ for two representatives of the same set does not change the state of the union-find data structure. In every step we will write the current edge index into and output array out at index k then we will add c to index k. After executing the loop the first $n_v - 1$ elements of out array contains the indexes of the MST while the last element contains always index of the last edge and can be ignored. Since the number of $\mathbf{union}$ and $\mathbf{find}$ calls is fixed for every input size we don't have to hide which operation has been executed on the data structure.

Algorithm 8 out $\leftarrow$ **Kruskal**(vertices, edges, n_v, n_e)

Require: List of edges is sorted

```
u ← UnionFind(n_v)
out ← ObliviousArray(n_v)
[k] ← [0]
for i ← 0 to n_e − 1 do
    ([v_0], [v_1]) ← edges[i]
    [r_0] ← u.Find([v_0])
    [r_1] ← u.Find([v_1])
    u.Union([r_0], [r_1])
    [c] ← [1] − EQZ([r_0] − [r_1])
    out.Set([k], [i])
    [k] ← [k] + [c]
end for
return out
```

6.1.3 Runtime

The runtime T of this "Blackbox" Kruskal algorithm is mostly depending on the runtime of the **union** and **find**-operations. It's given by the following equation:

$$T = T_I + n_e(2T_F + T_U + T_{EQZ} + T_{Set} + T_+) \tag{6.1.1}$$

With T_I denoting the runtime of initialization of the **UnionFind** and **ObliviousArray** data structures, n_e being the number of edges, T_F, T_U, T_{EQZ}, T_{Set} and T_+ respectively denoting the runtime of a single **Find**, **Union**, **EQZ**, **Set** operation and an add operation between two shares.

Using the trivial union-find data structure this results in asymptotic runtime of $\mathcal{O}(n_v n_e)$ with n_v denoting the number of vertices and n_e denoting the number of edges. Compared to that using the second union-find implementation with an ORAM proposed by Wang et al. [22] the algorithm has an asymptotic runtime of $\mathcal{O}(log(n_v)^4 n_e)$.

6.2 Predicated instruction algorithm

As described in the previous section Kruskal's algorithm can be written as an algorithm without any branches depending on input data. Hence, with the discussed optimizations it is not possible to improve the runtime of the "blackbox" version of the algorithm using predicated instructions. We can however improve the runtime by considering the whole union-find data structure as part of the algorithm.

This way, we can use a union-find algorithm with path compression to improve the overall runtime of the algorithm. Using path compression means after each **find**-operation we will update the path of every visited vertex in our union-find data structure to point to the representing vertex directly. This saves search steps when using the **find**-operation multiple times on the same element. Similar to the union-find structure above we use an ORAM to store the representatives of a node. It is shown that a union find data structure using both union-by-rank and path compression results in a amortized runtime $\mathcal{O}(\alpha(n))$ with $\alpha(n)$ being the inverse Ackermann function [20] which is a function whose value grows very slow so that we can assume a value of 5 for all reasonable input sizes n.

In order to find predicated instructions for our algorithm we look at the control-flow graph of our algorithm. In order to better illustrate this we split the main loop of Kruskal's algorithm with integrated union-find into these steps:

(i) Read the next edge in the sorted edge array and put store the vertex ids into registers.

(ii) Perform a single step of the find-operation. A single step contains following the next edge the graph represented by representative ORAM. Within this step we will also perform a path compression step.

(iii) Check whether or not the find-operation is finished.

(iv) Write the edge into the output array

(v) Perform a union-operation

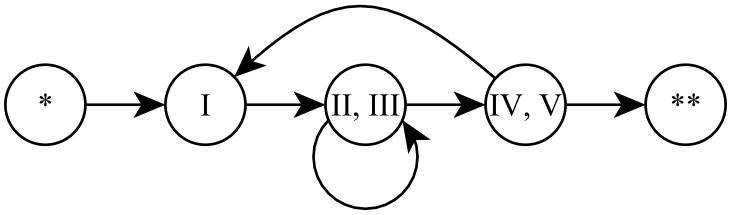

Figure 6.1: Control-flow graph for the main loop of Kruskal's algorithm. Every vertex represents a set of code that will be executed together. A directed edge means the code of the vertex being pointed at can be next to be executed after the code in the source vertex. This could be cause by either a conditional jump between those code segments or because there are non-jump instructions placed in succession.

The implementation of these steps are described respectively in algorithms 9, 10, 11, 12 and 13. We will use these algorithms to construct our predicated instructions. As such the first parameter in each algorithm is a share of a bit c which is $c = 1$ if this predicated instruction should be applied in this step otherwise $c = 0$ (except for algorithm 11 which doesn't use any writing function). This is a practical implementation of the **conditionalSet**-function in the abstract execution loop displayed in algorithm 3.

Figure 6.1 displays the control-flow of Kruskal's algorithm. The labels of the vertex describe which of the previous listed steps will be executed in this step of the algorithm. The vertex labeled with $*$ represents the initialization of data structures and the $**$ vertex represents some additional code for formatting the output. Since the graph has three vertices for the main loop of Kruskal's algorithm we will use three predicated instructions in our MPC algorithm. Every predicated instruction consists of the steps shown by the labels in figure 6.1.

The resulting algorithm is shown in algorithm 14. The main loop starting in line 13 contains these three predicated instructions. Each instruction begins with the line $[c] \leftarrow \mathbf{EQZ}([pi]-[x])$ where x is the number of said instruction. The bit share $[c]$ decides whether or not the following predicated instruction should be active in this step of the algorithm. As before the algorithm will successively go through the sorted edge array. The find operation will be performed with the two representatives for both vertices adjacent to the current edge at the same time. Notably the number of loops is limited by the inverse Ackermann function times the number of edges in the input graph. As shown by Harfst and Reingold in [6] the amortized cost for a single $\mathbf{find}$-operation is $\alpha(m,n)+3$ with m being the total number of operation on a n-element union-find data structure. That's why we can chose $l = n_e * 3 * (a(n_e, n_v) + 3)$ as limit for the number of loop iterations since we perform n_e operations on a union-find data structure of size n_v. The additional factor 3 is caused by the number of predicated instructions since at least $\frac{1}{3}$ of all iteration execute the relevant first predicated instruction.

This algorithm has a better asymptotic runtime than the "blackbox" version of Kruskal's algorithm. However, in practice it is slower since it has a very large constant factor. This is caused by the high number of costly accesses to the ORAM.

Algorithm 9 $([v_1],[v_2]) \leftarrow \mathbf{GetNextEdge}([c], \text{edgeArray}, [nextEdge], [v_1], [v_2])$

Require: $c = 0 \vee c = 1$

Ensure: Returns an edge from edgeArray with index $nextEdge$ if $c = 1$ otherwise the input parameters $([v_1],[v_2])$ will be returned.

$([\widetilde{v_1}],[\widetilde{v_2}]) \leftarrow \text{edgeArray}.\mathbf{Get}([nextEdge])$

$[v_1] = \mathbf{IfElse}([c],[\widetilde{v_1}],[v_1])$

$[v_2] = \mathbf{IfElse}([c],[\widetilde{v_2}],[v_2])$

return $([v_1],[v_2])$

Algorithm 10 $\mathbf{ConditionalPathCompression}(\mathrm{ufArray}, [cond], [elem])$

Ensure: If $cond = 1$ a single path compression step will be performed on ufArray for element $[elem]$, otherwise ufArray remains unchanged.

$[represent] \leftarrow \mathrm{ufArray}.\mathbf{Get}([elem])$
$[successor] = \mathrm{ufArray}.\mathbf{Get}([represent])$
$[successor] \leftarrow \mathbf{IfElse}([cond], [successor], [represent])$
$\mathrm{ufArray}.\mathbf{Set}([elem], [succesor])$

Algorithm 11 $[out] \leftarrow \mathbf{IsRepresentative}(\mathrm{ufArray}, [index])$

Ensure: $out = 1$ if the element at index $[index]$ is the representative of their group, otherweise $out = 0$.

$[represent] \leftarrow \mathrm{ufArray}.\mathbf{Get}([element])$
$[out] \leftarrow \mathbf{EQZ}([represent] - [element])$
return $[out]$

Algorithm 12 $\mathbf{CondSet}(\mathrm{arr}, [cond], [idx], [val])$

$[val_{old}] \leftarrow \mathrm{arr}.\mathbf{Get}([idx])$
$[val_{new}] \leftarrow \mathbf{IfElse}([cond], [val], [val_{old}])$
$\mathrm{arr}.\mathbf{Set}([idx], [val_{new}])$

Algorithm 13 $[out] \leftarrow \mathbf{ConditionalUnion}(\mathrm{uf}, [cond], [r_1], [r_2])$

$[out] \leftarrow [1] - \mathbf{EQZ}([r_1] - [r_2])$
$\mathbf{CondSet}(\mathrm{uf}, [cond], [r_2], [r_1])$
return $[out]$

Algorithm 14 Kruskal (predicated instructions)
$\text{out} \leftarrow \mathbf{Kruskal}(\text{vertices}, \text{edges}, n_v, n_e)$

Require: List of edges is sorted
Ensure: out is an array of share with $out[e] = [1]$ if $\text{edes}[e]$ is part of the calculated mst and $out[e] = [0]$ otherwise.

1: $\text{out} \leftarrow \mathbf{ObliviousArray}(n_v - 1)$
2: $\text{uf} \leftarrow \mathbf{ObliviousArray}(n_v)$ ▷ Initialize Union-Find Array
3: **for** $i \leftarrow 0$ **to** $n_v - 1$ **do**
4: $\quad \text{uf}.\mathbf{Set}([i], [i])$
5: **end for**
6: $[pi] \leftarrow [1]$
7: $([v_1], [v_2]) \leftarrow ([0], [0])$
8: $[nextEdge] \leftarrow [0]$
9: **for** $l \leftarrow 0$ **to** $2(\alpha(n_e, n_e) + 3)$ **do**
▷ Get next edge in sorted array
10: $\quad [c] \leftarrow \mathbf{EQZ}([pi] - [1])$
11: $\quad ([v_1], [v_2]) \leftarrow \mathbf{GetNextEdge}([c], \text{edgeArray}, [nextEdge], [v_1], [v_2])$
12: $\quad [pi] \leftarrow [pi] + [c]$
▷ Perform find operation on union find data structure
13: $\quad [c] \leftarrow \mathbf{EQZ}([pi] - [2])$
14: $\quad \mathbf{ConditionalPathCompression}(\text{uf}, [c], [v_1])$
15: $\quad \mathbf{ConditionalPathCompression}(\text{uf}, [c], [v_2])$
16: $\quad [r_1] \leftarrow \text{uf}.\mathbf{Get}([v_1])$
17: $\quad [r_2] \leftarrow \text{uf}.\mathbf{Get}([v_2])$
18: $\quad [nextPhase] \leftarrow \mathbf{IsRepresentative}(\text{uf}, [r_1]) \cdot \mathbf{IsRepresentative}(\text{uf}, [r_2])$
19: $\quad [pi] \leftarrow [pi] + [c] \cdot [nextPhase]$
▷ Perform union operation and set output bit
20: $\quad [c] \leftarrow \mathbf{EQZ}([pi] - [3])$
21: $\quad [inMST] \leftarrow \mathbf{ConditionalUnion}(\text{uf}, [c], [r_1], [r_2])$
22: $\quad \mathbf{CondSet}(\text{out}, [c], [inMST])$
23: $\quad [pi] \leftarrow \mathbf{IfElse}([c], [1], [pi])$
24: **end for**
25: **return** $[out]$

We will now discuss how we can optimize the algorithm by using the optimizations described in section 5.

First we can reduce the number of predicated instructions by merging together the vertex containing step iv, v and the vertex containing step i since these steps will always be executed successively (except for the first and last step execution). We can do this by splitting the algorithm into two phases. The first phase only contains a single execution of step i and is appended after the initialization phase. It is followed by a second phase containing

the two remaining predicated instructions. Since we now only have two different predicated instructions at most we cannot achieve much runtime improvement by reordering the predicated instructions as explained in section 5.3. Instead we can reduce the number of expensive array read and write operations. We do this by introducing a new function which combines step ii and iii. This function is shown in algorithm 15.

Algorithm 15 $([c], [r_{out}]) \leftarrow \mathbf{FindStep}(\text{ufArray}, [r_{in}])$

$[r_{out}] \leftarrow \text{ufArray}.\mathbf{Get}([r_{in}])$
$[c] \leftarrow \mathbf{EQZ}([r_{in}] - [r_{out}])$
$[successor] = \text{ufArray}.\mathbf{Get}([r_{out}])$
$\text{ufArray}.\mathbf{Set}([r_{in}], [succesor])$
return $([c], [r_{out}])$

The resulting algorithm is shown in algorithm 16. Since there are only two predicated instructions and by using the property that we can write into the output array as long as the last written value is the correct value we can omit the register pi which stores the program counter.

Algorithm 16 Kruskal (optimized)
$\text{out} \leftarrow \mathbf{Kruskal}(\text{vertices}, \text{edges}, n_v, n_e)$

Require: List of edges is sorted

1: $\text{out} \leftarrow \mathbf{ObliviousArray}(n_v - 1)$
2: $\text{uf} \leftarrow \mathbf{ObliviousArray}(n_v)$ ▷ Initialize Union-Find Array
3: **for** $i \leftarrow 0$ **to** $n_v - 1$ **do**
4: $\quad \text{uf}.\mathbf{Set}([i], [i])$
5: **end for**
6: $([r_1], [r_2]) \leftarrow \text{edges}.\mathbf{Get}([0])$
7: $([c_0], [c_1]) \leftarrow ([0], [0])$
8: $[nextEdge] \leftarrow [0]$
9: **for** $l \leftarrow 0$ **to** $2(\alpha(n_e, n_e) + 3)$ **do**
10: $\quad ([c_0], [r_0]) \leftarrow \mathbf{FindStep}(\text{uf}, [r_0])$
11: $\quad ([c_1], [r_1]) \leftarrow \mathbf{FindStep}(\text{uf}, [r_1])$
12: $\quad [c] \leftarrow [c_0] \cdot [c_1]$
13: $\quad \mathbf{CondSet}(\text{uf}, [c], [r_0], [r_1])$
14: $\quad [j] \leftarrow [j] + [c]$
15: $\quad [c] \leftarrow [c] \cdot ([1] - \mathbf{EQZ}([r_0] - [r_1]))$
16: $\quad \text{out}.\mathbf{Set}([k], [j] - [1])$
17: $\quad [k] \leftarrow [k] + [c]$
18: **end for**
19: **return** $[out]$

6.3 Experiments

In order to compare the algorithms in practice we implemented all algorithms using a MPC framework published by the Data61 group [9]. This framework allows us to write MPC algorithms in a custom high-level language based on python. This high-level language will be translated into a special byte-code format. Then the byte-code format will be used to execute the input program with multiple parties together. In our test version 0.1.3 of the MPC framework was used on a test system with a 2.30GHz dual-core CPU (Intel i5-6200U) and 8 GB of RAM.

The framework provides an implementation for a simple linear ORAM as well as a path ORAM. Whenever we used an ORAM in our implementation we use the "OptimalORAM"-class which automatically selects the ORAM implementation with the fastest runtime depending on the ORAM size (ORAMs smaller than 10000 elements use the linear ORAM while using path ORAM otherwise).

First we compared the runtime of the described Union-Find data structures. For this comparison we ran a multiple tests with different number of sets. Although the test data should not have any effect on the runtime in an MPC program, we note that our data was generated by selecting equally distributed parameters for the tested operations at compile-time of the high-level language using the default python random number generator. The runtime of a union and a find operation combined is shown as a graph in figure 6.2. Although the idea behind our trivial Union-Find data structure and the Union-Find data structure using linear ORAM is fairly similar the first achieves a faster runtime than the latter only for union operations at very small input sizes. This behavior is not expected for linear ORAM but for path ORAM which is only useful for large data size as mentioned before. We expect the reason for this behavior to be the internal usage of a sophisticated bit decomposition function by the linear ORAM which eliminates the need to perform a comparison for every set per operation required in the trivial Union-Find implementation. This could be also an explanation for the steep jump at an input size of 32 and 64.

Next we compared our different implementations for the MST problem. Similar to the comparison of union find implementations the test data was generated at compile-time of the high-level language. The input (a fully connected graph with n vertices and m edges with $n \leq m \leq n^2$) is generated in the following manner: Starting with a n unconnected vertices we select two random vertex and connect them. Then we will select a random unconnected vertex and connect them to a random already connected vertex. We repeat this until all vertices are connected. Finally we connect random vertices which are not yet connected until the number of edges is m. This set of edges will be written into an array using random order. All random numbers are chosen using the default python random number generator. Since there is no support for arrays storing tuples the input graph is given as a tuple of two ORAMs containing one of the adjacent vertices for every edge each. In the practical implementation of predicated instruction MST the inverse Ackermann's function α is being calculated at compile time.

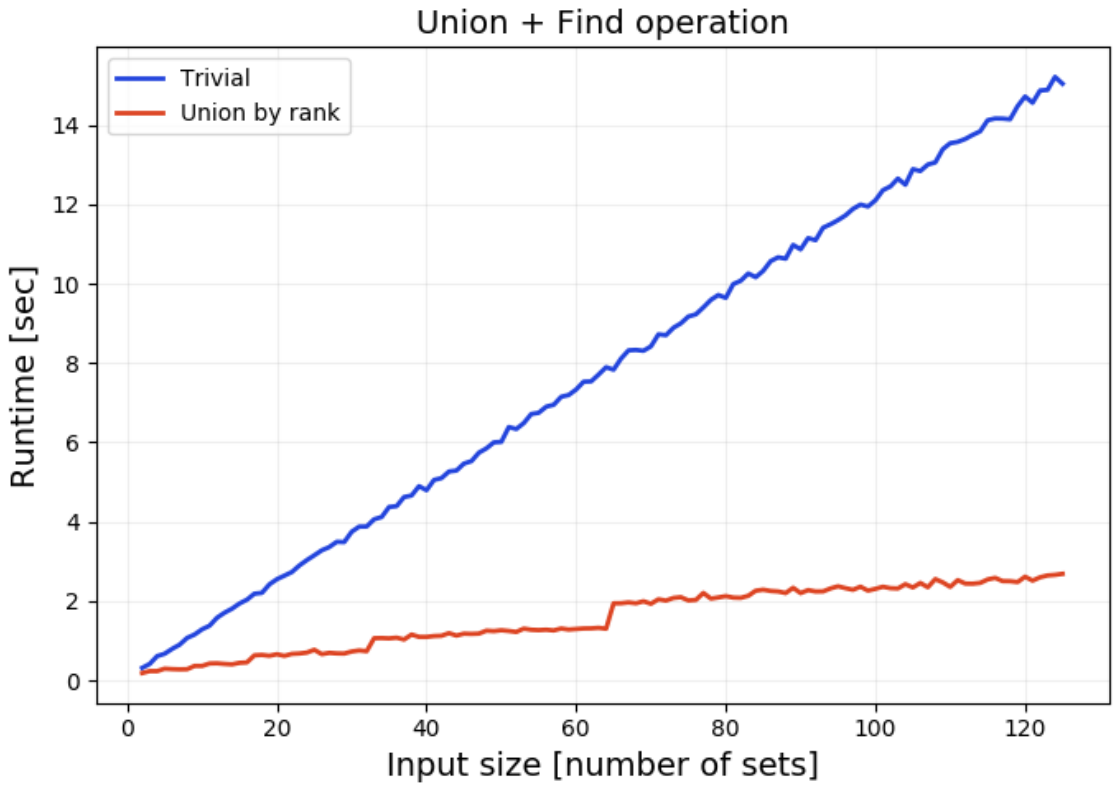

Figure 6.2: Runtime comparison between the naive union-find implementation and the implementation using ORAM with union by rank

The times for our benchmark between different MST implementation is shown in figure 6.3. As expected for smaller input sizes the black-box version of the MST algorithm outperforms the version using predicated instructions. We presume the high number of ORAM read and write accesses causes the constant factor of the predicated instruction version to be much higher than the black-box version. The graph for the Kruskal implementation using predicated instructions has some steep section between 32 to 35 and 66 to 67 which is probably a result of the characteristic of the inverse Ackermann's function. For input sizes with more than 36 edges the runtime of the predicated instruction implementation is always better than the black-box implementation. This is a consequence of the different asymptotic runtime behaviors.

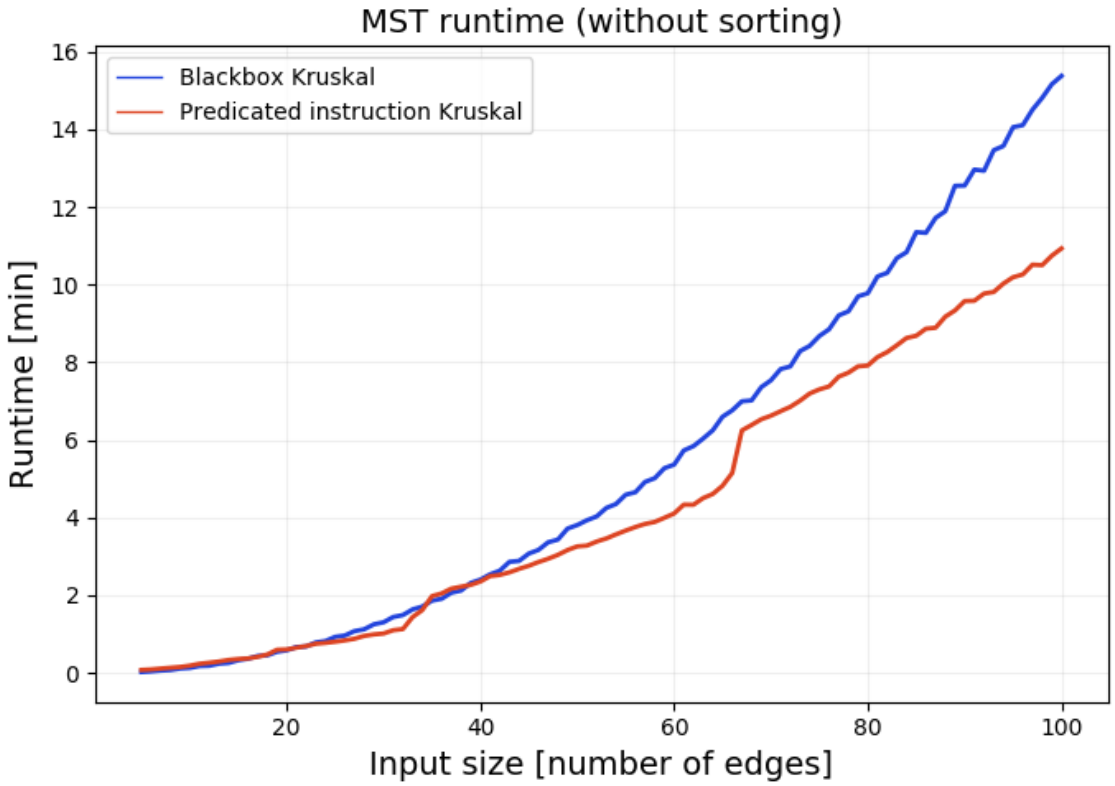

Figure 6.3: Runtime comparison for the online phase of both implementations of Kruskals algorithm

7 Discussion

We introduced a model which we can use as a new approach to create MPC programs from existing non-MPC programs using predicated instructions. We discussed different optimization approaches which can improve the MPC program's runtime: Removing duplicated machine instructions from the program, reordering the predicated instruction and duplicating instructions which are commonly used. Then we applied our results to create a MPC minimum spanning tree program using Kruskal's algorithm.

7.1 Conclusion

Compared to existing solutions (Keller [8], Wang [22]) our approach can result in MPC programs with a better asymptotic runtime. In the experimental part we showed this can succeed in theory as well as in practice. In direct comparison the MPC program implementing Kruskal's algorithm using predicated instructions has better runtime than the implementation using the ideas by Wang et al [11] for input graph with 36 or more edges.

In general we want the original non-MPC program to have certain properties in order for our approach to be successful:

- The program should use at least some branches depending on (hidden) input data, since without any branches there is no point in using predicated instructions.
- The input program should not have too many branches otherwise the overhead introduced by the oblivious execution is too big.
- The total number of jumps into certain branches should be predictable when considering all aggregated calls for each branch.

The last point allows making better assumptions for the optimization of the MPC program and also easily allows us to get an upper bound for the number of predicated instructions required to execute the program. If we don't have such an upper bound we can either estimate an upper by sampling the number for typical input for our program and selecting and upper bound with an acceptable error rate for our application or by periodically revealing if our program has finished executing. Both of these approaches might cause a problem with the security definition of the MPC model which has to be addressed.

7.2 Future Work

Since this is the first work using this kind of approach for MPC programming there are lots of open problem:

The ideas for the optimizations of predicated instruction programs discussed in chapter 5 were only applied to specific algorithms in the practical part of this work. We focused on the ideas behind the optimizations and implemented some improvements specific to the Kruskal's algorithm. However, it would be possible to use the suggested optimization ideas to implement a practical compiler that can turn any RAM program into an MPC protocol. But in particular the removal of duplicated instructions requires programs with more predicated instruction in order to be useful.

We also did not yet consider what is the best way to combine our different optimization approaches. This problem is not trivial since certain optimizations influence the effectiveness of each other.

Another potential optimization could be the use of parallelisation in our predicated instruction approach. We can execute multiple predicated instructions from our program at the same time and use the property that the program counter can only point to one of the predicated instructions.

Finally the idea of using predicated instruction could be combined in conjunction with other approaches to MPC algorithms. As mentioned in chapter 5 the idea behind this work can be applied solely to a specific section of a program. As such the idea behind predicated instructions could be used as a tool when constructing sophisticated algorithms for specific problems.

Bibliography

[1] Kai-Min Chung and Rafael Pass. A simple oram. Cryptology ePrint Archive, Report 2013/243, 2013. https://eprint.iacr.org/2013/243.

[2] Ronald Cramer, Ivan Damgård, Daniel Escudero, Peter Scholl, and Chaoping Xing. Spdz2k: Efficient mpc mod 2^k for dishonest majority. Cryptology ePrint Archive, Report 2018/482, 2018. https://eprint.iacr.org/2018/482.

[3] I. Damgard, V. Pastro, N.P. Smart, and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. Cryptology ePrint Archive, Report 2011/535, 2011. https://eprint.iacr.org/2011/535.

[4] O. Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In *Proceedings of the nineteenth annual ACM conference on Theory of computing - STOC '87*. ACM Press, 1987.

[5] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM (JACM)*, 43(3):431–473, May 1996.

[6] Gregory Harfst and Edward Reingold. A potential-based amortized analysis of the union-find data structure. *SIGACT News*, 31:86–95, 09 2000.

[7] Riko Jacob, Kasper Green Larsen, and Jesper Buus Nielsen. Lower bounds for oblivious data structures. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2439–2447. SIAM, 2019.

[8] Marcel Keller. The oblivious machine - or: How to put the c into mpc. Cryptology ePrint Archive, Report 2015/467, 2015. https://eprint.iacr.org/2015/467.

[9] Marcel Keller. MP-SPDZ: A versatile framework for multi-party computation. Cryptology ePrint Archive, Report 2020/521, 2020. https://eprint.iacr.org/2020/521.

[10] Marcel Keller, Emmanuela Orsini, and Peter Scholl. Mascot: Faster malicious arithmetic secure computation with oblivious transfer. Cryptology ePrint Archive, Report 2016/505, 2016. https://eprint.iacr.org/2016/505.

[11] Marcel Keller and Peter Scholl. Efficient, oblivious data structures for mpc. In Palash Sarkar and Tetsu Iwata, editors, *Advances in Cryptology – ASIACRYPT 2014*, pages 506–525, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[12] Marcel Keller and Avishay Yanai. Efficient maliciously secure multiparty computation for ram. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in*

Cryptology – EUROCRYPT 2018, pages 91–124, Cham, 2018. Springer International Publishing.

[13] Kasper Green Larsen and Jesper Buus Nielsen. Yes, there is an oblivious ram lower bound! In *Annual International Cryptology Conference*, pages 523–542. Springer, 2018.

[14] Peeter Laud. Privacy-preserving minimum spanning trees through oblivious parallel ram for secure multiparty computation. Cryptology ePrint Archive, Report 2014/630, 2014. https://eprint.iacr.org/2014/630.

[15] Steve Lu and Rafail Ostrovsky. How to garble ram programs. Cryptology ePrint Archive, Report 2012/601, 2012. https://eprint.iacr.org/2012/601.

[16] Peter Sanders. Suchalgorithmen auf simd-rechnern – weitere ergebnisse zu polyautomaten. 1994.

[17] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, November 1979.

[18] Elaine Shi, T. H. Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious ram with o((logn)3) worst-case cost. In Dong Hoon Lee and Xiaoyun Wang, editors, *Advances in Cryptology – ASIACRYPT 2011*, pages 197–214, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[19] Emil Stefanov, Marten van Dijk, Elaine Shi, T-H. Hubert Chan, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path oram: An extremely simple oblivious ram protocol. Cryptology ePrint Archive, Report 2013/280, 2013. https://eprint.iacr.org/2013/280.

[20] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, April 1975.

[21] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, 1937.

[22] Xiao Shaun Wang, Kartik Nayak, Chang Liu, T-H. Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. Oblivious data structures. Cryptology ePrint Archive, Report 2014/185, 2014. https://eprint.iacr.org/2014/185.

[23] A. C. Yao. Protocols for secure computations. In *23rd Annual Symposium on Foundations of Computer Science (sfcs 1982)*, pages 160–164, Nov 1982.