Optimizing Secure Computation Programs with Private Conditionals

Peeter Laud[1] and Alisa Pankova[1,2,3]

[1] Cybernetica AS
[2] Software Technologies and Applications Competence Centre (STACC)
[3] University of Tartu
{peeter.laud|alisa.pankova}@cyber.ee

Abstract. Secure multiparty computation platforms are often provided with a programming language that allows to write privacy-preserving applications without thinking of the underlying cryptography. The control flow of these programs is expensive to hide, hence they typically disallow branching on private values. The application programmers have to specify their programs in terms of allowed constructions, either using *ad-hoc* methods to avoid such branchings, or the general methodology of executing all branches and obliviously selecting the effects of one at the end. There may be compiler support for the latter.
The execution of all branches introduces significant computational overhead. If the branches perform similar private operations, then it may make sense to compute repeating patterns only once, even though the necessary bookkeeping also has overheads. In this paper, we propose a program optimization doing exactly that, allowing the overhead of private conditionals to be reduced. The optimization is quite general, and can be applied to various privacy-preserving platforms.

1 Introduction

There exist a number of sufficiently practical methods for privacy-preserving computations [35, 14, 8] and secure multiparty computation (SMC) platforms implementing them [24, 5, 7, 11]. To facilitate the use of such platforms, and to hide the cryptographic details from the application programmer, the platforms allow the compilation of protocols from higher-level descriptions, where the latter are specified in some domain-specific language [4, 24, 26, 31, 25] or in a subset of some general language, e.g. C, possibly with extra privacy annotations [12, 37]. Operations with private values are compiled to protocols transforming the representations of inputs of these operations to the representation of the output. If there is no protocol for some operation, then it either has to be forbidden or transformed out.

In SMC protocol sets based on secret sharing [14, 8, 6, 10], the involved parties are usually partitioned into input, computation, and output parties [28], with the computation parties holding the private values in secret-shared form between them, and performing the bulk of computation and communication. In this case, `if`- and `switch`-statements with private conditions are among unsupported operations, because the taken branch should not be revealed to anyone, but it is difficult to hide the control flow of the program. Instead, a program transformation can be applied, where all branches are executed and the final values of all program variables are chosen obliviously from the outcomes of all branches [37, 27]. This introduces a significant overhead. An obvious optimization idea, but which has not received much attention so far except for [19] in a different setting, is to locate similar operations in different branches and try to fuse them into one. The operation is not trivial, because the gathering of inputs to fused operations introduces additional oblivious choices. This paper is devoted to the study of fusing the operations in different branches and evaluating them on top of the Sharemind SMC platform [6].

In this work, we consider a simple imperative language with variables typed “public” and “private”, invoking SMC protocols to process private data. It allows to use private types in the conditions of `if` and `switch` statements. We translate a program written in this language into a computational circuit. We optimize the circuit, trying to fuse together the sets of operations, where the outcome of at most one of them is used in any concrete execution.

2 Preliminaries

Secure multiparty computation In secure multiparty computation (SMC), several parties are communicating over a network. They want to compute some function on secret inputs, where each party is allowed to see only its own inputs and outputs, but not the inputs of the other parties. In secret-sharing based SMC, the actual values of the inputs are provided by input parties and are not known to any of the computing parties, and the values are shared amongst the computing parties according to some secret sharing scheme. The final output of the function that the parties compute can be either revealed to certain output parties, or it may stay shared, being used as an input in subsequent computations.

Languages for secure multiparty computation A programmer who writes a particular application for secure multiparty computation would not like to write out the protocol for each party separately. It is easier to write a program that describes the computed functionality on a higher level, without taking into account how exactly the inputs are shared, and hiding all the underlying cryptography. Existing privacy-preserving application platforms are usually provided with such a language [37, 6, 24, 9]. A program looks very similar to an ordinary imperative language (such as Java, Python, or C), but it does much more, as it is being compiled to a sequence of cryptographic protocols.

Oblivious choice Suppose that there are n secret values $x_1,\dots,x_n$ and a secret index $i \in \{1,\dots,n\}$. The goal is to compute x_i. In the case of branchings with private conditions, we are given the bits $b_1,\dots,b_n$, at most one of which is 1, where $b_i = 1$ means that the i-th branch should be taken. In particular, if x_i is the value that the variable x gains after executing the i-th branch, then the value of x can be computed as $x = b_1x_1 + \dots + b_nx_n$ (after executing all the n branches).

Computation Circuits A computation circuit is a directed acyclic graph where each node is assigned a value that can be computed from its immediate predecessors, except the input nodes which have no predecessors and obtain their values externally. We use circuits to represent the computation that takes place inside the branches of private conditionals.

Mixed integer linear programming [30] A *linear programming* task is an optimization task stated as

$$\text{minimize } \mathbf{c}^{\mathrm{T}} \cdot \mathbf{x},\ \text{subject to } \mathrm{A}\mathbf{x} \leq \mathbf{b}, \mathbf{x} \geq \mathbf{0} \quad , \tag{1}$$

where $\mathbf{x} \in \mathbb{R}^n$ is a vector of variables that are optimized, and the quantities $A \in \mathbb{R}^{m\times n}$, $\mathbf{b} \in \mathbb{R}^n$, $\mathbf{c} \in \mathbb{R}^n$ are the parameters that define the task itself.

Adding constraints $x_i \in \mathbb{N}$ for $i \in \mathcal{I}$ for some $\mathcal{I} \subseteq \{1,\dots,n\}$ gives us a *mixed integer linear programming* task. Adding constraints $x_i \in \{0,1\}$ for $i \in \mathcal{I}$ gives us a *mixed binary integer linear programming* task:

$$\text{minimize } \mathbf{c}^{\mathrm{T}} \cdot \mathbf{x},\ \text{s.t } \mathrm{A}\mathbf{x} \leq \mathbf{b}, \mathbf{x} \geq \mathbf{0}, x_i \in \{0,1\} \text{ for } i \in \mathcal{I} \quad . \tag{2}$$

We will reduce the optimization of a computational circuit to a mixed binary integer linear programming task.

Notation We denote the subset of integers $\{1,\dots,n\}$ by $[n]$. We write $e[x \leftarrow v]$ to express the substitution of the variable x in the expression e with some variable or value v. We denote vectors by $\mathbf{x} = [x_1,\dots,x_n]$.

3 Related Work

There are a number of languages for specifying privacy-preserving applications to be run on top of SMC platforms, these may be either domain-specific languages [4, 24, 31] or variants of general-purpose languages [12]. Often these languages do not offer support for private conditionals.

The support of private conditionals is present in SMCL [26], as well as in newer languages and frameworks, such as PICCO [37], Obliv-C [36], Wysteria [29], SCVM [21], or the DSL embedded in Haskell by Mitchell et al. [25]. A necessary precondition of making private conditions possible is forbidding any public side effects inside the private branches (such as assignments to public variables or termination), since that may leak information about which branch has been executed. All the branches are executed simultaneously, and the value of each variable that could have been modified in at least one of these branches is updated by selecting its value obliviously. Planul and Mitchell [27] have more thoroughly investigated the leakage through conditionals. They have formally defined the transformation for executing all branches and investigated the limits of its applicability to programs that have potentially non-terminating sub-programs.

$$
\begin{aligned}
\mathit{prog} &::= \mathit{stmt}\\
f &::= \mathit{arithmetic\ blackbox\ function}\\
\mathit{exp} &::= x^{\mathrm{pub}} \mid x^{\mathrm{priv}} \mid c \mid f\texttt{(}\ \mathit{exp}^{*}\ \texttt{)} \mid \texttt{declassify(}\ \mathit{exp}\ \texttt{)}\\
\mathit{stmt} &::= x\ \texttt{:=}\ \mathit{exp}\\
&\quad \mid\ \texttt{skip}\\
&\quad \mid\ \mathit{stmt}\ \texttt{;}\ \mathit{stmt}\\
&\quad \mid\ \texttt{if}\ \mathit{exp}\ \texttt{then}\ \mathit{stmt}\ \texttt{else}\ \mathit{stmt}
\end{aligned}
$$

Fig. 1: Syntax of the imperative language

The existing compilers that support private conditionals by executing both branches do not attempt to reduce the computational overhead of such execution. We are aware of only a single optimization attempt targeted towards this sort of inefficiencies [19], but the details of their setting are quite different from ours. They are targeting privacy-preserving applications running on top of garbled circuits (GC), building a circuit into which all circuits representing the branches can be embedded. Their technique significantly depends on what can be hidden by the GC protocols about the details of the circuits. Our approach is more generic and applies at the language level.

More generally, the hiding of conditionals can be seen as an instance of oblivious computation. There exist methods for executing Random Access Machines (RAM) in a fully oblivious manner, without taking all branches [16, 22, 32], these techniques make heavy use of Oblivious RAM [15, 33]. Currently, these techniques are not competitive performance-wise with non-oblivious methods of execution, even if the latter support private conditionals only through the execution of all branches. However, oblivious computation can be highly efficient if *ad-hoc* methods for decoupling the control flow from private data are available, which is the case for e.g. sorting [17] or for certain graph algorithms [2]. Still, these techniques are not closely related to the optimizations we propose in this paper.

4 Our Contribution

In this work, we reduce the computational overhead of private conditionals by *fusing* mutually exclusive operations of different branches into a single operation, introducing additional oblivious choice gates that allow to select the appropriate inputs for it. Our optimization is based on mixed integer linear programming, but some greedy heuristics are proposed as well for better performance. The main difference from related work is that our optimization is very generic and can be applied on the program level, without the need of decomposing high-level operations to arithmetic or boolean circuits. We do the optimization for some simple programs with private conditionals, and benchmark their running time on a particular SMC platform, showing that the optimization is indeed useful in practice.

We describe the settings in Sec. 5, describe the optimization in Sec. 6, and report the benchmark results in Sec. 7.

5 Programs and Circuits

5.1 The imperative language with Private Conditions

We start from a simple imperative language, given in Fig. 1, which is just a list of assignments and conditional statements. The variables x in the language are typed either as public or private, these types also flow to expressions. Namely, the expression $f(e_1,\ldots,e_n)$ is private iff at least one of e_i is private. The declassification operation turns a private expression to a public one. An assignment of a private expression to a public variable is not allowed. Only private variables may be assigned inside the branches of private conditions [37, 27]. The syntax c denotes compile-time constants.

During the execution of a program on top of a secret-sharing based SMC platform, public values are known by all computation parties, while private values are secret-shared among them [4]. An *arithmetic blackbox function* is an arithmetic, or relational, or boolean, etc. operation, for which we have implementations for all partitionings of its arguments into public and private values. E.g. for integer multiplication, we have the multiplication of public values, as well as protocols to multiply two private values, as well as a public and a private value [6].

The programs in the language of Fig. 1 cannot all be executed due to the existence of private conditionals. They can be executed after translating them into computational circuits. These circuits are not convenient for expressing looping constructs. Also, our optimizations so far do not handle loops. For this reason, we have left them out of the language. We note that loops with public conditions could in principle be handled inside private conditionals [37].

The semantics of the initial imperative language are formally defined in App. A.

5.2 Computational Circuits

Given a set *Var* a program variables, we define a circuit that modifies the values of (some of) these variables. It consists of the set of gates G doing the computation, the mapping X that maps the input wires of G to the program variables *Var*, so that we can feed their valuations to the circuit, and the mapping Y that maps the variables of *Var* to the output wires of G, so that we may assign to the program variables the new valuations obtained from the circuit execution.

Definition 1. *Let V be the global set set of wire names. Let Val be the set of program variables. A computational circuit is a triple $\mathsf{G} = (G, X, Y)$ where:*

1. *$G = \{g_1, \ldots, g_m\}$ for some $m \in \mathbb{N}$, where each $g \in G$ is of the form $g = (v, op, [v_1, \ldots, v_n])$ where:*
   - *$v \in V$ is a unique gate identifier;*
   - *op is the operation that the gate computes (an arithmetic blackbox function of the SMC platform);*
   - *$[v_1, \ldots, v_n]$ for $v_i \in V$ is the list of the arguments to which the operation op is applied when the gate is evaluated.*
2. *X is a mapping whose domain defines the* input wires *$I(\mathsf{G}) \subseteq V$ of the circuit G. Formally $X : I(\mathsf{G}) \to Var$ assigns to each wire $v \in I(\mathsf{G})$ the variable $X(v)$.*
3. *Y is a mapping whose range defines the set of* output wires *$O(\mathsf{G}) \subseteq V$ whose values are finally output. Formally, $Y : Var \to O(\mathsf{G})$ assigns to $y \in Var$ a wire $Y(y)$.*

We define the set of all wires of G as the set of all gate identifiers and their arguments (which do not necessarily have to be in turn some gate identifiers):

$$V(\mathsf{G}) := \{v \mid \exists op, \mathbf{v} :\ (v, op, \mathbf{v}) \in G\} \cup \{v \mid \exists u, op, \mathbf{v} :\ (u, op, \mathbf{v}) \in G, v \in \mathbf{v}\} \ .$$

In order to easily switch between the sets of G and $V(\mathsf{G})$, we define a function $\mathsf{gate} : V(\mathsf{G}) \to G$ s.t $\mathsf{gate}(v) = (v, op, \mathbf{v})$ if $\exists op, \mathbf{v} :\ (v, op, \mathbf{v}) \in G$, and $\mathsf{gate}(v) = \bot$ otherwise. Since the gate names are unique, the function inverse gate^{-1} is well-defined.

We use $\mathcal{G}$ to denote the set of all circuits.

The circuits that we work on are going to contain gates whose operation is the *oblivious choice*; such gates are introduced while transforming out private conditionals. Such gate is defined as $(v, oc, [b_1, v_1, \ldots, b_n, v_n])$, and it returns the output of $\mathsf{gate}(v_i)$ iff the output of $\mathsf{gate}(b_i)$ is 1. If there is no such b_i, then it outputs 0. It works on the assumption that at most one $\mathsf{gate}(b_i)$ outputs 1. This assumption needs to be ensured by the transformation that constructs a circuit from a program.

The formal definition of evaluating the set of gates G on the input X, and the definition of the oblivious choice gate in particular, are given in App. B.1. The composition of circuits as syntactic objects and the semantics preservation proof of this operation are given in App. B.2.

Example 1. A circuit that chooses z obliviously from $x_1 * y_1$, $x_2 * y_2$, $x_3 * y_3$ according to the choice bits b_1, b_2, b_3 would be defined as:

- $G = \{(u_1, *, [v_1, w_1]), (u_2, *, [v_2, w_2]), (u_3, *, [v_3, w_3]), (v, oc, [v_1^b, u_1, v_2^b, u_2, v_3^b, u_3])\}$;
- $X = \{v_1 \leftarrow x_1, v_2 \leftarrow x_2, v_3 \leftarrow x_3, w_1 \leftarrow y_1, w_2 \leftarrow y_2, w_3 \leftarrow y_3, v_1^b \leftarrow b_1, v_2^b \leftarrow b_2, v_3^b \leftarrow b_3\}$;
- $Y = \{z \leftarrow v\}$.

This circuit is depicted in Fig.2.

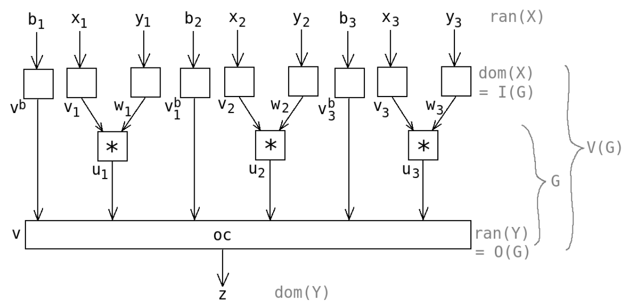


Fig. 2: Example 1

5.3 Transforming a Program to a Circuit

We need to transform the private conditional statements of the initial imperative language to a circuit. Intuitively, each assignment $y := f(x_1,\ldots,x_n)$ of the initial program can be viewed as as single circuit computing a set of gates G defined by the description of f on inputs $x_1,\ldots,x_n$, where X maps the input wires of the circuit to the variables $x_1,\ldots,x_n$, and Y maps y to the output wire of the circuit. A sequence of assignments is put together into a single circuit using circuit composition.

If the program statement is not an assignment, but a private conditional statement, all its branches are first transformed to independent circuits (G_i, X_i, Y_i). The value of each variable y is then selected obliviously amongst $Y_i(y)$ as $y := \sum_i b_i Y_i(y)$, where b_i is the condition of executing the i-th branch. So far, the transformation is similar to the related work [37, 27], and the only formal difference is that we construct a computational circuit at this point.

The formal definition of transforming a program to a circuit is given in App. C.

Example 2. Suppose we are given the following conditional statement with a private condition b:

```
if b:
    x := x + y;
else:
    y := 5*x;
    x := 2;
```

This would be transformed to the following circuit:

- $G = \{(u_{add}, +, [v_x, v_y]), (\bar{v}_b, \neg, [v_b]), (u_{mul}, *, [v_{const5}, v_y]),$
  $(w_x, oc, [v_b, u_{add}, \bar{v}_b, v_{const2}]), (w_y, oc, [v_b, v_y, \bar{v}_b, u_{mul}])\}$;
- $X = \{v_x \leftarrow x, v_y \leftarrow y, v_b \leftarrow b, v_{const5} \leftarrow 5, v_{const2} \leftarrow 2\}$;
- $Y = \{x \leftarrow w_x, y \leftarrow w_y\}$.

This circuit is depicted by Fig.3.

6 Optimizing the Circuit

The circuit G obtained from the transformation of Sec. 5.3 may be non-optimal. Namely, it contains executions of all the branches of private conditional statements, although only one of the branches will be eventually needed. In this section, we present an optimization that eliminates excessive computations caused by the unused branches.

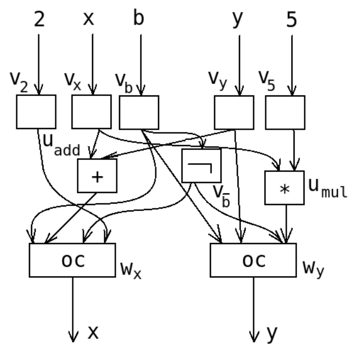


Fig. 3: Example 2

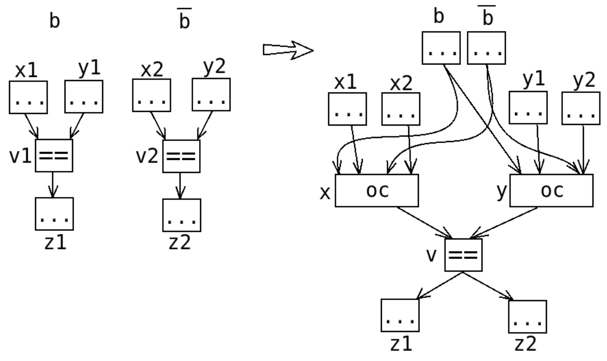


Fig. 4: Example 3

6.1 Informal Description of the Optimization

Let $\mathsf{G} = (G, X, Y)$ be a computational circuit. The *weakest precondition* ϕ_v^{G} of evaluating a gate $g = \mathtt{gate}(v) \in G$ is a boolean expression over the conditional variables, such that $\phi_v^{\mathsf{G}} = 1$ if the result of evaluating gate g is needed for the given valuations of conditional variables. A more formal definition of the weakest precondition is given in App. D.

The main idea of our optimization is the following. Let $g_1 = (v_1, op, [x_1^1, \ldots, x_n^1]), \ldots, g_k = (v_k, op, [x_1^k, \ldots, x_n^k]) \in G$. Let $\phi_{v_1}^{\mathsf{G}}, \ldots, \phi_{v_k}^{\mathsf{G}}$ be mutually exclusive. This happens for example if each g_i belongs to a distinct branch of a set of nested conditional statements. In this case, we can *fuse* the gates $g_1, \ldots, g_k$ into a single gate g that computes the same operation op, choosing each of its inputs x_j obliviously amongst $x_j^1, \ldots, x_j^k$. This introduces n new oblivious choice gates, but leaves just one gate g computing op.

Example 3. Let a comparison operation $(==)$ be located in both the `if`-branch, and the corresponding `else`-branch. Let the gates be $(v_1, ==, [x_1, y_1]), (v_2, ==, [x_2, y_2]) \in G$. Let $b \in V(\mathsf{G})$ be the wire whose value is the condition of the `if`-branch, and $\bar{b} \in V(\mathsf{G})$ the wire whose value is b's negation (which is the condition of the `else`-branch). Since the branches can never be executed simultaneously, we may replace these gates with $(x, oc, [b, x_1, \bar{b}, x_2])$, $(y, oc, [b, y_1, \bar{b}, y_2])$, and $(v, ==, [x, y])$. All the references to v_1 and v_2 in the rest of the circuit are now substituted with the reference to v. The transformation is shown in Fig. 4.

We now describe this optimization in more details.

Preprocessing Let $n = |V(\mathsf{G})|$, and $m = |G|$. First, we find the set U of all pairs of gates that can never be evaluated simultaneously. For each gate g_i, find the weakest precondition ϕ_i^{G} that be must true for g_i to be evaluated. Define $U = \{(g_i, g_j) \mid g_i, g_j \in G, \phi_i^{\mathsf{G}} \wedge \phi_j^{\mathsf{G}}$ is unsatisfiable$\}$. Although there is no efficient algorithm for solving the unsatisfiability

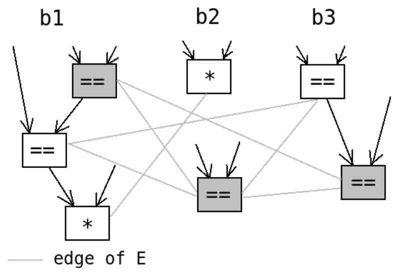


Fig. 5: Fusing gates into cliques

problem, in practice, it suffices to find only a subset $U' \subseteq U$. It makes the optimization less efficient (we do not fuse as many gates as we could), but nevertheless correct.

For each formula ϕ_i^{G}, we need to construct the circuit that computes the value of ϕ_i^{G}. Depending on how exactly ϕ_i^{G} is computed, evaluating this circuit may in turn have some cost. Each ϕ_i^{G} is represented by a boolean formula over the conditions of the `if`-statements of the initial program, which can be read out from G by observing its oblivious choice gates. The additional $\vee$ and $\wedge$ gates are need in the cases where a gate is located inside several nested `if`-statements (need $\wedge$), or it is used in several different branches (need $\vee$).

Plan We partition the gates into sets C_k, planning to leave only one gate of C_k after the optimization. The following conditions should hold:

- $\forall g_i, g_j \in C_k : \ g_i \neq g_j \implies (g_i, g_j) \in U$: we put together only mutually exclusive gates, so that indeed at most one gate of C_k will actually be executed.
- $\forall g_i, g_j \in C_k : \ op_i = op_j$: only the gates that compute the same operation are put together.
- Let $E := \{(C_i, C_j) \mid \exists k, \ell : g_k \in C_i, g_\ell \in C_j, g_k$ is an immediate predecessor of g_ℓ in $G\}$. In this way, if $(C_i, C_j) \in E$, then C_i should be evaluated strictly before C_j. We require that the graph $(\{C_k\}_{k \in [m]}, E)$ is acyclic. Otherwise, we might get the situation where some gates of C_j have to be computed necessarily before C_i, and at the same time some gates of C_i should be computed necessarily before C_j, so evaluating all the gates of C_i at once would be impossible.

If we consider U as edges, we get that C_k form a set of disjoint cliques on the graph. A possible fusing of gates into a clique is shown in Fig. 5, where the gray lines connect the pairs $(g_i, g_j) \in U$, and the dark gates are treated as a single clique.

Transformation The plan gives us a collection of sets of gates C_j, each having gates of certain operation op_j. Consider any $C_j = \{g_1, \ldots, g_{m_j}\}$. Let the inputs of the gate g_i be $x_1^i, \ldots, x_n^i$. Let b_i be the wire that outputs the value of ϕ_i^{G}. Introduce n new oblivious choice gates $(v_\ell, oc, [b_1, x_\ell^1, \ldots, b_{m_j}, x_\ell^{m_j}])$ for $\ell \in [n]$. Add a new gate $(g, op_j, [v_1, \ldots, v_n])$. Discard all the gates g_i. If any gate in the rest of the circuit has used any g_i as an input, substitute it with g instead. We may additionally omit any oblivious choice in the graph if there is just one option to select from.

For each oc gate that has been already present in the circuit, check how many distinct inputs it has. It is possible that some inputs have been fused into one due to belonging to the same clique. In this case, it may happen that the oc gate is left with a single choice, and since $(v_\ell, oc, [b_i, x_\ell^i])$ just returns x_ℓ^i, its cost is 0. An example of leaving just one clique representative that allows to eliminate the further oblivious choice can be seen in Fig. 6.

The Cost Our goal is to partition the cliques in such a way that the *cost* of the resulting circuit is minimal. Each gate operation corresponds to some SMC protocol that requires some amount of bits to be communicated between the parties. We choose the total number of communicated bits as the cost. Since this metric is additive, we may easily estimate the total cost of the circuit by summing up the communicated bits of the gates. The particular costs of the gates depend on the chosen SMC platform.

We note that introducing intermediate oblivious choices may increase the number of of rounds. We need to be careful, since increasing the number of rounds may make executing the circuit with a lower communication cost actually take more time.

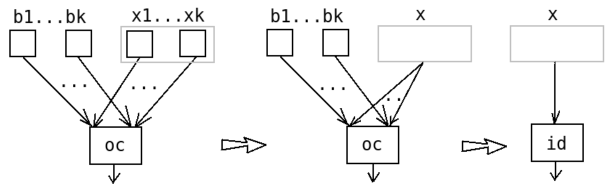


Fig. 6: Leaving just one clique representative

6.2 Notation for describing the optimization

We do some formalization on the circuit that allows to make further optimization easier.

Shorthand Notation Let $\mathsf{G} = (G, X, Y)$, $(g, op, [v_1, \ldots, v_n]) \in G$. We introduce the following shorthand notation:

- $\mathsf{op}^{\mathsf{G}}(g) = op$;
- $\mathsf{args}^{\mathsf{G}}(g) = [v_1, \ldots, v_n]$;
- $\mathsf{arity}^{\mathsf{G}}(g) = n$;

For $g \in V(\mathsf{G})$, we write $g \in G$ for $\mathsf{gate}(g) \in G$.

Gate and Clique Enumeration Without loss of generality, let $V(\mathsf{G}) = \{1, \ldots, n\}$. This ordering allows to define the constraints for linear program variables more easily.

The number of cliques varies between 1 (if all the gates are fused together into one) and $|G|$ (if each gate is a singleton clique). For simplicity, we assume that we always have exactly $|G|$ cliques, and some of them may just be left empty. We denote the clique $\{i_1, \ldots, i_k\}$ by C_j, where $j = \min(i_1, \ldots, i_k)$ is the *representative* of the clique C_j. Without loss of generality, let the representative be the only gate that is left of C_j after the fusing.

The Weakest Precondition We write $\phi^{\mathsf{G}}(i)$ to refer to the weakest precondition of the gate $i \in V(\mathsf{G})$, as it is defined by the structure of G. The particular algorithm for computing the mapping ϕ^{G} for the circuit G is given in App. D.2.

Direct and Conditional Predecessors In order to ensure that the optimized circuit contains no cycles, we need to remember which gates have been predecessors of each other. We define the following auxiliary predicates that can be easily computed from the initial circuit.

- $\mathsf{pred}^{\mathsf{G}}(i,k) = 1$ iff $k \in \mathsf{args}^{\mathsf{G}}(i)$;
- $\mathsf{cpred}^{\mathsf{G}}(i,k) = 1$ iff $k \in \phi^{\mathsf{G}}(i)$;

The predicate $\mathsf{pred}^{\mathsf{G}}(i,k)$ is true just if k is an immediate predecessor of i in G. The predicate $\mathsf{cpred}^{\mathsf{G}}(i,k)$ is true if k is used to compute the weakest precondition of i. This means that k does not have to be computed strictly before i in general. However, if i is fused with some other gate, we will need the value of k for computing the oblivious choice of the arguments of i, and in this case k has to be computed strictly before i. In this way, k is a predecessor of i on the condition that i is fused with at least one other gate.

Which gates can be fused We define an auxiliary predicate that denotes which gates are allowed to be fused:

$$\mathsf{fusable}^{\mathsf{G}}(i,j) = 1 \text{ iff } (i = j) \vee (\phi^{\mathsf{G}}(i) \wedge \phi^{\mathsf{G}}(j) \text{ is unsatisfiable}) \ .$$

Although there exists no efficient algorithm for computing unsatisfiablity in general, we may allow some algorithm that provides false negatives. This results in having $\mathsf{fusable}^{\mathsf{G}}(i,j) = 0$ for gates that could have actually been fused, and hence the final solution may be non-optimal, but nevertheless correct.

Since fusing forces all the gate arguments to become chosen obliviously, all the inputs of a fused gate in general become private (unless there was just one choice for some public input). Depending on the SMC platform and the particular operation, this may formally change the gate operation. Some operations still retain the same cost, while some gates may increase their cost significantly if some of their public inputs become private. Moreover, it may happen that the new operation is not supported by the SMC platform at all. We define $\mathsf{fusable}^{\mathsf{G}}(i,j) = 0$ for the gates that have any public inputs, and whose cost depends on their privacy.

6.3 Uniting Gates Into Subcircuits

Let $\mathsf{G} = (G, X, Y) \in \mathcal{G}$. In some cases, there are obvious repeating patterns of gates in G which could be treated as a single gate. Uniting them into one gate would reduce the total number of gates involved in the optimization, increasing its efficiency.

We propose a particular algorithm for partitioning G into a set of disjoint subcircuits. We define the sets of subcircuits A_k inductively as follows.

$$\begin{aligned}
A_0 &:= \{\{g\} \mid g \in G\} \ ; \\
A'_{n+1} &:= \{S \cup \bigcup \{S_i \in \mathsf{args}^{A_n}(S) \mid \ \nexists T \in A'_{n+1} : S_i \subseteq T, \nexists S' \neq S \in A_n : \ S_i \in \mathsf{args}^{A_n}(S')\} \mid S \in A_n\} \ ; \\
A_{n+1} &= \{S \mid \mathsf{count}(S, A'_{n+1}) \geq 2\} \\
&\quad \cup \ \{S' \mid S \in A'_{n+1}, \mathsf{count}(S, A'_{n+1}) = 1, S' \sqsubseteq S\} \ ;
\end{aligned}$$

Where $S' \sqsubseteq S$ denotes that $S' \in A_n$ is a subcircuit that has been united into $S \in A_{n+1}$, and $\mathsf{count}(S, A'_{n+1})$ is the number of elements in A'_{n+1} that are isomorphic to S. The isomorphisms, which require the sameness of structure and operations, are simple to find out due to the inputs of all gates being ordered.

The arguments of the subcircuits are also defined inductively as

$$\begin{aligned}
\mathsf{args}^{A_0}(\{g\}) &:= \{\{a\} \mid a \in \mathsf{args}^{G}(g)\} \ ; \\
\mathsf{args}^{A_{n+1}}(S) &:= \{S_i \mid S' \sqsubseteq S, S_i \in \mathsf{args}^{A_n}(S'), S_i \not\sqsubseteq S\} \ .
\end{aligned}$$

We start from the initial set of gates G, treating each gate as a singleton subcircuit. On each iteration, we extend each subcircuit with its argument gates that it does not share with any other subcircuits. We want the subcircuits to be disjoint, and hence if some $S_i \in A_n$ has already been extended on this iteration, then we are not trying to use S_i to extend some other $S \in A_n$ – that is how the condition $S_i \subseteq T \notin A'_{n+1}$ should be interpreted.

If any subcircuit S occurs only once in A_{n+1}, then it does not make sense to include S into the optimization, since it cannot be merged with any other gate anyway. In this case, after each iteration we may leave only those subcircuits of A_{n+1} that occur at least twice. Each $S \in A'_{n+1}$ that occurs only once is decomposed back to its subcircuits $S' \sqsubseteq S$, $S' \in A_n$.

Let $\mathsf{Subcircuit}(G, n)$ be a function computing the set of subcircuits A_n for the given G and n. After composing the subcircuits in this way, we are only allowed to use the final outputs of the subcircuits. The outputs of the gates that are swallowed by a subcircuit can only be used inside that subcircuit. Hence we need to prove that the set of new gates A_n still does exactly the same computation as G. This is stated and proved in Thm. 3, which can be found in App. E. In this way, the optimizations proposed in further subsections can be applied to single gates as well as to the partitions formed by the function $\mathsf{Subcircuit}$.

6.4 Simple Greedy Optimizations

We have investigated simple, heuristic optimizations, described in this subsection. We have also considered more complex optimizations based on integer linear programming; these are described in the next subsection. The outline of a simple optimization is the following.

Grouping Gates by Operations First of all, the gates G are grouped as subsets $Gss = \{\{g \in G \mid \mathsf{op}^{\mathsf{G}}(g) = F\} \mid F$ is a gate operation$\}$ by their operation. These subsets are sorted according to the cost of their operation, so that more expensive gates come first.

Forming Cliques The subsets are turned into cliques one by one, starting from the most expensive operation. A clique C_k is formed only if it is valid and is not in contradiction with already formed cliques, i.e:

- any two gates $g_i, g_j \in C_k$ satisfy $\mathsf{fusable}^{\mathsf{G}}(g_i, g_j) = 1$;
- no $g_i \in C_k$ has already been included into some other clique;
- C_k does not introduce cycles.

We use three different strategies for forming a set of cliques for a particular subset of gates $Gs \in Gss$:

1. **Largest Cliques First**. In this approach, we are trying to extract from Gs as large clique as possible before proceeding with the other cliques. The task of finding one maximum clique is NP-hard, and so we approximate the task by generating some bounded amount of cliques and taking the largest of them.
2. **Pairwise Merging**. The gates are first merged pairwise. We are not trying to maximize the number of pairs, and just take the first valid pair that we find. After no more pairs can be formed, we proceed merging the obtained pairs in turn pairwise. We continue until the number of cliques formed by Gs cannot be decreased anymore.
3. **Pairwise Merging with Maximum Matching**. This approach is similar to the previous one, and the only difference is that, instead of fixing the first valid pair, it finds the *maximum* matching on each step.

The more formal descriptions of the algorithms are given in App. F. We also prove in App. F that these strategies do not lead to a dead end, i.e after a clique has been fixed, it is always possible to assign the remaining gates to cliques without backtracking.

6.5 Reduction to an integer programming task

As an alternative to greedy algorithms, we may reduce the gate fusing task to an integer program and solve it using an external integer linear program solver (such as [13]).

We consider mixed integer programs of the form (2). Let $\mathcal{ILP}$ be the set of all mixed binary integer programs defined as tuples $(A, \mathbf{b}, \mathbf{c}, \mathcal{I})$. For our particular task, we define a transformation $T_{ILP}^{\rightarrow} : \mathcal{G} \rightarrow \mathcal{ILP}$, such that $T_{ILP}^{\rightarrow}(G, X, Y) = (A, \mathbf{b}, \mathbf{c}, \mathcal{I})$. We now describe in details how these quantities are constructed.

In order to make integer programming solutions better comparable to greedy algorithms, we consider two levels of optimization:

- **Basic:** try to optimize only the total cost of the gates, without taking into account the *oc* gates.
- **Extended:** take into account the new *oc* gates, the weakest preconditions, and also the number of inputs of the old *oc* gates.

Throughout this section, we use G to refer to the initial circuit, and G' to refer to the circuit obtained after the transformation.

Variables For a clique C_j, let us denote set of all possible choices for the ℓ-th input of the clique C_j as $\mathsf{args}^G(C_j)[\ell] := \{k \mid i \in C_j, k = \mathsf{args}^G(i)[\ell]\}$. We will now describe the meaning of different variables in our ILP task.

The core of our optimization are the variables that affect the cost of the transformed circuit.

- $g_i^j = \begin{cases} 1, \text{if } i \in C_j \\ 0, \text{otherwise} \end{cases}$ for $i, j \in G$.
  The gate j will be the representative of C_j. Namely, $g_j^j = 1$ iff C_j is non-empty. Fixing the representative reduces the number of symmetric solutions significantly. This also allows us to compute the cost of all the cliques.
- $uc^j = |\mathsf{args}^{G'}(j)| - 1$ for $j \in G$, $\mathsf{op}^G(j) = oc$ is number of decisions left for the *oc* gate j, after some of its choices have potentially been fused together.
- $u^j = \begin{cases} 1, \text{if } |\mathsf{args}^{G'}(j)| > 1 \\ 0, \text{otherwise} \end{cases}$ for $j \in G$, $\mathsf{op}^G(j) = oc$.
  If $u^j = 0$, then the *oc* gate j can be removed since there is only one choice left. The variables uc^j and u^j allow to count for the updated cost of the old *oc* gates.
- $sc_\ell^j = |\mathsf{args}^{G'}(j)[\ell]| - 1$ for $j \in G$, $\ell \in \mathsf{arity}^G(j)$ is the number of decisions to make for choosing the ℓ-th argument of C_j.
- $s_\ell^j = \begin{cases} 1, \text{if the } \ell\text{-th input of } j \text{ should be a new } oc \text{ gate} \\ 0, \text{otherwise} \end{cases}$
  for $j \in G$, $\ell \in \mathsf{arity}^G(j)$.
  The variables sc_ℓ^j and s_ℓ^j allow to count for the total cost of the new *oc* gates introduced by the optimization.

– $b_i = \begin{cases} 1, \text{if the weakest precondition of } i \text{ is needed} \\ 0, \text{otherwise} \end{cases}$
for $i \in G$.
Fusing the gates requires their inputs to be chosen obliviously. For that, we may need to compute the weakest preconditions of the participating gates.

We also need some variables that help to avoid cycles after fusing the gates.

– $\ell_j \in \mathbb{R}$ for $j \in G$ is the circuit topological level on which the j-th gate is evaluated, where all the gates with the same level are evaluated simultaneously. Each gate must have a strictly larger level than all its predecessors.
– $c_j = \begin{cases} 1, \text{if the gate } g_j \text{ is fused with some other gate} \\ 0, \text{otherwise} \end{cases}$ for $j \in G$.
Each gate should have a strictly larger level than all its *conditional* predecessors iff it participates in a clique of size at least 2. After the gates are fused into a clique, their inputs are going to be chosen obliviously, and hence the condition will have to be known strictly *before* the fused gates are evaluated.

The vector **x** of variables will actually contain some more auxiliary variables that help to establish relations between the main variables, but do not have special meaning otherwise. We will see these variables when we define constraints.

Cost function The total cost of the circuit is defined by the following quantities.

– $\mathsf{C}_g = \sum_{j=1,\mathsf{op}^G(j)\neq oc}^{|G|} cost(\mathsf{op}^G(j)) \cdot g_j^j$ is the total cost of the cliques after fusing (except the *oc* gates).
– $\mathsf{C}_{oc1} = \sum_{j=1,\mathsf{op}^G(j)=oc}^{|G|} cost(oc_{base}) \cdot u^j + cost(oc_{step}) \cdot uc^j$ is the total cost of all the old *oc* gates, where $cost(oc_{base})$ is the base cost of using an *oc* gate, and $cost(oc_{step})$ is the cost of a single choice of the *oc* gate.
– $\mathsf{C}_{oc2} = \sum_{j,\ell=1,1}^{|G|,\mathsf{arity}^G(j)} cost(oc_{base}) \cdot s_\ell^j + cost(oc_{step}) \cdot sc_\ell^j$ is the total cost of all the new *oc* gates.
– $\mathsf{C}_b = \sum_{j=1}^{|G|} cost(\phi^{\mathsf{G}}(j)) \cdot b_j$ is the cost of all the boolean conditions needed for the new *oc* gates.

We may now take one of the following quantities as the cost:

– **Basic cost:** C_g, which is just the total cost of the obtained cliques (in this case, the costs of the old *oc* gates are included into C_g).
– **Extended cost:** $\mathsf{C}_g + \mathsf{C}_{oc1} + \mathsf{C}_{oc2} + \mathsf{C}_b$, which takes into account also the cost of the new *oc* gates.

This describes the full cost of the gates involved in the sum, since the bit communication metric of the gates is additive. This sum would not work if we had chosen the number of rounds as the cost.

Inequality constraints The constraints $A\mathbf{x} \leq \mathbf{b}$ state the relations between the variables defined in Sec. 6.5. Since $A\mathbf{x} \geq \mathbf{b}$ can be expressed as $-A\mathbf{x} \leq -\mathbf{b}$, we may as well use $\leq$, $\geq$, and $=$ relations in the constraints.

Building blocks for constraints There are some logical statements that are used several times in the constraints. We will now describe how such statements are encoded as sets of constraints (possibly with some auxiliary variables). We also define special notations for these sets of constraints.

– **Multiplication by a bit:** $z = x \cdot y$ for $x \in \{0,1\}$, $y, z \in \mathbb{R}$, where C is a known upper bound on y. This can be expressed by
  • $C \cdot x + y - z \leq C$;
  • $C \cdot x - y + z \leq C$;
  • $C \cdot x - z \geq 0$.
We denote this set of constraints by $\mathcal{P}(C, x, y, z)$.

– **Threshold:**
$y = \begin{cases} 1 \text{ if } \sum_{x\in\mathcal{X}} x \geq A \\ 0 \text{ otherwise} \end{cases}$
for $\forall x \in \mathcal{X} : x \in \{0,1\}$, $y \in \mathbb{R}$, some constant A. This can be expressed by
  - $\mathcal{P}(1,y,x,z_x)$ for all $x \in \mathcal{X}$, where z_x are fresh variable names;
  - $A \cdot y - \sum_{x\in\mathcal{X}} z_x \leq 0$;
  - $\sum_{x\in\mathcal{X}} x - \sum_{x\in\mathcal{X}} z_x + (A-1)y \geq (A-1)$.

  We denote this set of constraints by $\mathcal{F}(A,\mathcal{X},y)$.

– **Implying inequality:** $(z=1) \implies (x-y \geq A)$ for $z \in \{0,1\}$, $x,y \in \mathbb{R}$, some constant A, where C is a known upper bound on x,y. This can be expressed by
  - $(C+A)\cdot z + y - x \geq C$.

  We denote this constraint by $\mathcal{G}(C,A,x,y,z)$.

The correctness of these sets of constraints is proven in Appendix G.1.

Basic constraints The particular constraints of the integer program are the following.

1. $g_i^j + g_k^j \leq 1$ for $i,k \in G$, $\neg\mathsf{fusable}^{\mathsf{G}}(i,k)$.
   If the gates are not mutually exclusive, then they cannot belong to the same clique.
2. $\sum_{j=1}^{|G|} g_i^j = 1$ for all $i \in G$.
   Each gate belongs to exactly one clique.
3. $g_i^j = 0$ if $\mathsf{op}^G(i) \neq \mathsf{op}^G(j)$.
   The clique and gate operations should match. In order to avoid putting gates of different operations into one clique, we assign operations to the cliques, such that the operation of the j-th clique equals the operation of the j-th gate. The gates are allowed to belong only to the cliques C_j of the same operation as the gate i is.
4. $g_j^j - g_i^j \geq 0$ for all $i \in G$, $j \in G$.
   If the clique C_j is non-empty, then it contains the gate no. j. This makes gate j the representative of C_j.
5. $g_j^j = 1$ for all j such that $cost(\mathsf{op}^G(j)) = 0$.
   We are more interested in fusing the gates with positive cost. Actually, in some cases, even fusing gates of cost 0 can be useful, since it may in turn eliminate some *oc* gates. This constraint makes the optimization faster, although we may lose some valuable solution in this way.
6. (a) $\ell_i - \ell_k \geq 1$ for all $i,k \in G, \mathsf{pred}^{\mathsf{G}}(i,k)$;
   (b) $\mathcal{G}(|G|,0,\ell_i,\ell_j,g_i^j)$ for all $i,j \in G$;
   (c) $\mathcal{G}(|G|,0,\ell_j,\ell_i,g_i^j)$ for all $i,j \in G$;
   (d) $\ell_i \geq 0, \ell_i \leq |G|$.
   After the gates are fused into cliques, their dependencies on each other are not allowed to form cycles. We assign a level ℓ_i to each gate i. If i is a predecessor of k, then $\ell_i < \ell_k$, but to avoid degenerate solutions to the ILP, we introduce some difference between the levels. If a gate i belongs to the clique C_j, then $\ell_i = \ell_j$. We may split the implication $g_i^j = 1 \implies \ell_i = \ell_j$ into two parts $g_i^j = 1 \implies (\ell_i - \ell_j) \geq 0$, $g_i^j = 1 \implies (\ell_j - \ell_i) \geq 0$, reducing them to the constraint $\mathcal{G}$. We take the maximal value for ℓ_i as $|G|$, since we need at most $|G|$ distinct levels even if all the gates are on different levels.
   We would also like to take into account the conditional predecessors.
   (e) $d_j = (1 - g_j^j)$ for all $j \in G$;
   (f) $\mathcal{F}(1,\{d_j\} \cup \{g_i^j \mid i \in G, i \neq j\}, c_j)$ for all $j \in G$;
   The constrains fix the variable c_j so, that $c_j = 1$ iff the gate j is fused with some other gate. That is, either $d_j = 1$ (implying $g_j^j = 0$, or that j belongs to some other clique), or $\sum_{i\in G, i\neq j} g_i^j \geq 1$, implying that there is some other gate belonging to g_j^j.
   (g) $\mathcal{G}(|G|,1,\ell_i,\ell_k,c_i)$ for all $i,k \in G$, $\mathsf{cpred}^{\mathsf{G}}(i,k)$;
   The last constraint states that, if $c_i = 1$ (the gate i is fused with some other gate), then $l_k - l_i \geq 1$ (the gate i should be computed strictly before its conditional predecessor k).

Extended constraints The basic constraints are sufficient to optimize the total cost of the gates, at the same time avoiding cycles (as the greedy optimizations do). Since computing the boolean conditions may also produce some additional costs, we define some more variables with associated constraints that take them into account.

7. We want to define $sc^j_\ell = |\mathsf{args}^{G'}(j)[\ell]| - 1$, and $s^j_\ell = 1$ iff $\mathsf{args}^{G'}(j)[\ell]$ is an oc gate. The ℓ-th argument of C_j requires a new oc gate iff the number of distinct ℓ-th inputs used by the gates $i \in C_j$ is at least 2.
   (a) $\mathcal{F}(1, \{g^j_i \mid i \in G, k = \mathsf{args}^G(i)[\ell]\}, fx^{jk}_\ell)$:
   Define $fx^{jk}_\ell = 1$ iff $k \in \mathsf{args}^G(C_j)[\ell]$;
   (b) $\mathcal{P}(1, fx^{jk}_\ell, g^i_k, e^{jk}_{i\ell})$ for all $k \in V(\mathsf{G})$:
   Define $e^{jk}_{i\ell} = 1$ iff $k \in \mathsf{args}^G(C_j)[\ell]$, and $k \in C_i$;
   (c) $\mathcal{F}(1, \{e^{jk}_{i\ell} \mid k \in V(\mathsf{G})\}, fg^{ji}_\ell)$ for all $i \in G$:
   Define $fg^{ji}_\ell = 1$ iff $\exists k \in C_i :\ k \in \mathsf{args}^G(C_j)[\ell]$;
   (d) $fg^{jk}_\ell = fx^{jk}_\ell$ for $k \in I(\mathsf{G})$;
   Together with (7c), it defines $fg^{jk}_\ell = 1$ iff $C_k \in \mathsf{args}^G(C_j)[\ell]$ for $k \in V(\mathsf{G})$, where for $k \in I(\mathsf{G})$ we denote $C_k = k$.
   (e) $\mathcal{F}(2, \{fg^{jk}_\ell \mid k \in V(\mathsf{G})\}, s^j_\ell)$:
   Define $s^j_\ell = 1$ iff the total number of ℓ-th inputs after fusing the gates of C_j is at least 2.
   (f) $sc^j_\ell = \sum_{k \in V(\mathsf{G})} fg^{jk}_\ell - g^j_j$:
   This defines $sc^j_\ell = |\mathsf{args}^G(C_j)[\ell]| - 1$ for a non-empty clique. If $g^j_j = 0$, then also $\sum_{k \in V(\mathsf{G})} fg^{jk}_\ell = 0$, so it is not counted for empty cliques (we discuss it in more details when we prove the feasibility of the task in App. G.2).
8. Similarly, for the old oc gates, we are going to define $uc^j = |\mathsf{args}^{G'}(j)| - g^j_j$, and $u^j = 1$ iff $|\mathsf{args}^{G'}(j)| > 1$.
   (a) $\mathcal{F}(1, \{fg^{jk}_\ell \mid \ell \in [\mathsf{arity}^G(j)] \cap 2\mathbb{N}\}, fg^{jk})$ for all $j \in G$, $k \in V(\mathsf{G})$, $\mathsf{op}^G(j) = oc$;
   Define $fg^{jk} = 1$ iff C_k is a choice of the oc gate j.
   (b) $\mathcal{F}(2, \{fg^{jk} \mid k \in V(\mathsf{G})\}, u^j)$ for all $j \in G$, $\mathsf{op}^G(j) = oc$;
   Define $u_j = 1$ iff there are at least 2 choices left for the oc gate j.
   (c) $uc^j_\ell = \sum_{k \in V(\mathsf{G})} fg^{jk} - g^j_j$:
   This defines $uc^j_\ell = |\mathsf{args}^{G'}(j)| - g^j_j$ for $\mathsf{op}^G(j) = oc$.
9. We would like to check whether the weakest precondition $\phi^{\mathsf{G}}(j)$ of the gate g_j must be computed. We want to define $b_j = 1$ iff $\phi^{\mathsf{G}}(j)$ is needed.
   (a) $\mathcal{F}(1, \{s^j_\ell \mid \ell \in [\mathsf{arity}^G(j)]\}, t^j)$ for $j \in G$.
   This checks if there will be an oblivious choice of at least one input of the clique C_j. If it is so, then we will need to compute $\phi^{\mathsf{G}}(i)$ for $i \in C_j$.
   (b) $\mathcal{P}(1, g^j_i, t^j, t^j_i)$ for $j \neq i \in G$.
   (c) $t^j_j = 0$ for $j \in G$.
   The variable t^j_i now denotes if the weakest precondition of g_i is needed for the clique C_j. Since we may set one of the choices to negation of all the other choices, we may eliminate one of the weakest preconditions participating in the choice. We choose it to be the choice of gate j, and hence we set $t^j_j = 0$.
   (d) $b_i = \sum_{j \in G} t^j_i$ for $i \in G$.
   Since we know that each gate belongs to exactly one clique, we know that, for a fixed i, we have $t^j_i = 1$ for exactly one j, and so it suffices to sum them up.

Binary constraints Since we are dealing with a mixed integer program, we need to state explicitly that some variables are binary:

$$g^j_i \in \{0,1\} \text{ for all } i, j \in G\ .$$

The statement $sc^j_\ell, uc^j_\ell \in \mathbb{Z}$, and the binariness of all the other variables (except $\ell_j \in \mathbb{R}$) follow from the binariness of g^j_i. We prove it in Appendix G.1. We will need this property in our further proofs of transformation correctness.

Feasibility We want to be sure that the obtained integer linear program indeed has at least one solution.

Theorem 1. *For any $(G,X,Y) \in \mathcal{G}$, if $(A,\mathbf{b},\mathbf{c},\mathcal{I}) = T_{ILP}^{\rightarrow}(G,X,Y)$, then the integer linear programming task*

$$\textit{minimize } \mathbf{c}^T\mathbf{x},\ \textit{s.t } A\mathbf{x} \leq \mathbf{b}, \mathbf{x} \geq \mathbf{0}, x_i \in \{0,1\} \textit{ for } i \in \mathcal{I}$$

has at least one feasible solution.

The proof on this theorem can be found in Appendix G.2. It just shows that it is always possible to take the solution where no gates are fused at all.

6.6 Applying the solution of ILP to G

Let $(G,X,Y) \in \mathcal{G}$ be the initial circuit. Let $Sol(G,X,Y)$ be the set of all feasible solutions to $(A,\mathbf{b},\mathbf{c},\mathcal{I}) = T_{ILP}^{\rightarrow}(G,X,Y)$. Now we define a backwards transformation $T_{ILP}^{\leftarrow} :\ Sol(\mathcal{G}) \times \mathcal{G} \rightarrow \mathcal{G}$, which takes any feasible solution to $(A,\mathbf{b},\mathbf{c},\mathcal{I})$ and applies it to (G,X,Y), forming a new circuit $\mathsf{G}' = (G',X',Y')$. Let $cost : \mathcal{G} \rightarrow \mathbb{Z}$ be function computing the total communication cost of a circuit.

The work of $T_{ILP}^{\leftarrow}$ is pretty straightforward. If just fuses the gates according to the variables g_i^j of the ILP solution that denote which gate belongs to which clique. It introduces all the necessary oblivious choices, and also removes the old oblivious choices that are left with just one choice. An algorithm implementing $T_{ILP}^{\leftarrow}$ and the proof of its correctness are given in App. H.1.

We may use the same algorithm to construct a circuit from a set of cliques obtained from a greedy algorithm of Sec. 6.4, as it can be easily reduced to a linear programming solution of $T_{ILP}^{\rightarrow}(G,X,Y)$. This is stated and proved in Thm. 6 of App. H.3.

We also want to estimate the communication cost of (G',X',Y'). We show that it is indeed the value estimated by the ILP, so it makes sense minimizing it. The corresponding theorem is stated and proved in Appendix H.2.

If we use the greedy algorithms of Sec. 6.4, or use only the basic constrains of the integer program for better convergence, then it may happen that the found solution is worse than the initial one. Indeed, although the total cost of the gates may only decrease, the additional oblivious choices provide computational overhead that is not taken into account by these algorithms and may provide even larger overhead that the eliminated gates' cost was. In Sec. 7, we see that these approaches nevertheless give reasonable execution times, and their optimization times are significantly better in practice.

7 Implementation and evaluation

We have implemented the transformation of the program to a circuit, the optimizations, and the transformation of the circuit according to the obtained set of cliques in SWI-Prolog [34]. The ILP is solved externally by the GLPK solver [13]. The optimized circuit is translated to a Sharemind program for evaluation.

The optimizations have been tested on small programs. Since we are dealing with a relatively new problem, there are no good test sets, and we had to invent some sample programs ourselves. In general, the programs with private conditionals are related to evaluation of decision diagrams with private decisions. We provide five different programs, each with its own specificity. Their pseudocodes are given in App. I.

- loan (31 gates): A simple binary decision tree, which decides whether a person should be given a loan, based on its background data. Such simple applications are often used as an introduction to the decision tree topic. Only the leaves contain assignments, and the optimization is only trying to fuse the comparison operations that make the decisions. Uses only integer operations.
- sqrt (123 gates): Uses binary search to compute the square root of an integer. Since the input is private, it makes a fixed number of iterations. The division by 2 is on purpose inserted into both branches, modified in such a way that it cannot be trivially outlined without arithmetic theory. The optimizer does this outlining by fusing the divisions. Uses only integer operations.

Table 1. Optimization times in seconds for loan

	greed_1	greed_2	greed_3	$\mathsf{lp}_{\mathsf{basic}}$	$\mathsf{lp}_{\mathsf{ext}}$
0	0.046	0.047	0.053	0.097	0.121
1	0.049	0.041	0.042	0.095	0.109
2	0.042	0.043	0.05	0.083	0.115

Table 2. Optimization times in seconds for sqrt

	greed_1	greed_2	greed_3	$\mathsf{lp}_{\mathsf{basic}}$	$\mathsf{lp}_{\mathsf{ext}}$
0	0.523	0.549	0.538	0.78	1.3
1	0.549	0.531	0.544	0.644	0.762
2	0.556	0.526	0.518	0.612	0.654
3	0.536	0.495	0.571	0.584	0.596
4	0.55	0.51	0.511	0.623	0.593

Table 3. Optimization times in seconds for driver

	greed_1	greed_2	greed_3	$\mathsf{lp}_{\mathsf{basic}}$	$\mathsf{lp}_{\mathsf{ext}}$
0	0.119	0.11	0.125	0.181	0.381
1	0.084	0.082	0.081	0.159	0.164
2	0.075	0.083	0.076	0.138	0.157
3	0.082	0.077	0.078	0.118	0.171

Table 4. Optimization times in seconds for stats

	greed_1	greed_2	greed_3	$\mathsf{lp}_{\mathsf{basic}}$	$\mathsf{lp}_{\mathsf{ext}}$
0	0.185	0.181	0.185	0.292	16.291
1	0.115	0.123	0.12	0.17	0.262
2	0.103	0.103	0.107	0.154	0.226
3	0.104	0.106	0.104	0.144	0.22
4	0.103	0.11	0.106	0.142	0.214
5	0.107	0.103	0.107	0.156	0.222

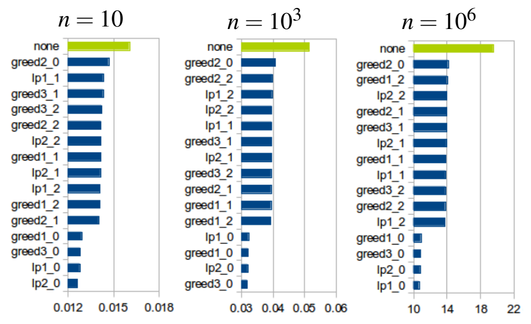
Table 5. Optimization times in seconds for erf

	greed_1	greed_2	greed_3	$\mathsf{lp}_{\mathsf{basic}}$	$\mathsf{lp}_{\mathsf{ext}}$
0	43.405	43.111	–	47.03	47.601
1	5.532	5.523	–	–	–
2	3.581	3.649	8.468	8.136	–
3	3.243	3.267	6.082	6.275	–
4	2.719	2.754	2.702	3.496	–
5	2.613	2.626	2.608	3.088	525.34
6	2.559	2.583	2.578	2.983	15.248
7	2.604	2.64	2.838	3.008	15.27

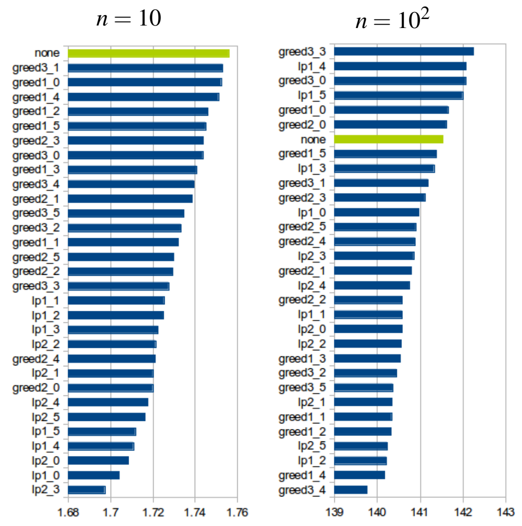
– driver (53 gates): We took the decision tree that is applied to certain parameters of a piece of music in order to check how well it wakes up a sleepy car driver [23], assuming a privacy-preserving setting of this task. In this tree, some decisions require more complex operations, such as logarithms and inverses (computing Shannon entropy), so it was interesting to try to fuse them. Works with floating point arithmetic [18].
– stats (68 gates): The motivation for the problem is that choosing a particular statistical test for the analysis may depend on the type of data (ordinal, binary). Here we assume that the decision bits (which analysis to choose) are already given, but are private. The complex computation starts in leaves, where a particular statistical test is chosen. It chooses amongst the Student t-test, the Wilcoxon test, the Welch test, and the χ^2 test, whose privacy-preserving implementations are taken from [3]. Works with floating point arithmetic.
– erf (335 gates): The program evaluates the error function of a floating point number, which is represented as a triple (sign, significand, exponent) of integers [18]. The implementation is taken from [20]. In this program, the method chosen to compute the answer depends on the range in which the initial input is located, and since the input is private, this choice cannot be leaked.

All our programs are vectorized. We treat vector operations as single gates, so that optimizing 10^6 operations per gate would be feasible. For simplicity, we assumed that all vectors in the program have the same length. Fusing together vector operations of different length can be treated as a future work.

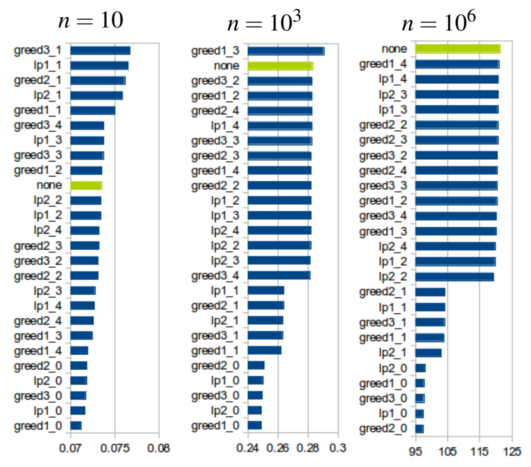
We ran the optimizer on a Lenovo X201i laptop with a 4-core Intel Core i3 2.4GHz processor and 4GB of RAM running Ubuntu 12.04. The execution times are given in the Tables 1- 5. The rows correspond to different subcircuit depths, which are constructed as described in Sec. 6.3. We tried all possible depths, until it was not possible to increase the depth anymore since all the subgraphs would have become unique. The columns correspond to the different optimization techniques. The columns greed_1, greed_2, and greed_3 are the three different greedy strategies that are described in Sec. 6.4. The columns $\mathsf{lp}_{\mathsf{basic}}$ and $\mathsf{lp}_{\mathsf{ext}}$ correspond to the mixed integer programming approach of Sec. 6.5, using the basic and the extended constraints respectively.



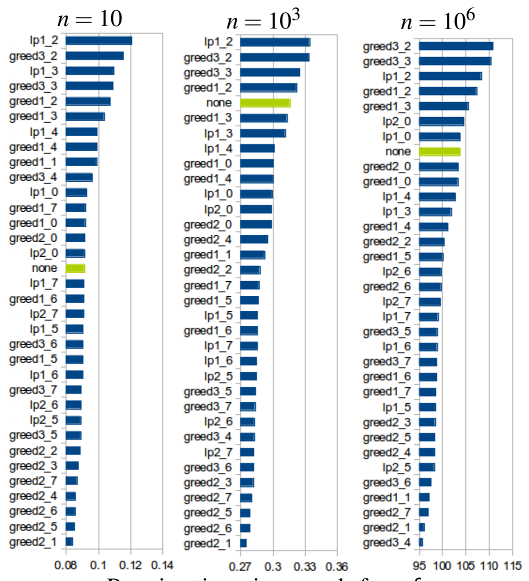
Running times in seconds for loan



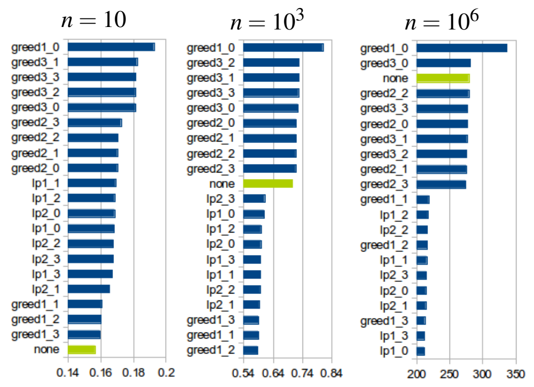
Running times in seconds for stats



Running times in seconds for sqrt



Running times in seconds for erf



Running times in seconds for driver

Fig. 7: Running times in seconds

We compiled the optimized graphs into programs, executed them on Sharemind (three servers on a local 1Gbps network; the speed of the network is the bottleneck in these tests) and measured their running time. The runtime benchmarks can be found in the Figure 7. The X-axis corresponds to different optimizations, including all the combinations of the 5 strategies with all used subcircuit depths, sorted by runtime. The Y-axis represents the running times. The parameter n is the vector length — the number of executions run in parallel. We see that, since the computation time depends not only on the communication, but also on the number of rounds, our optimization was rather harmful on inputs of small size. However, as the total amount of communication and computation increases, our optimized programs are becoming more advantageous.

For driver, sqrt, and loan, we see that the optimized programs are clearly grouped by their running times. The differences inside a single group are insignificant, so we may treat the entire group as having the same cost. For stats and erf, we do not see such a partitioning. The results are too varying, and hence we cannot claim if the optimizations were harmful or useful. Running stats on 10 inputs shows advantage of $\mathsf{lp}_{\mathsf{basic}}$and $\mathsf{lp}_{\mathsf{ext}}$, but it gets lost on 100 inputs. This most probably indicates that the advantage has come from fusing together non-vector operations which are less significant for larger inputs.

In general, it is preferable not to merge the initial gates into subcircuits (take depth 0). The greedy strategies work quite well for the given programs, but their results are too unpredictable and can be very good as well as very bad. The results of ILP are in general better. In practice, it would be good to estimate the approximate runtime of the program before it is actually executed, so that we could take the best variant. Our optimizations seem to be most useful for library functions, where several different optimized versions can be compiled and benchmarked before choosing the final one.

8 Conclusion

We have presented an optimization for programs written in an imperative language with private conditions. The reduction and the optimization are not restricted to any specific privacy-preserving platforms. We have optimized and benchmarked some programs on Sharemind.

Currently, we are using arithmetic blackbox operations as the gates of the circuit. We have chosen arithmetic black boxes as subcircuits, since then it should be relatively easy to transform programs without knowing how exactly the blackbox operations are constructed (inside, they may actually be some asymmetric protocols that are not decomposable further). As a future work, we could try to decompose the operations as much as possible, getting an arithmetic (or a boolean) circuit, possibly allowing to fuse together some parts of different blackbox functions. Taking into account vectors of different lengths would be another useful improvement.

References

1. *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*. IEEE Computer Society, 2014.
2. Marina Blanton, Aaron Steele, and Mehrdad Aliasgari. Data-oblivious graph algorithms for secure computation and outsourcing. In Kefei Chen, Qi Xie, Weidong Qiu, Ninghui Li, and Wen-Guey Tzeng, editors, *8th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '13, Hangzhou, China - May 08 - 10, 2013*, pages 207–218. ACM, 2013.
3. Dan Bogdanov, Liina Kamm, Sven Laur, and Ville Sokk. Rmind: a tool for cryptographically secure statistical analysis. Cryptology ePrint Archive, Report 2014/512, 2014. `http://eprint.iacr.org/2014/512`.
4. Dan Bogdanov, Peeter Laud, and Jaak Randmets. Domain-polymorphic programming of privacy-preserving applications. In Alejandro Russo and Omer Tripp, editors, *Proceedings of the Ninth Workshop on Programming Languages and Analysis for Security, PLAS@ECOOP 2014, Uppsala, Sweden, July 29, 2014*, page 53. ACM, 2014.
5. Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A framework for fast privacy-preserving computations. In Sushil Jajodia and Javier López, editors, *ESORICS*, volume 5283 of *Lecture Notes in Computer Science*, pages 192–206. Springer, 2008.
6. Dan Bogdanov, Margus Niitsoo, Tomas Toft, and Jan Willemson. High-performance secure multi-party computation for data mining applications. *Int. J. Inf. Sec.*, 11(6):403–418, 2012.
7. Martin Burkhart, Mario Strasser, Dilip Many, and Xenofontas Dimitropoulos. SEPIA: Privacy-preserving aggregation of multi-domain network events and statistics. In *USENIX Security Symposium*, pages 223–239, Washington, DC, USA, 2010.

8. Ronald Cramer, Ivan Damgård, and Ueli M. Maurer. General secure multi-party computation from any linear secret-sharing scheme. In Bart Preneel, editor, *EUROCRYPT*, volume 1807 of *Lecture Notes in Computer Science*, pages 316–334. Springer, 2000.
9. Ivan Damgård, Martin Geisler, Mikkel Krøigaard, and Jesper Buus Nielsen. Asynchronous multiparty computation: Theory and implementation. In Stanislaw Jarecki and Gene Tsudik, editors, *Public Key Cryptography - PKC 2009, 12th International Conference on Practice and Theory in Public Key Cryptography, Irvine, CA, USA, March 18-20, 2009. Proceedings*, volume 5443 of *Lecture Notes in Computer Science*, pages 160–179. Springer, 2009.
10. Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO*, volume 7417 of *Lecture Notes in Computer Science*, pages 643–662. Springer, 2012.
11. Daniel Demmler, Thomas Schneider, and Michael Zohner. ABY - A framework for efficient mixed-protocol secure two-party computation. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2014*. The Internet Society, 2015.
12. Martin Franz, Andreas Holzer, Stefan Katzenbeisser, Christian Schallhart, and Helmut Veith. CBMC-GC: an ANSI C compiler for secure two-party computations. In Albert Cohen, editor, *Compiler Construction - 23rd International Conference, CC 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, volume 8409 of *Lecture Notes in Computer Science*, pages 244–249. Springer, 2014.
13. GLPK. GNU Linear Programming Kit.
`http://www.gnu.org/software/glpk`.
14. Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In Alfred V. Aho, editor, *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA*, pages 218–229. ACM, 1987.
15. Oded Goldreich and Rafail Ostrovsky. Software Protection and Simulation on Oblivious RAMs. *J. ACM*, 43(3):431–473, 1996.
16. S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. Secure two-party computation in sublinear (amortized) time. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, pages 513–524. ACM, 2012.
17. Koki Hamada, Ryo Kikuchi, Dai Ikarashi, Koji Chida, and Katsumi Takahashi. Practically Efficient Multi-party Sorting Protocols from Comparison Sort Algorithms. In *Proc. of ICISC'12*, volume 7839 of *LNCS*, pages 202–216. Springer, 2013.
18. Liina Kamm and Jan Willemson. Secure floating point arithmetic and private satellite collision analysis. *International Journal of Information Security*, pages 1–18, 2014.
19. W. Sean Kennedy, Vladimir Kolesnikov, and Gordon Wilfong. Overlaying circuit clauses for secure computation. Cryptology ePrint Archive, Report 2016/685, 2016. `http://eprint.iacr.org/2016/685`.
20. Toomas Krips and Jan Willemson. Hybrid model of fixed and floating point numbers in secure multiparty computations. In *International Conference on Information Security*, pages 179–197. Springer, 2014.
21. Chang Liu, Yan Huang, Elaine Shi, Jonathan Katz, and Michael W. Hicks. Automating efficient ram-model secure computation. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014* [1], pages 623–638.
22. Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. Oblivm: A programming framework for secure computation. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 359–376. IEEE Computer Society, 2015.
23. Ning-Han Liu, Cheng-Yu Chiang, and Hsiang-Ming Hsu. Improving driver alertness through music selection using a mobile eeg to detect brainwaves. *Sensors*, 13(7):8199–8221, 2013.
24. D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay - a secure two-party computation system. In *Proceedings of the 13th USENIX Security Symposium*, pages 287–302, Berkeley, CA, USA, 2004. USENIX Association.
25. John C. Mitchell, Rahul Sharma, Deian Stefan, and Joe Zimmerman. Information-flow control for programming on encrypted data. In Stephen Chong, editor, *25th IEEE Computer Security Foundations Symposium, CSF 2012, Cambridge, MA, USA, June 25-27, 2012*, pages 45–60. IEEE Computer Society, 2012.
26. Janus Dam Nielsen and Michael I. Schwartzbach. A domain-specific programming language for secure multiparty computation. In Michael W. Hicks, editor, *Proceedings of the 2007 Workshop on Programming Languages and Analysis for Security, PLAS 2007, San Diego, California, USA, June 14, 2007*, pages 21–30. ACM, 2007.
27. Jérémy Planul and John C. Mitchell. Oblivious program execution and path-sensitive non-interference. In *2013 IEEE 26th Computer Security Foundations Symposium, New Orleans, LA, USA, June 26-28, 2013*, pages 66–80. IEEE, 2013.
28. Pille Pruulmann-Vengerfeldt, Liina Kamm, Riivo Talviste, Peeter Laud, and Dan Bogdanov. Capability Model, March 2012. UaESMC Deliverable 1.1.
29. Aseem Rastogi, Matthew A. Hammer, and Michael Hicks. Wysteria: A programming language for generic, mixed-mode multiparty computations. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014* [1], pages 655–670.

30. A. Schrijver. *Theory of Linear and Integer Programming*. Wiley Series in Discrete Mathematics & Optimization. John Wiley & Sons, 1998.
31. Axel Schröpfer, Florian Kerschbaum, and Günter Müller. L1 - an intermediate language for mixed-protocol secure computation. In *Proceedings of the 35th Annual IEEE International Computer Software and Applications Conference, COMPSAC 2011, Munich, Germany, 18-22 July 2011*, pages 298–307. IEEE Computer Society, 2011.
32. Xiao Wang, S. Dov Gordon, Allen McIntosh, and Jonathan Katz. Secure computation of mips machine code. Cryptology ePrint Archive, Report 2015/547, 2015. `http://eprint.iacr.org/2015/547`.
33. Xiao Shaun Wang, Yan Huang, T.-H. Hubert Chan, Abhi Shelat, and Elaine Shi. SCORAM: oblivious RAM for secure computation. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 191–202. ACM, 2014.
34. Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. SWI-Prolog. *Theory and Practice of Logic Programming*, 12(1-2):67–96, 2012.
35. Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *23rd Annual Symposium on Foundations of Computer Science, Chicago, Illinois, USA, 3-5 November 1982*, pages 160–164. IEEE Computer Society, 1982.
36. Samee Zahur and David Evans. Obliv-c: A language for extensible data-oblivious computation. Cryptology ePrint Archive, Report 2015/1153, 2015. `http://eprint.iacr.org/2015/1153`.
37. Yihua Zhang, Aaron Steele, and Marina Blanton. PICCO: a general-purpose compiler for private distributed computation. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 813–826. ACM, 2013.

A Semantics of The initial imperative language

Let Var be the set of program variables, and Val the set of values that the variables may take. Let $State : Var \to Val$ be a *program state*, which assigns a value to each program variable.

The semantics $[\![\cdot]\!]$ defines how executing a program statement modifies the state. Let P be a program written in a language whose syntax is given in Sec. 5.1. We define $[\![P]\!] : State \to State$ as follows.

– $[\![\texttt{skip}]\!]\, s = s$;
– $[\![y := e]\!]\, s = s[y \leftarrow ([\![e]\!]\, s)]$;
– $[\![S_1 \,;\, S_2]\!]\, s = [\![S_2]\!]\,([\![S_1]\!]\, s)$;
– $[\![\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2]\!]\, s = \begin{cases} [\![S_1]\!]\, s \text{ if } [\![b]\!]\, s \neq 0 \\ [\![S_2]\!]\, s \text{ if } [\![b]\!]\, s = 0 \end{cases}$;

For expressions e, we have $[\![e]\!] : State \to Val$.

– $[\![x]\!]\, s = s(x)$ if $x \in dom(s)$;
– $[\![f(e_1,\ldots,e_k)]\!]\, s = [\![f]\!]([\![e_1]\!]\, s, \ldots, [\![e_n]\!]\, s)$, where $[\![f]\!]$ is defined by the underlying SMC platform of the programming language.

B Circuit Evaluation and Composition

Let $\mathsf{G} = (G, X, Y)$ be defined as in Sec. 5.2. In this section we formally define the circuit evaluation and composition.

B.1 Circuit Evaluation

First of all, we formally define the circuit evaluation on its inputs, without treating it as a part of a program.

Definition 2. *Let $W : I(\mathsf{G}) \to Val$ be an arbitrary valuation of the input wires of G. Let $u \in V(\mathsf{G})$. We define $[\![G]\!]\, W$ inductively on $|G|$.*

– $[\![\emptyset]\!]\, W\, u = W(u)$*, which is correct since* $u \in V(\mathsf{G}) = I(\mathsf{G}) \cup \emptyset = I(\mathsf{G})$.
– $[\![G \cup (v, f, [v_1, \ldots, v_n])]\!]\, W\, u$
$= (u \neq v\ ?\ [\![G]\!]\, W\, u\ :\ [\![f]\!]([\![G]\!]\, W\, v_1, \ldots, [\![G]\!]\, W\, v_n))$.

In Sec. 5.2, we have defined a special *oblivious choice* gate. We now formally define its evaluation.

Definition 3. *Let $b_1,\dots,b_n \in V(\mathsf{G})$ be such that, for any input wire valuation $W : I(\mathsf{G}) \to Val$, $\sum_{i=1}^{n} [\![G]\!]\, W\, b_i \in \{0,1\}$, and $\forall i :\ [\![G]\!]\, W\, b_i \in \{0,1\}$. The output of an* oblivious choice gate $(v, oc, [b_1, v_1, \dots, b_n, v_n])$ *is defined as*

$$[\![G]\!]\, W\, v = \sum_{i=1}^{n} ([\![G]\!]\, W\, b_i) \cdot ([\![G]\!]\, W\, v_i) \ .$$

In this definition, we allow $\sum_{i=1}^{n} [\![G]\!]\, W\, b_i \in \{0,1\}$, although $\sum_{i=1}^{n} [\![G]\!]\, W\, b_i = 1$ may seem more reasonable, especially if we want to treat one of the choices as the *default* choice that is the negation of all the other choices. The formal reason why we allow $\sum_{i=1}^{n} [\![G]\!]\, W\, b_i = 0$ is that, since we use the weakest preconditions of gates for making the choice, it may happen that all the preconditions of the fused gates are false. In this case, the output of the oblivious choice gate is not going to matter because later there will be some other oblivious choice gate that drops it. Hence it does not matter whether it outputs 0 or some particular v_i. Therefore, one of the choices is allowed to be set to a default choice anyway, and there is no difference whether we allow $\sum_{i=1}^{n} [\![G]\!]\, W\, b_i \in \{0,1\}$ or just $\sum_{i=1}^{n} [\![G]\!]\, W\, b_i = 1$. The first option just makes the presentation simpler.

Since we use the circuit evaluation as a part of the program execution, we must translate it to a program statement. Let v be a name of a circuit wire. We extend the syntax of the program with a new type of statement.

$$\begin{aligned} exp &::= \texttt{eval}\ (\ gate^*\ ,\ (x,\ v)^*\ ,(\ x,\ v\ )^*\ ) \\ gate &::= (\ v\ ,\ abb\ ,[\ v^*]\ ) \end{aligned}$$

The statement $\texttt{eval}(G,X,Y)$ evaluates the gates G, where X assigns the input values to the input wires $I(\mathsf{G})$ of G, and Y defines the set of output wires $O(\mathsf{G})$ from which the values have to be eventually taken. The gates of G are evaluated according to the definition of $[\![G]\!]$.

The evaluation statement is defined as

$$[\![\texttt{eval}(G,X,Y)]\!]\, s = \texttt{upd}(Y \circ [\![G]\!](s \circ X), s)$$

where

$$\texttt{upd}(s', s) = x \in dom(s')\ ?\ s'(x)\ :\ s(x)$$

is the result of updating the state s with the variable valuations of some other state s'.

As a shorthand notation, we write $\texttt{eval}(\mathsf{G}) = \texttt{eval}(G,X,Y)$ for $\mathsf{G} = (G,X,Y)$.

B.2 Circuit composition

Lemma 1. *Let $\mathsf{G}_1 = (G_1,X_1,Y_1)$ and $\mathsf{G}_2 = (G_2,X_2,Y_2)$ where*

1. *$V(\mathsf{G}_1) \cap V(\mathsf{G}_2) = \emptyset$;*
2. *$dom(Y_1) \cap dom(Y_2) = \emptyset$;*
3. *$dom(Y_1) \cap ran(X_2) = \emptyset$.*

We can define a new circuit $\mathsf{G} = (G = G_1 \cup G_2, X = X_1 \cup X_2, Y = Y_1 \cup Y_2)$ such that

$$\forall s :\ [\![\mathit{eval}(G,X,Y)]\!]\, s = [\![\mathit{eval}(\mathsf{G}_2)]\!]([\![\mathit{eval}(\mathsf{G}_1)]\!]\, s) \ .$$

Proof: Let us write the expressions out.

- $[\![\texttt{eval}(G_1,X_1,Y_1)]\!]\, s = \texttt{upd}(Y_1 \circ [\![G_1]\!](s \circ X_1), s)$;
- $[\![\texttt{eval}(G_2,X_2,Y_2)]\!]\, s = \texttt{upd}(Y_2 \circ [\![G_2]\!](s \circ X_2), s)$;
- $[\![\texttt{eval}(G,X,Y)]\!]\, s = \texttt{upd}((Y_1 \cup Y_2) \circ [\![G_1 \cup G_2]\!](s \circ (X_1 \cup X_2)), s)$.

Let $y \in Var$. Let $s' = \texttt{upd}(Y \circ [\![G]\!](s \circ X), s)$.

- If $y \notin dom(Y_1) \cup dom(Y_2)$, then $s'(y) = \mathtt{upd}((Y_1 \cup Y_2) \circ [\![G_1 \cup G_2]\!](s \circ (X_1 \cup X_2)), s)(y) = s(y)$, and also $\mathtt{upd}(Y_2 \circ [\![G_2]\!](\mathtt{upd}(Y_1 \circ [\![G_1]\!](s \circ X_1), s) \circ X_2), \mathtt{upd}(Y_1 \circ [\![G_1]\!](s \circ X_1), s)) = \mathtt{upd}(Y_1 \circ [\![G_1]\!](s \circ X_1), s) = s(y)$.
- If $y \in dom(Y_i)$, then there is $u \in V(\mathsf{G}_i)$ such that $Y_i(y) = u$. For any input wire valuations $W_1 : I(\mathsf{G}_1) \to Val$, $W_2 : I(\mathsf{G}_2) \to Val$, and since $V(\mathsf{G}_1) \cap V(\mathsf{G}_2) = \emptyset$, we can define $W_1 \cup W_2 = W : I(\mathsf{G}) \to Val$. We have
  $[\![G_1 \cup G_2]\!]\, W\, u = \begin{cases} [\![G_1]\!]\, W_1\, u \text{ if } u \in V(\mathsf{G}_1) \\ [\![G_2]\!]\, W_2\, u \text{ if } u \in V(\mathsf{G}_2) \end{cases}$
  1. If $y \in dom(Y_1) \setminus dom(Y_2)$, then

  $$\begin{aligned} s'(y) &= \mathtt{upd}(Y \circ [\![G]\!](s \circ X), s)(y) \\ &= \mathtt{upd}((Y_1 \cup Y_2) \circ [\![G_1 \cup G_2]\!](s \circ (X_1 \cup X_2)), s)(y) \\ &= \mathtt{upd}(Y_1 \circ [\![G_1]\!](s \circ X_1), s)(y) \ \ . \end{aligned}$$

  Let $s'' = \mathtt{upd}(Y_1 \circ [\![G_1]\!](s \circ X_1), s)$. Since updating the variables of $dom(Y_2)$ does not affect the value of $y \notin dom(Y_2)$, we have

  $$\begin{aligned} s'(y) &= s''(y) \\ &= \mathtt{upd}(Y_2 \circ [\![G_2]\!](s'' \circ X_2), s'')(y) \\ &= [\![\mathtt{eval}(\mathsf{G}_2)]\!]([\![\mathtt{eval}(\mathsf{G}_1)]\!]\, s(y) \ \ . \end{aligned}$$

  2. Let $y \in dom(Y_2)$.

  $$\begin{aligned} s'(y) &= \mathtt{upd}(Y \circ [\![G]\!](s \circ X), s)(y) \\ &= \mathtt{upd}(Y_2 \circ [\![G_2]\!](s \circ X_2), s)\, y \ \ . \end{aligned}$$

  Let $s'' = \mathtt{upd}(Y_1 \circ [\![G_1]\!](s \circ X_1), s)$. Note that, for all $x \in ran(X_2)$, we have $s(x) = s''(x)$ due to $dom(Y_1) \cap ran(X_2) = \emptyset$. We get

  $$s'(y) = \mathtt{upd}(Y_2 \circ [\![G_2]\!](s'' \circ X_2), s)(y) \ \ .$$

  Also, if $y \in dom(Y_2)$, then $s(y) = s''(y)$ due to $dom(Y_1) \cap dom(Y_2) = \emptyset$. We get

  $$\begin{aligned} s'(y) &= \mathtt{upd}(Y_2 \circ [\![G_2]\!](s'' \circ X_2), s'')(y) \\ &= [\![\mathtt{eval}(\mathsf{G}_2)]\!]([\![\mathtt{eval}(\mathsf{G}_1)]\!]\, s\, y \ \ . \end{aligned}$$

We have proven the claim for all $y \in Var$. □

C Transformations of Programs to Circuits

Formally, we define a transformation $T_P : \ prog \ \to \ prog$ that substitutes all private conditionals of the initial program with circuit evaluations. The transformation T_P does not modify the statements outside of the private conditionals. If it is applied to a private conditional, it uses an auxiliary transformation $T_C : \ statement \ \to \ \mathcal{G}$ to construct a circuit, and substitutes the private conditional block with $\mathtt{eval}(G,X,Y)$, where (G,X,Y) are generated by T_C. We give the recursive definitions of T_P and T_C.

- $T_P(S_1 \ ; \ S_2) = T_P(S_1) \ ; \ T_P(S_2)$.
- $T_P(a := b) = (a := b)$.
- $T_P(\mathtt{if}\ b\ \mathtt{then}\ S_1\ \mathtt{else}\ S_2) = \mathtt{if}\ b\ \mathtt{then}\ T_P(S_1)\ \mathtt{else}\ T_P(S_2)$ for a public b.
- $T_P(\mathtt{if}\ b\ \mathtt{then}\ S_1\ \mathtt{else}\ S_2) = \mathtt{eval}(G,X,Y)$ where $(G,X,Y) = T_C(\mathtt{if}\ b\ \mathtt{then}\ S_1\ \mathtt{else}\ S_2)$ for a private b.

The transformation T_C creates the circuits corresponding to the computation inside the private conditionals, and arranges the mappings X and Y that establish relations between the circuit wires and the program variables.

- $T_C(\mathtt{skip}) = (\emptyset, \emptyset, \emptyset)$.

– $T_C(y := x) = (\emptyset, \{v \leftarrow x\}, \{y \leftarrow v\})$, where x is a variable name or a constant. There are no gates, and the value for y is taken directly from the wire to which the value of x is assigned.
– $T_C(y := f(x_1, \ldots, x_n)) = (G, X, Y)$, where
  - $x_1, \ldots, x_n$ are program variables and constants;
  - f is some arithmetic blackbox function defined in the programming language;
  - $G = (w, [\![f]\!], [v_1, \ldots, v_n])$ is a gate computing f;
  - $X = \{v_1 \leftarrow x_1, \ldots, v_n \leftarrow x_n\}$;
  - $Y = \{y \leftarrow w\}$.
– $T_C(y := f(e_1, \ldots, e_n))$
  $= T_C(y_{i_1} := e_{i_1} \; ; \; \ldots \; ; \; y_{i_n} := e_{i_n}; \; y := f(y_1, \ldots, y_n))$, where
  - $\{e_{i_1}, \ldots, e_{i_n}\} \subseteq \{e_1, \ldots, e_n\}$ are compound expressions (not variables/constants);
  - $y_i = e_i$ for $i \notin \{i_1, \ldots, i_n\}$.
– $T_C(S_1 \; ; \; S_2) = (G, X, Y)$ where
  - $(G_i, X_i, Y_i) = T_C(S_i)$ for $i \in \{1, 2\}$;
  - $X = X_1 \cup X_2'$ where $X_2' = (X_2 \setminus \{v \leftarrow x \mid x \in dom(Y_1)\})$: the inputs of both X_1 and X_2 will be needed during the computation, but the variables of X_2 that are modified by S_1 should be taken from the output of S_1's circuit instead.
  - $Y = Y_2 \cup Y_1'$ where $Y_1' = (Y_1 \setminus \{y \leftarrow w \mid y \in dom(Y_2)\})$: all the variables that are modified throughout the execution of $S_1 \; ; \; S_2$ are in Y. If a variable is modified in both S_1 and S_2, its value is taken from the output of S_2.
  - Now G_1 and G_2 should be combined. Take $G = G_1 \cup G_2'$ where $G_2' = G_2[\{v \leftarrow w \mid (v \leftarrow x) \in X_2, (x \leftarrow w) \in Y_1\})]$. This connects the inputs of G_2 to the outputs of G_1.
– $T_C(\texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2) = T_C(S)$ where
  - $(G_i, X_i, Y_i) = T_C(S_i)$ for $i \in \{1, 2\}$,
  - $Y_i' = \{z_{iy} \leftarrow w \mid (y \leftarrow w) \in Y_i\}$ for $i \in \{1, 2\}$: this renames the variables y of Y_i by introducing new variable names z_{iy},
  - $S = \begin{cases} b_1 := b; \\ b_2 := (1 - b); \\ S_1' = S_1[(y \leftarrow z_{1y}) \in Y_1' \mid \exists w \; (y \leftarrow w) \in Y_1]; \\ S_2' = S_2[(y \leftarrow z_{2y}) \in Y_2' \mid \exists w \; (y \leftarrow w) \in Y_2]; \\ y := oc(b_1, r(1, y), b_2, r(2, y)) \; \forall y \in Y; \end{cases}$,
  - $r(i, y) = \begin{cases} Y_i'(w) \text{ if } (y \leftarrow w) \in Y_i \\ y \text{ otherwise} \end{cases}$.

  This computes S_1 and S_2 sequentially (renaming all the assigned variables in order to ensure that there are no conflicts), and then applies a new binary oblivious choice (b or $\neg b$) to the outputs of G_1 and G_2. All the outputs of all the branches should be available in an oblivious selection. If any variables are modified only in one of the branches, they should be output also by the circuit that corresponds to the other branch, in order to make them indistinguishable. Such variables y are just copied from the input directly.

As the result, if P is the initial program with private conditions, $T_P(P)$ is a program without private conditions, but with some instances of function call $\texttt{eval}(G, X, Y)$ in its code. We need to prove that $T_P(P)$ does the same computation as P.

Theorem 2. *For any program P, $s \in State$, $[\![P]\!]\, s = [\![T_P(P)]\!]\, s$.*

Proof It is sufficient to prove the correctness of T_P, which it turn will require to prove the correctness of T_C. Since the transformations T_P and T_C are defined inductively, we prove their correctness also inductively.

– $T_P(S_1 \; ; \; S_2) = T_P(S_1) \; ; \; T_P(S_2)$.

$$\begin{aligned} [\![T_P(S_1 \; ; \; S_2)]\!]\, s &= [\![T_P(S_1) \; ; \; T_P(S_2)]\!]\, s \\ &= [\![T_P(S_2)]\!]([\![T_P(S_1)]\!]\, s) \\ &= [\![S_2]\!]([\![S_1]\!]\, s) = [\![(S_1 \; ; \; S_2)]\!]\, s \; . \end{aligned}$$

– $T_P(a := b) = (a := b)$.
  $[\![T_P(a := b)]\!]\,s = [\![(a := b)]\!]\,s$.
– $T_P(\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2) = \texttt{if}\ b\ \texttt{then}\ T_P(S_1)\ \texttt{else}\ T_P(S_2)$ for a public b.

$$\begin{aligned} S &= [\![\texttt{if}\ b\ \texttt{then}\ T_P(S_1)\ \texttt{else}\ T_P(S_2)]\!]\,s \\ &= \begin{cases} [\![T_P(S_1)]\!]\,s \text{ if } [\![b]\!]\,s \neq 0 \\ [\![T_P(S_2)]\!]\,s \text{ if } [\![b]\!]\,s = 0 \end{cases} \\ &= \begin{cases} [\![S_1]\!]\,s \text{ if } [\![b]\!]\,s \neq 0 \\ [\![S_2]\!]\,s \text{ if } [\![b]\!]\,s = 0 \end{cases} \\ &= [\![\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2]\!]\,s\ \ . \end{aligned}$$

– $T_P(\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2) = \texttt{eval}(G,X,Y)$ where $(G,X,Y) = T_C(\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2)$ for a private b. Assuming the correctness of T_C, we have

$$[\![\texttt{eval}(G,X,Y)]\!](s) = [\![\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2]\!]\,s\ \ .$$

We now prove the correctness of T_C.

– $T_C(skip) = (\emptyset,\emptyset,\emptyset)$.
  $[\![\texttt{eval}(\emptyset,\emptyset,\emptyset)]\!]\,s = \texttt{upd}(\emptyset,s) = s = [\![skip]\!]\,s$.
– $T_C(y := x) = (\emptyset, \{v \leftarrow x\}, \{y \leftarrow v\})$, where x is a variable name or a constant. There are no gates, and the value for y is taken directly from the wire to which the value of x is assigned.

$$\begin{aligned} [\![\texttt{eval}(\emptyset,\{v \leftarrow x\},\{y \leftarrow v\})]\!]\,s &= \texttt{upd}((y \leftarrow v) \circ [\![\emptyset]\!](s \circ (v \leftarrow x)), s) \\ &= \texttt{upd}((y \leftarrow v) \circ s \circ (v \leftarrow x), s) \\ &= s[y \leftarrow s(x)] \\ &= [\![y := x]\!]\,s\ \ ; \end{aligned}$$

– $T_C(y := f(x_1,\dots,x_n)) = (G,X,Y)$, where
  - $x_1,\dots,x_n$ are program variables and constants;
  - f is some arithmetic blackbox function defined in the programming language;
  - $G = (w, [\![f]\!], [v_1,\dots,v_n])$ is a gate computing f;
  - $X = \{v_1 \leftarrow x_1, \dots, v_n \leftarrow x_n\}$;
  - $Y = \{y \leftarrow w\}$.

$$\begin{aligned} [\![\texttt{eval}(G,\{v_1 \leftarrow x_1,\dots,v_n \leftarrow x_n\},\{y \leftarrow w\})]\!]\,s &= \texttt{upd}((y \leftarrow w) \circ [\![G]\!](s \circ \{v_1 \leftarrow x_1,\dots,v_n \leftarrow x_n\})), s) \\ &= \texttt{upd}((y \leftarrow w) \circ s'[w \leftarrow f(x_1,\dots,x_n)], s) \\ &= \texttt{upd}((y \leftarrow f(x_1,\dots,x_n)), s) \\ &= s[y \leftarrow f(x_1,\dots,x_n)] \\ &= [\![(y := f(x_1,\dots,x_n)]\!]\,s\ \ . \end{aligned}$$

– $T_C(S_1\ ;\ S_2) = (G,X,Y)$ where
  - $(G_i,X_i,Y_i) = T_C(S_i)$ for $i \in \{1,2\}$;
  - $X = X_1 \cup X_2'$ where $X_2' = (X_2 \setminus \{v \leftarrow x \mid x \in dom(Y_1)\})$: the inputs of both X_1 and X_2 will be needed during the computation, but the variables of X_2 that are modified by S_1 should be taken from the output of S_1's circuit instead.
  - $Y = Y_2 \cup Y_1'$ where $Y_1' = (Y_1 \setminus \{y \leftarrow w \mid y \in dom(Y_2)\})$: all the variables that are modified throughout the execution of $S_1\ ;\ S_2$ are in Y. If a variable is modified in both S_1 and S_2, its value is taken from the output of S_2.

- Now G_1 and G_2 should be combined. Take $G = G_1 \cup G'_2$ where

$$G'_2 = G_2[\{v \leftarrow w \mid (v \leftarrow x) \in X_2, (x \leftarrow w) \in Y_1\})] \ .$$

  This connects the inputs of G_2 to the outputs of G_1.

Denote:
${X_2}^{Y_1} = \{(v \leftarrow x) \in X_2 \mid x \in dom(Y_1)\}$,
${Y_2}^{Y_1} = \{(y \leftarrow w) \in Y_2 \mid y \in dom(Y_1)\}$.
We can now write

$$\begin{aligned}[\![\texttt{eval}(G_2, X_2, Y_2)]\!]\, s &= [\![\texttt{eval}(G_2[{X_2}^{Y_1}], X_2 \setminus {X_2}^{Y_1}, Y_2)]\!]\, s \\ &= [\![\texttt{eval}(G_2[{X_2}^{Y_1}], X'_2, Y_2)]\!]\, s \ ,\end{aligned}$$

since the wires that are not evaluated by X'_2 are defined inside G_2 as $G_2[{X_2}^{Y_1}]$. Note that

$$\begin{aligned}{X_2}^{Y_1} \circ Y_1 &= \{(v \leftarrow x) \in X_2 \mid x \in dom(Y_1)\} \circ \{(y \leftarrow w) \in Y_1\} \\ &= \{v \leftarrow w \mid (v \leftarrow x) \in X_2, (x \leftarrow w) \in Y_1\} \ ,\end{aligned}$$

and hence $G'_2 = G_2[{X_2}^{Y_1} \circ Y_1]$.
The composition:

$$\begin{aligned}[\![\texttt{eval}(G_2, X_2, Y_2)]\!]\, ([\![\texttt{eval}(G_1, X_1, Y_1)]\!]\, s) &= [\![\texttt{eval}(G_2[{X_2}^{Y_1}], X_2 \setminus {X_2}^{Y_1}, Y_2)]\!] ([\![\texttt{eval}(G_1, X_1, Y_1)]\!]\, s) \\ &= [\![\texttt{eval}(G_2[{X_2}^{Y_1} \circ Y_1], X_2 \setminus {X_2}^{Y_1}, Y_2)]\!]\, ([\![\texttt{eval}(G_1, X_1, Y_1 \setminus {Y_2}^{Y_1})]\!]\, s) \\ &= [\![\texttt{eval}(G'_2, X'_2, Y_2)]\!]\, ([\![\texttt{eval}(G_1, X_1, Y'_1)]\!]\, s) \ .\end{aligned}$$

By definition, we have
- $dom(Y'_1) \cap dom(Y_2) = \emptyset$;
- $dom(Y_1) \cap ran(X'_2) = \emptyset$.

By Lemma 1,

$$[\![\texttt{eval}(G'_2, X'_2, Y_2)]\!]([\![\texttt{eval}(G_1, X_1, Y'_1)]\!]\, s) = [\![\texttt{eval}(G_1 \cup G'_2, X_1 \cup X'_2, Y'_1 \cup Y_2)]\!]\, s.$$

– $T_C(y := f(e_1, \ldots, e_n)) = T_C(y_{i_1} := e_{i_1} \ ; \ \ldots \ ; \ y_{i_n} := e_{i_n}; \ y := f(y_1, \ldots, y_n))$, where $e_{i_1}, \ldots, e_{i_n}$ are non-variable/constant expressions, and $y_i = e_i$ for $i \notin \{i_1, \ldots, i_n\}$. Since we have already defined T_C on a sequential composition, by induction hypothesis, $T_C(y_{i_1} := e_{i_1} \ ; \ \ldots \ ; \ y_{i_n} := e_{i_n} = (G_1, X_1, Y_1)$ such that, for each $i \in [n]$, we have $Y_1(y_i) = [\![e_i]\!]$. We also have $T_C(y := f(y_1, \ldots, y_n)) = (G_2, X_2, Y_2)$ where $Y_2(y) = [\![f(y_1, \ldots, y_n)]\!]$, as $y_1, \ldots, y_n$ are now all either program variables or constants, so it has also been treated on the previous induction steps. Composing them together, get $T_C(y := f(e_1, \ldots, e_n)) = (G, X, Y)$ such that $Y(y) = [\![f]\!]([\![e_1]\!] s, \ldots, [\![e_n]\!]\, s) y = [\![f(e_1, \ldots, e_n)]\!]$.

– $T_C(\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2) = T_C(S)$ where
- $(G_i, X_i, Y_i) = T_C(S_i)$ for $i \in \{1, 2\}$,
- $Y'_i = \{z_{iy} \leftarrow w \mid (y \leftarrow w) \in Y_i\}$ for $i \in \{1, 2\}$: this renames the variables y of Y_i by introducing new variable names z_{iy},
- $S = \begin{cases} b_1 := b; \\ b_2 := (1 - b); \\ S'_1 = S_1[(y \leftarrow z_{1y}) \in Y'_1 \mid \exists w \ (y \leftarrow w) \in Y_1]; \\ S'_2 = S_2[(y \leftarrow z_{2y}) \in Y'_2 \mid \exists w \ (y \leftarrow w) \in Y_2]; \\ y := oc(b_1, r(1, y), b_2, r(2, y)) \ \forall y \in Y; \end{cases}$,
- $r(i, y) = \begin{cases} Y'_i(w) \text{ if } (y \leftarrow w) \in Y_i \\ y \text{ otherwise} \end{cases}$.

First, we claim that $[\![\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2]\!]\, s = [\![S]\!]\, s$. Let $y \in Var$.

- If $y \notin Y$, then $([\![S]\!]\,s)(y) = s(y)$ since S reassigns only b_1 and b_2 (which are not the part of s) and $y \in Y$.
  At the same time, $y \notin Y \implies y \notin (Y_1 \cup Y_2)$, and since S_i reassigns only variables of Y_i, $([\![\mathtt{if}\ b\ \mathtt{then}\ S_1\ \mathtt{else}\ S_2]\!]\,s)(y) = s(y)$.
- Let $y \in Y$. Let $[\![b]\!]\,s = 1$, Then $b_1 = 1$, and $b_2 = 0$, so $y = oc((b_1, r(1,y)), (b_2, r(2,y)) = r(1,y)$.
  - If $(y \leftarrow w) \in Y_1$, then $r(1,y) = Y_1'(w) = z_{1y}$. The only place where z_{1y} can be assigned is the statement S_1', and $([\![S_1[(y \leftarrow z_{1y}) \in Y_1' \mid (y \leftarrow w) \in Y_1]]\!]\,s)(z_{1y}) = ([\![S_1]\!]\,s)(y)$. Hence if $b = 1$, then $([\![S]\!]\,s)(y) = ([\![S_1]\!]\,s)(y)$.
  - Otherwise, $r(1,y) = y$, so the final statement is $y := y$. Since it is the first assignment of y in S, we have $([\![S]\!]\,s)(y) = s(y)$. At the same time, $([\![S_1]\!]\,s)(y) = s(y)$ since if $z_{1y} \notin dom(Y_1')$, then $y \notin dom(Y_1)$, and hence it is not reassigned in S_1.

  The proof is analogous for $[\![b]\!]\,s = 0$. We have got that:

$$([\![S]\!]\,s)(y) = \begin{cases} ([\![S_1]\!]\,s)(y) \text{ if } [\![b]\!]\,s = 0 \\ ([\![S_2]\!]\,s)(y) \text{ otherwise} \end{cases}$$
$$= ([\![\mathtt{if}\ b\ \mathtt{then}\ S_1\ \mathtt{else}\ S_2]\!]\,s)(y) \ .$$

By induction hypothesis, $[\![\mathtt{eval}(G_i, X_i, Y_i)]\!]\,s = [\![S_i]\!]\,s$ for $i \in \{1,2\}$. Hence

$$\begin{aligned} [\![S_i']\!] &= [\![S_i[(y \leftarrow z_{iy}) \in Y_i' \mid (y \leftarrow w) \in Y_i]]\!] \\ &= [\![\mathtt{eval}(G_i, X_i, Y_i)[(y \leftarrow z_{iy}) \in Y_i' \mid (y \leftarrow w) \in Y_i]]\!] \\ &= [\![\mathtt{eval}(G_i, X_i, Y_i')]\!] \ . \end{aligned}$$

Let $S_b = (b_1 := b\,;\, b_2 := (1-b))$, $S_C = (y := oc(b_1, r(1,y), b_2, r(2,y)))_{\forall y \in Y}$. We take $(G', X', Y') = T_C(S_b\,;\, S_1'\,; S_2'\,;\, S_C)$. Assuming by induction that T_C is defined correctly on sequential composition,
$[\![\mathtt{eval}(G', X', Y')]\!]\,s = [\![S_b\,;\, S_1'\,; S_2'\,;\, S_C]\!]\,s$. □

D The Weakest Precondition (WP)

We describe in more details the weakest precondition of computing a gate, that we first defined in Sec. 6.1. We propose an algorithm for computing the weakest preconditions of all the gates of a circuit.

D.1 Definitions

We give a more formal definition of the weakest precondition, that depends on the definition of semantics of circuit evaluation (Def. 2) and the transformation of expressions to circuits T_C (App. C).

Definition 4. *Let $\mathsf{G} = (G, X, Y)$ be a circuit. Let $s \in State$. For a wire $v \in V(\mathsf{G})$, we define a predicate $\mathsf{used}(v,s)$ as follows.*

1. $\mathsf{used}(v,s) = 1$ *for $v \in O(G)$ for any $s \in State$.*
2. *If* $\mathsf{used}(v,s) = 1$, $(v, op, \mathbf{a}) \in G$ *for $op \neq oc$, then* $\mathsf{used}(w,s) = 1$ *for all $w \in \mathbf{a}$.*
3. *If* $\mathsf{used}(v,s) = 1$, *and $(v, oc, [b_1, v_1, \ldots, b_n, v_n]) \in G$, then* $\mathsf{used}(b_i, s) = 1$ *for all $i \in \{1, \ldots, n\}$ (the choice conditions b_i), and* $\mathsf{used}(v_j, s) = 1$ *for $[\![G]\!]\,(s \circ X)\, b_j = 1$ (the choice that the oc gate makes).*

Definition 5. *In the circuit $\mathsf{G} = (G, X, Y)$, the weakest precondition ϕ_v^{G} of the wire $v \in V(\mathsf{G})$ is a boolean expression over $V(\mathsf{G})$ such that $[\![\phi_v^{\mathsf{G}}]\!]\,s = 1$ iff* $\mathsf{used}(v,s) = 1$, *where the semantics of a boolean expression is defined as $[\![\phi_v^{\mathsf{G}}]\!]\,s := [\![u := \phi_v^{\mathsf{G}}]\!]\,s\,u$.*

Algorithm 1: WP finds the weakest preconditions of all wires of G

```
Data: A circuit $\mathsf{G} = (G, X, Y)$
Result: A mapping $\phi : V(\mathsf{G}) \to BF(V(\mathsf{G}))$ that maps a wire to its weakest precondition
begin $\mathsf{WP}(G, X, Y)$
    $\phi \leftarrow \{\}$; $\psi \leftarrow \{\}$;
    foreach $v \in O(\mathsf{G})$ do
        $\mathsf{process}(v, 1)$
    return $\phi$
end
begin $\mathsf{process}(v, \phi_{in})$
    $\phi_v \leftarrow \phi(v)$;
    if $\phi_v \neq \bot$ then
        $\phi(v) \leftarrow (\phi_{in} \vee \phi_v)$
        return $\psi_v$
    else
        $\phi(v) \leftarrow \phi_{in}$
        switch $\mathsf{op}^G(v)$ do
            case $bop$                                  // $bop \in \{\wedge, \vee\}$
                $[a_1, a_2] \leftarrow \mathsf{args}^G(v)$
                $\psi^1_{out} \leftarrow \mathsf{process}(a_1, \phi(v))$
                $\psi^2_{out} \leftarrow \mathsf{process}(a_2, \phi(v))$
                return $\psi^1_{out}\ bop\ \psi^2_{out}$
            case $oc$
                $\mathbf{a} \leftarrow []$; $\mathbf{b} \leftarrow []$;
                foreach $(b, a) \in \mathsf{args}^G(v)$ do
                    $b' \leftarrow \mathsf{process}(b, \phi(v))$
                    $a' \leftarrow \mathsf{process}(a, b' \wedge \phi(v))$
                    $\mathbf{a} \leftarrow \mathbf{a} || (a')$
                    $\mathbf{b} \leftarrow \mathbf{b} || (b')$
                return $\sum_{i=1}^{|\mathbf{a}|=|\mathbf{b}|} b_i \cdot a_i$
            otherwise
                foreach $a \in \mathsf{args}^G(v)$ do
                    $\mathsf{process}(v, \phi(v))$
                return $v$
end
```

D.2 Computing the weakest precondition

We define a particular algorithm for computing the weakest preconditions. Let $BF(V)$ denote the set of all boolean formulas over a set of variables V. The algorithm constructs a mapping $\phi : V(\mathsf{G}) \to BF(V(\mathsf{G}))$, such that $\phi(v) = \phi_v^{\mathsf{G}}$. The construction of ϕ is given in Algorithm 1.

The function $\mathsf{process}(v, \phi_{in})$ takes a wire $v \in V(\mathsf{G})$ and some initially known overestimation of $\phi_{in} \in BF(V(\mathsf{G}))$ of v, which is in general the weakest precondition of some of the v's successors. This function returns a boolean formula that a wire $v \in V$ outputs, decomposed to boolean operations as far as possible. If the decomposition is impossible (for example, the gate operation is not a boolean operation, or it is an input wire), it just returns v. As a side effect, it updates the definition of function ϕ, and also the auxiliary function ψ that is used to remember the outcome of $\mathsf{process}(v, \phi_{in})$ in order to avoid processing the same wire multiple times.

We start from running $\mathsf{process}$ on each final output gate of G (lines 2-3). The first precondition that we propagate is 1, This means that there are no conditional constraints on v yet. Since the graph G is actually a statement of a larger program, it may happen that, instead of 1, there is a more precise condition that is coming from some public variable.

If v has already been visited ($\phi(v) \neq \bot$), it means that we have found another computational path that uses v. The condition of executing that path is ϕ_{in}, and we have already found another path with condition $\phi_v = \phi(v)$ before. Since both conditions are sufficient for forcing the computation of v, we update $\phi(v) \leftarrow \phi_v \vee \phi_{in}$, and return $\psi(v)$ that we have already computed before (lines 7-8).

If v has not been visited yet ($\phi(v) = \bot$), then, since ϕ_{in} is the weakest precondition of one of the computational paths that use v, we initialize $\phi(v) \leftarrow \phi_{in}$ on line 10. If $\mathsf{op}^G(v) = \wedge$, then we compute the outputs ψ^1_{out} and ψ^2_{out} of its arguments, and return $\psi^1_{out} \wedge \psi^2_{out}$ (lines 13-16). We do it analogously for $\vee$.

The precondition ϕ_{in} will be propagated to both arguments as $\phi(v)$. It is important that here $\phi(v)$ is passed not by value, but by reference, and in the case $\phi(v)$ gets updated after v will be reached via some other branch on further steps, this update will be propagated to all its predecessors.

In the case of *oc*, we process the arguments in pairs (b, a), where b is condition, and a is the choice. The conditions are just processed recursively as $b' \leftarrow \mathsf{process}(b, \phi(v))$. However for choices we have to extend $\phi(v)$ with the output of the corresponding condition as $a' \leftarrow \mathsf{process}(a, \phi(v) \wedge b')$, since b' adds an additional restriction on the precondition of a (lines 19-23). The output of the *oc* gate is defined by the line 24. The output condition of an *oc* gate is $\sum_{i=1}^{|\mathbf{a}|=|\mathbf{b}|} b_i \cdot a_i$, where b_i is a condition that has to be satisfied in order that the choice a_i would be output.

For any other gate, we just process the arguments recursively (line 27), and return v on line 28.

As the result, each wire $v \in V(\mathsf{G})$ will be assigned a boolean formula $\phi_v^{\mathsf{G}} = \phi(v)$ over $V(\mathsf{G})$. For a wire that should be computed in any case, we have $\phi^{\mathsf{G}}(v) = 1$. We will never fuse gates having such output wires.

The *cost* of the weakest precondition (denoted $cost(\phi_i^{\mathsf{G}})$) is just the total cost of all the $\vee$ and $\wedge$ operations used in it, without taking into account the complexity of computing its variables (their cost is estimated separately). In order to improve the optimization, we could try to rearrange $\vee$ and $\wedge$ operations in each ϕ_i^{G} to make the cost optimal. This is not in scope of this work.

The correctness of Algorithm 1 is stated in Proposition 1.

Proposition 1. *On input $\mathsf{G} \in \mathcal{G}$, Algorithm 1 returns a mapping ϕ such that, for all $v \in V(\mathsf{G})$, $\phi^{\mathsf{G}}(v)$ is the weakest precondition of v according to Def. 5.*

Proof: The proof of Algorithm 1 is split into two steps: the correctness of ψ definition and the correctness of ϕ^{G} definition. We prove it as two separate lemmas.

Lemma 2. *For each $v \in V(\mathsf{G})$ and for any ϕ_{in}, $\mathsf{process}(v, \phi_{in})$ returns an boolean expression ψ_{out} over $V(\mathsf{G})$ such that $[\![\psi_{out}]\!]\,[\![G]\!](s \circ X) = [\![G]\!](s \circ X)\,v$ (i.e ψ_{out} is an expression over $V(\mathsf{G})$ that computes the same value as v).*

Proof: The proof is based on induction, starting from the inputs $I(\mathsf{G})$.

- **Base:** for $v \in I(\mathsf{G})$, the algorithm returns $\psi_{out} = v$ (there is no gate operation), so $[\![\psi_{out}]\!]\,[\![G]\!](s \circ X) = [\![u := v]\!]\,[\![G]\!](s \circ X)\,u = [\![G]\!](s \circ X)[v]$.
- **Step:** If $\phi(v) \neq \bot$, then v has already been processed, and the algorithm returns $\psi(v)$, which is correct by induction hypothesis. Let us now assume that $\phi(v) = \bot$. Let $[\![\psi^i_{out}]\!]\,[\![G]\!](s \circ X) = [\![G]\!](s \circ X)[v_i]$ for all $v_i \in \mathsf{args}^G(v)$. There are now several cases for $\mathsf{op}^G(v)$.

- If $\mathsf{op}^G(v) = \wedge$, then

$$\begin{aligned}\llbracket G\rrbracket(s\circ X)[v] &= \llbracket\wedge\rrbracket(\llbracket G\rrbracket(s\circ X)[v_1], \llbracket G\rrbracket(s\circ X)[v_2]) \\ &= \llbracket\wedge\rrbracket(\llbracket\psi^1_{out}\rrbracket\,\llbracket G\rrbracket(s\circ X), \llbracket\psi^2_{out}\rrbracket\,\llbracket G\rrbracket(s\circ X)) \\ &= \llbracket\psi_{out}\rrbracket\,\llbracket G\rrbracket(s\circ X)\end{aligned}$$

  The proof is analogous for $\vee$.
- If $\mathsf{op}^G(v) = oc$, $\mathsf{arity}^G(v) = n$, then

$$\llbracket\psi_{out}\rrbracket\,\llbracket G\rrbracket(s\circ X) = \sum_{i=1}^{n}(\llbracket G\rrbracket(s\circ X)[b_i])\cdot(\llbracket G\rrbracket(s\circ X)[a_i]) \ ,$$

  where $(a_i, b_i) \in \mathsf{args}^G(v)$. By Definition 3, we have

$$\llbracket G\rrbracket(s\circ X)[v] = \sum_{i=1}^{n}(\llbracket G\rrbracket(s\circ X)[b_i])\cdot(\llbracket G\rrbracket(s\circ X)[a_i]).$$

- Otherwise, the algorithm returns $\psi_{out} = v$. This is similar to the base case. □

Lemma 3. *Algorithm 1 outputs ϕ such that, for all $v \in G$, $s \in State$, we have $\phi(v) = 1$ iff $\mathsf{used}(v,s) = 1$ according to Definition 4.*

Proof: The proof is based on induction, starting from the subset of outputs $O_f(\mathsf{G}) \subseteq O(\mathsf{G})$ that are *not used by any other gate as as an argument*. Since our circuits are finite acyclic graphs, at least one such output does exist.

– **Base:** for $v \in O_f(\mathsf{G})$, the function process takes the argument $\phi_{in} = 1$. Since each $v \in O_f(\mathsf{G})$ is not an input of any other gate, is visited only once. In this case, $\phi(v) = \perp$, and the algorithm assigns $\phi(v) = \phi_{in} = 1$. We have $\llbracket\phi(v)\rrbracket\, s = 1$ for all $s \in State$, and by the condition 1 of Definition 4 for all $v \in O(G)$ we have $\mathsf{used}(v,s) = 1$.

– **Step:** The definition of $\phi(v)$ may be constructed in several steps if v is visited several times. Only the final result must satisfy the lemma statement. Let $\llbracket\phi_{v_i}\rrbracket\, s = 1$ iff $\mathsf{used}(v_i,s) = 1$ for all v_i such that $v \in \mathsf{args}^G(v_i)$. By Lines 7 and 10, we finally have $\phi(v) = \phi^1_{in} \vee \cdots \vee \phi^n_{in}$, where ϕ^i_{in} has been passed as the second argument of $\mathsf{process}(v, \phi^i_{in})$ by its successor v_i. The exact value of ϕ^i_{in} depends on $\mathsf{op}^G(v_i)$.
  - If $\mathsf{op}^G(v_i) \neq oc$, then $\mathsf{process}(v, \phi^i_{in})$ may be called on the Lines 14, 15, 27. In either case, $\phi^i_{in} = \phi(v_i)$, since ϕ^i_{in} was passed as not a value, but as a reference, so it updates dynamically and is finally equal to $\phi(v_i)$. We get $\phi(v) = \phi(v_1) \vee \cdots \vee \phi(v_n)$. This is sufficient for the proof since $\llbracket\phi(v_i)\rrbracket\, s = 1$ iff $\mathsf{used}(v_i,s) = 1$, and having $\phi(v) = \phi(v_1) \vee \cdots \vee \phi(v_n)$ assigns $\llbracket\phi(v)\rrbracket, s = 1$ if for at least one i we have $s(v_i) = 1$. This satisfies the condition 2 of Definition 4, since if $\mathsf{used}(v_i,s) = 1$, then $\mathsf{used}(v,s) = 1$.
  - If $\mathsf{op}^G(v) = oc$, then $\mathsf{process}(v, \phi^i_{in})$ for an odd i is called at the Line 20 with $\phi^i_{in} = \phi(v_i)$. For an even i, this happens on the Line 21 with $\phi^i_{in} = b'_i \wedge \phi(v_i)$ where $\llbracket b'_i\rrbracket\,\llbracket G\rrbracket(s\circ X) = \llbracket G\rrbracket(s\circ X)[b_i]$ by Lemma 2. Let s be now fixed. Let j be such that $\llbracket G\rrbracket(s\circ X)[b_j] = 1$. For all even arguments, we have $\llbracket\phi^j_{in}\rrbracket\, s = \llbracket\phi(v_j)\rrbracket\, s$ and $\llbracket\phi^i_{in}\rrbracket s = 0$ for all $i \neq j$. This satisfies the condition 3 of Definition 4: if $\mathsf{used}(v_i,s) = 1$, then $\mathsf{used}(v,s) = 1$ if v is an odd argument b_i, or an even argument a_j such that $\llbracket G\rrbracket(s\circ X)[b_j] = 1$.

  So far we have proven that $\llbracket\phi^{\mathsf{G}}_v\rrbracket\, s = 1 \implies \mathsf{used}(v,s)$. Now we prove the other direction. Considering now both cases together, we see that, if $\llbracket\phi^{\mathsf{G}}_v\rrbracket\, s = 1$, then either at least one $\llbracket\phi^{\mathsf{G}}_{v_i}\rrbracket\, s = 1$ for some $\mathsf{op}^G(v_i) \neq oc$, or at least one $\llbracket\phi^{\mathsf{G}}_{v_i}\rrbracket\, s = 1$ for some $\mathsf{op}^G(v_i) = oc$ such that v is an odd input of v_i, or $\llbracket\phi^{\mathsf{G}}_{v_i}\rrbracket\, s = 1$ and $\llbracket G\rrbracket(s\circ X)[b_i] = 1$ such that v is an even input of v_i. There are no other options. Hence $\mathsf{used}(v,s) = 0$ unless one of the conditions 1-3 of Def. 4 is satisfied. □

Lemma 2 and Lemma 3 together immediately prove Proposition. 1.

E Proof of Correctness of Subcircuit Partitioning Algorithm

Let the sets A_n be defined as in Sec. 6.3. We need to show that, after the gates are merged into subcircuits, the resulting circuit still has the same impact on the state as the initial circuit.

First, we need to formally define the operation that a subcircuit S computes, as we did it for the gates. By construction, each A_k has only one output wire $w \in O(S)$, since except the root gate, we do not include any subgraphs whose outputs are used by some other subgraphs. Hence we may define the operation computed by S as

$$\mathsf{op}^{A_k}(S)(x_1,\dots,x_n) = [\![S]\!]\,(v_1 \leftarrow x_1,\dots,v_n \leftarrow x_n)\,w\ \ ,$$

for $v_i \in I(S)$, $n = |\mathsf{args}^{A_k}(S)|$.

Theorem 3. *Let $G \in \mathcal{G}$, $n \in \mathbb{N}$. Then the following statements hold:*

- *$I(\mathsf{G}) = I(A_n)$;*
- *for all $W : I(\mathsf{G}) \to Val$ we have $[\![G]\!](W) = [\![A_n]\!](W)$;*

Proof: For shortness of notation, let us define the predicate $\mathsf{correct}(S)$ for $S \in A_n$, s.t $\mathsf{correct}(S) = 1$ iff for all $W : I(S) \to Val$, $w \in O(S)$ we have

$$[\![S]\!]\,([\![A_n]\!](W)\mathsf{args}^{A_n}(S))\,w = [\![S]\!]\,([\![G]\!](W)\mathsf{args}^{G}(S))\,w\ \ .$$

In other words, $\mathsf{correct} = 1$ iff the output of S is the same, regardless of whether its inputs are evaluated in A_n or G. This is a bit weaker property than the one we are proving.

- **Base:** If $n == 0$, then each gate g is treated as a separate subcircuit, so $\mathsf{op}^{A_0}(\{g\}) = \mathsf{op}^G(g)$, $\mathsf{args}^{A_0}(\{g\}) = \mathsf{args}^G(g)$, $\mathsf{arity}^{A_0}(\{g\}) = \mathsf{arity}^G(g)$. The circuit has not changed, so $G = A_0$ and hence $I(\mathsf{G}) = I(A_0)$, and $\forall W : I(\mathsf{G}) \to Val, [\![G]\!](W) = [\![A_0]\!](W)$. Moreover, $\mathsf{correct}(S)$ holds for any subcircuit S of A_0.
- **Step:** Assume that we have $I(\mathsf{G}) = I(A_n)$, and $\forall W : I(\mathsf{G}) \to Val, [\![G]\!](W) = [\![A_n]\!](W)$ for a circuit A_n obtained for depth n. Now we are trying to unite each subcircuit S of A_n with $\mathsf{args}^{A_n}(S)$. Only those elements of $\mathsf{args}^G(S)$ that are used only by S are added, and each element is used on the current iteration only once. Hence, if the subcircuits of A_n are mutually exclusive, then so are the subcircuits of A'_{n+1}.
The subcircuits S that occur at most one time are decomposed back to the subcircuits S' of A_n, and by induction hypothesis $I(S') = I(T')$, $\forall W : I(S') \to Val, [\![S']\!](W) = [\![T']\!](W)$, where T' are the circuits of the gates of S' in G.
The subcircuit S that occurred at least 2 times is left in A_{n+1}, and the function $\mathsf{args}^{A_{n+1}}(S)$ is updated. Each such subcircuit is of the form $S = \{S_0, S_1, \dots, S_n\}$, where $\{S_1,\dots,S_n\} \subseteq \mathsf{args}^{A_n}(S_0)$, and $\forall i \in \{0,\dots,n\},\ \mathsf{correct}(S_i)$ hold by induction hypothesis. For all $W : I(S_i) \to Val$, we have

$$[\![S_i]\!]([\![A_{n+1}]\!](W)\mathsf{args}^{A_n}(S_i))\,w_i = [\![S_i]\!]([\![G]\!](W)\mathsf{args}^{A_n}(S_i))\,w_i$$

for $w_i \in O(S_i)$, and hence the subcircuits $\{S_1,\dots,S_n\}$ provide to S_0 the same inputs it would get in G. Therefore S_0 also outputs the same value it would output in G, so for all $W : I(S) \to Val$ we have $[\![S]\!]([\![A_{n+1}]\!](W)\mathsf{args}^{A_{n+1}}(S)) = [\![S]\!]([\![G]\!](W)\mathsf{args}^G(S))$, and $\mathsf{correct}(S)$ holds.
All the subcircuits of A_n have been included into A_{n+1}, either in their initial form, or united together with some other circuits. We have $\bigcup_{S \in A_{n+1}} = A_n$, and hence $I(A_n) = I(A_{n+1})$, and $\forall S \in A_n,\ \mathsf{correct}(S)$ implies $\forall W : I(A_n) \to Val, [\![A_n]\!](W) = [\![A_{n+1}]\!](W)$. By transitivity, $I(\mathsf{G}) = I(A_{n+1})$, and $\forall W : I(\mathsf{G}) \to Val, [\![G]\!](W) = [\![A_{n+1}]\!](W)$. □

F Greedy Algorithms

We give particular algorithms for fusing the gates in a greedy manner, using the strategies mentioned in Sec. 6.4. Just for this section, we will extend the predicates $\mathsf{pred}^{\mathsf{G}}$ and $\mathsf{cpred}^{\mathsf{G}}$ to their transitive closures, i.e define additionally $\mathsf{pred}^{\mathsf{G}}(i,j) = \exists k :\ \mathsf{pred}^{\mathsf{G}}(i,k) \wedge \mathsf{pred}^{\mathsf{G}}(k,j)$ and $\mathsf{cpred}^{\mathsf{G}}(i,j) = \exists k :\ \mathsf{cpred}^{\mathsf{G}}(i,k) \wedge \mathsf{cpred}^{\mathsf{G}}(k,j)$. Moreover, we also define $\mathsf{cpred}^{\mathsf{G}}(i,j) = \exists k :\ \mathsf{cpred}^{\mathsf{G}}(i,k) \wedge \mathsf{pred}^{\mathsf{G}}(k,j)$, since all the predecessors j of a conditional predecessor k of i will also become predecessors of i if i gets fused into some clique. We note that making $\mathsf{cpred}^{\mathsf{G}}$ transitive is an overestimation,

Algorithm 2: Greed fuses mutually exclusive gates of G into cliques

```
Data: A set of gates $G$
Result: A set of cliques $Cs$ of gates $G$
1  $Gs \leftarrow \{(cost(F), \{g \mid g \in G, \mathsf{op}^G(g) = F\}) \mid F$ is a gate op$\}$;
2  $Gs \leftarrow \mathsf{sort}(Gs, 0, 1)$;
3  $Cs \leftarrow \emptyset$;
4  foreach $G \in Gs$ do
5      $Cs \leftarrow Cs \cup \{\mathsf{FuseX}(G, Cs)\}$;
6  return Cs;
```

since if i is fused, then j is not necessarily a predecessor of i if k is not fused. This reduces the number of allowed fusings in the circuit, but nevertheless does not introduce any incorrect solutions.

The outline of all our strategies starts from the function Greed given in Alg. 2. First of all, the set of gates G is partitioned into subsets $Gs = \{G_1, \ldots, G_n\}$ (line 1), where each G_i corresponds to a gate operation. The sets G_i are sorted according to the cost of their gates (the operation uniquely determines the cost, so all the gates of G_i have the same cost). The function $\mathsf{sort}(\mathbf{x}, i, j)$ on line 2 can be any algorithm that sorts a list of tuples $\mathbf{x}$ according to their i-th elements, leaving behind the j-th component of each tuple.

After that, the algorithm starts fusing the gates into cliques, starting from the most expensive gates, adding a clique only if it is not in contradiction with already formed cliques. The function call $\mathsf{FuseX}(G, Cs)$ on line 5 returns a partitioning of gates G to cliques, choosing a particular strategy X, which can be any of Alg. 4, Alg. 5, and Alg. 6. It takes into account the set of already formed cliques Cs to avoid contradictions.

The function $\mathsf{goodClique}(C, Cs)$ (Alg. 3) checks whether a clique C is not in contradiction with already formed cliques Cs, and it also assigns a *level* to each good clique – the round on which the gate should be evaluated. In this way, if some predecessor of a gate is not computed on a strictly earlier round than the gate itself, then we have a cycle in the circuit, so something must be wrong with the formed cliques. For this, the maximal level of all the predecessors of C and the minimal level of all the successors of C (the sets $PredLevels$ and $SuccLevels$ computed on lines 3-4), checking if each gate of C can be assigned a level strictly between these two (lines 5-7). For conditional predecessors $\mathsf{cpred}^{\mathsf{G}}$, we additionally check whether the size of the successor clique is more than 1, i.e whether the conditional predecessor actually becomes a predecessor after fusing. The number 10000 on line 6 is just an upper bound on the number of levels, and it could be as well assigned to 1 since we may treat levels as rational numbers.

The particular strategies of extracting a clique are given in Alg. 4, Alg. 5, and Alg. 6.

Largest Cliques First. In Alg. 4, we are trying to fuse into one clique as many gates as possible before proceeding with the other cliques. The task of finding one maximum clique is NP-hard, and so we just generate some bounded amount of maximal cliques, taking the largest of them. Extracting one clique is done in the function largestClique. Fixing some gate as a starting point, we sequentially try to add each other gate, checking whether we still have a clique (line 4). Having done it for each gate as a starting point, the obtained cliques are sorted by size on line 5. The loop on line 6 takes the largest clique that is valid w.r.t already existing cliques. After extracting a clique on line 2 of function Fuse1, the function largestClique is applied again to the remaining gates, doing a recursive call to Fuse1.

Pairwise Merging. In Alg. 5, we first try to fuse the gates pairwise, and only after the pairs are formed, we proceed fusing the obtained cliques in turn pairwise, until the number of cliques cannot be decreased anymore. The function matching just takes the first valid matching that it succeeds to construct.

Pairwise Merging with Maximum Matching. Alg. 6 is very similar to Alg. 6, and the only difference is that it finds the *maximum* matching on each step. The function $\mathsf{someMatching}(Gs)$ generates non-deterministically all possible matchings of Gs, including the invalid cliques (in contrast, matching of Alg. 5 takes the first valid solution it finds). Alg. 6 then takes the largest of them that satisfies valid clique definition.

We need to show that the proposed strategies are indeed terminating, i.e if we already have fixed a clique greedily, it will not prevent the other gates from being taken at all. Intuitively, whatever cliques we have fixed, as far as they do not contradict each other, all the other gates may be at least added as singleton cliques without causing any problems. We state it in the following lemma.

Algorithm 3: goodClique checks if the clique is valid

```
Data: A clique $C$ whose correctness we check, and the set of already existing cliques $Cs$
Result: A bit denoting whether $C$ is valid w.r.t $Cs$
begin goodClique($C, Cs$)
1   foreach $i, j \in C$ do
2       if not $\mathsf{fusable}^{\mathsf{G}}(i, j)$ then
            return false
3   $PredLevels \leftarrow \{\mathsf{level}(k) \mid i \in C, \mathsf{pred}^{\mathsf{G}}(i,k) \vee \mathsf{cpred}^{\mathsf{G}}(i,k) \wedge |C| > 1\}$;
4   $SuccLevels \leftarrow \{\mathsf{level}(k) \mid i \in C, \mathsf{pred}^{\mathsf{G}}(k,i) \vee \mathsf{cpred}^{\mathsf{G}}(k,i) \wedge k \in C', C' \in Cs, |C'| > 1\}$;
5   $n_1 \leftarrow \max(\{0\} \cup PredLevels)$;
6   $n_2 \leftarrow \min(\{10000\} \cup SuccLevels)$;
7   if $n_2 < n_1$ then
        return false;
8   foreach $i \in C$ do
        $\mathsf{level}(i) \leftarrow (n_1 + n_2)/2$;
9   return true;
end
```

Algorithm 4: Fuse1 partitions the gates into cliques

```
Data: A set of gates $G$ of the same operation type
Result: Partitioning $Cs$ of cliques of $G$
begin Fuse1($G, Cs$)
1   if $G = \emptyset$ then
        return $\emptyset$;
2   $C \leftarrow \mathsf{largestClique}(G, Cs)$;
3   return $\mathsf{Fuse1}(G \setminus C, Cs \cup \{C\})$;
end
begin largestClique($G, Cs$)
4   $Gss \leftarrow \{(|Hs|, Hs) \mid i \in G, Hs \leftarrow \{i\} \cup \{j \mid j \in G, \forall k \in Hs :\ \mathsf{fusable}^{(}k, j)\}\}$;
5   $Gss \leftarrow \mathsf{sort}(Gss, 0, 1)$;
6   repeat
7       $Gs \in Gss$;
    until $\forall G \in Gs :\ \mathsf{goodClique}(G, Cs \cup Gs)$ ;
8   return $Gs$;
end
```

Lemma 4. *The function* Greed *terminates for any set of gates G with properly defined predicates* $\mathsf{fusable}^{\mathsf{G}}$, $\mathsf{pred}^{\mathsf{G}}$, *and* $\mathsf{cpred}^{\mathsf{G}}$, *producing a partitioning Cs of gates G that satisfy* $\mathsf{goodClique}(C, Cs) = \mathsf{true}$ *for any $C \in Cs$.*

Proof: The loops of Alg. 5 and Alg. 6 that wait until the set of gates Gs does not decrease anymore (both on lines 2-6) will definitely terminate since the size of a finite set cannot decrease infinitely. The other source of possible non-terminations are the loops that look for the solution that satisfies goodClique. We want to show that, regardless of the cliques that have already been fixed, it is always possible to add all the remaining gates at least as singleton cliques.

Let Cs be the set of cliques collected so far. Each strategy fixes a clique only if it has passed the goodClique test at some point. Assume that $\mathsf{goodClique}(C, Cs) = \mathsf{true}$ for all $C \in Cs$. Now we want to add a clique $\{g\}$ to Cs, where $g \in G$ is an arbitrary gate. We need to show that $\mathsf{goodClique}(C, Cs \cup \{g\}) = \mathsf{true}$ for all $C \in Cs \cup \{g\}$.

1. First, we show that $\mathsf{goodClique}(\{g\}, Cs \cup \{g\}) = \mathsf{true}$ holds. Suppose by contrary that it is impossible. This may happen in the following cases:
   (a) For some $g \in \{g\}$, $\mathsf{fusable}^{\mathsf{G}}(g, g) = \mathsf{false}$. By definition, is should be $\mathsf{fusable}^{\mathsf{G}}(g, g) = \mathsf{true}$.

Algorithm 5: Fuse2 partitions the gates into cliques

```
   Data: A set of gates $G$ of the same operation type
   Data: A set of already existing cliques $Cs$
   Result: Partitioning $Gs$ of cliques of $G$
   begin Fuse2($G, Cs$)
1      $Gs \leftarrow \{\{g\} \mid g \in G\}$;
2      repeat
3          $n \leftarrow |Gs|$;
4          $Gs \leftarrow \mathsf{matching}(Gs, \emptyset, Cs)$;
       until $|Gs| \geq n$ ;
5      return $Gs$;
   end
   begin matching($Gs, Hs, Cs$)
6      if $Gs = \emptyset$ then
           return $Hs$;
7      repeat
8          $G_1 \in Gs$;
9          $G_2 \in Gs$;
           $C \leftarrow G_1 \cup G_2$;
       until goodClique($C, Cs \cup Hs$) ;
10     return matching($Gs \setminus \{G_1\} \setminus \{G_2\}, Hs \cup \{C\}, Cs$);
   end
```

Algorithm 6: Fuse3 partitions the gates into cliques

```
   Data: A set of gates $G$ of the same operation
   Data: A set of already existing cliques $Cs$
   Result: Partitioning $Gs$ of cliques of $G$
   begin Fuse3($G, Cs$)
1      $Gs \leftarrow \{\{g\} \mid g \in G\}$;
2      repeat
3          $n \leftarrow |Gs|$;
4          $Gs \leftarrow \mathsf{maxMatching}(Gs, Cs)$;
       until $|Gs| \geq n$ ;
5      return $Gs$;
   end
   begin maxMatching($Gs, Cs$)
6      $Hss \leftarrow \{(|Hs|, Hs) \mid Hs \leftarrow \mathsf{someMatching}(Gs)\}$;
7      $Hss \leftarrow \mathsf{sort}(Hss, 0, 1)$;
8      repeat
9          $Hs \in Hss$;
       until $\forall H \in Hs :$ goodClique($H, Cs \cup Hs$) ;
10     return $Hs$;
   end
```

(b) It happens that $n_1 \geq n_2$ for $n_1 \leftarrow \mathsf{max}(\{0\} \cup \mathit{PredLevels})$, $n_2 \leftarrow \mathsf{min}(\{10000\} \cup \mathit{SuccLevels})$, where $\mathit{PredLevels} \leftarrow \{\mathsf{level}(k) \mid g \in C, \mathsf{pred}^{\mathsf{G}}(g,k) \vee \mathsf{cpred}^{\mathsf{G}}(g,k) \wedge |C| > 1\}$, and $\mathit{SuccLevels} \leftarrow \{\mathsf{level}(k) \mid g \in C, \mathsf{pred}^{\mathsf{G}}(k,g) \vee \mathsf{cpred}^{\mathsf{G}}(k,g) \wedge k \in C', C' \in Cs, |C'| > 1\}$. This means that there are some gates k, j belonging to cliques C_k and C_j such that $\mathsf{pred}^{\mathsf{G}}(g,k)$ (for $\mathsf{cpred}^{\mathsf{G}}(g,k)$ case, $|\{g\}| > 1$ never holds) and $\mathsf{pred}^{\mathsf{G}}(j,g)$ (or $\mathsf{cpred}^{\mathsf{G}}(j,g)$ and $|C_j| > 1$), and $\mathsf{level}(j) \leq \mathsf{level}(k)$. However, by transitivity of $\mathsf{pred}^{\mathsf{G}}$ and $\mathsf{cpred}^{\mathsf{G}}$, the statements $\mathsf{pred}^{\mathsf{G}}(j,k)$ (or $\mathsf{cpred}^{\mathsf{G}}(j,k)$ and $|C_j| > 1$) are also true, which contradicts the fact that $\mathsf{goodClique}(C_j, Cs) = \mathsf{true}$ and $\mathsf{goodClique}(C_k, Cs) = \mathsf{true}$.

Since $\mathsf{goodClique}(\{g\}, Cs \cup \{g\}) = \mathsf{true}$, we assign $\mathsf{level}(g) = n_g$ for some n_g as a side-effect.

2. After having assigned $\mathsf{level}(g) = n_g$, we need to prove that it has not broken the correctness of any old cliques, i.e $\mathsf{goodClique}(C, Cs \cup \{g\}) = \mathsf{true}$ holds for all $C \in Cs$, where Cs is the set of old cliques. Since $\mathsf{goodClique}(C, Cs) = \mathsf{true}$ holds due to induction hypothesis (adding a gate does not modify any predicates concerning the cliques that are already fixed), it remains to prove that we have $\mathsf{goodClique}(C, \{g\}) = \mathsf{true}$.
Let $C \in Cs$. Let n_1 and n_2 be the sizes of old sets *Predlevels* and *SuccLevels* before adding $\{g\}$. After adding $\{g\}$, there may be now more values that may get into these sets. Without loss of generality, let g be some successor of C. The minimal successor level is now $n_2' = \min(n_g, n_2)$. Since we have already shown that $\mathsf{goodClique}(\{g\}, Cs \cup \{g\}) = \mathsf{true}$ holds, we have assigned $\mathsf{level}(g) = n_g > \mathsf{level}(k)$ for all $k \in Cs$, so $n_g > (n_1 + n_2)/2$, and we have $\mathsf{level}(k) = n_1 < (n_1 + \min(n_g, n_2))/2 < \min(n_g, n_2) = n_2'$, so $n_1' < n_2$, and $\mathsf{goodClique}(C, Cs \cup \{g\}) = \mathsf{true}$. □

In App. H.3, we prove that the obtained cliques provide a valid transformation.

G Integer Linear Programming

In this section, we prove the correctness of the definitions of the building block constraints of Sec. 6.5. We also prove which variables are binary.

G.1 Proofs of some Auxiliary Lemmas

Lemma 5. *If $x \in \{0,1\}$, $y \leq C \in \mathbb{R}$, then*
$\mathcal{P}(C,x,y,z) = \mathsf{true} \iff z = x \cdot y$.

Proof: The correctness and completeness of these constraints can be easily verified by case distinction on x for any $y \leq C$.

1. Substitute $x = 0$ into the constraints:
   - $y - z \leq C$,
   - $-y + z \leq C$,
   - $-z \geq 0$.

   The last constraint uniquely defines $z = 0$. The first two constraints are true since $y \leq C$.
2. Substitute $x = 1$ into the constraints:
   - $y - z \leq 0$,
   - $-y + z \leq 0$,
   - $C - z \geq 0$.

   The first two constraints uniquely define $z = y$. The last constraint is true since $z = y \leq C$. □

Lemma 6. *If $x \in \{0,1\}$ for all $x \in \mathcal{X}$, $0 \leq y \leq C \in \mathbb{R}$, then*
$\mathcal{F}(A,C,\mathcal{X},y) = \mathsf{true}$ *iff* $y = 1 \iff \sum_{x \in \mathcal{X}} x \geq A$.

Proof: $\Longrightarrow$ Let the constraints be satisfied. By Lemma. 5, we have $z_x = x \cdot y$ for all $x \in \mathcal{X}$. Substituting z_x into the last two constraints, we get:

- $A \cdot y - y \cdot \sum_{x \in \mathcal{X}} x \leq 0$,
- $\sum_{x \in \mathcal{X}} x - y \cdot \sum_{x \in \mathcal{X}} x + (A-1)y \leq (A-1)$.

We can rewrite these constraints as

- $y(A-\sum_{x\in\mathcal{X}}x)\leq 0$,
- $\sum_{x\in\mathcal{X}}x(1-y)\leq(A-1)(1-y)$.

We get that $\sum_{x\in\mathcal{X}}x\geq A$ unless $y=0$, and $\sum_{x\in\mathcal{X}}x\leq(A-1)$ unless $y=1$. Hence the constraints are satisfiable only if $y\in\{0,1\}$. If $y=1$, then $\sum_{x\in\mathcal{X}}x\geq A$, and if $y=0$, then $\sum_{x\in\mathcal{X}}x\leq(A-1)$.

$\Longleftarrow$ Let $y=1\iff\sum_{x\in\mathcal{X}}x\geq A$. In order to satisfy the constraints $\mathcal{P}(C,y,x,z_x)$, by Lemma. 5 we take $z_x=x\cdot y$. We show by case distinction that the remaining two constraints are satisfied for both $y=0$ and $y=1$.

1. Let $y=0$.
   - $A\cdot 0-0\cdot\sum_{x\in\mathcal{X}}x\leq 0$,
   - $\sum_{x\in\mathcal{X}}x-0\cdot\sum_{x\in\mathcal{X}}x+(A-1)\cdot 0\leq(A-1)$.

   The first constraint is always true, and the second one is satisfied if $\sum_{x\in\mathcal{X}}x\leq(A-1)$, which is equivalent to $\sum_{x\in\mathcal{X}}x<A$ since $x\in\{0,1\}$.
2. Let $y=1$.
   - $A-\sum_{x\in\mathcal{X}}x\leq 0$,
   - $\sum_{x\in\mathcal{X}}x-\sum_{x\in\mathcal{X}}x+(A-1)\leq(A-1)$.

   The second constraint is always true, and the first one is satisfied if $\sum_{x\in\mathcal{X}}x\geq A$. □

Lemma 7. *If $z\in\{0,1\}$, $0\leq x,y\leq C\in\mathbb{R}$, then*
$\mathcal{G}(C,A,y,x,z)=\mathsf{true}$ *iff* $z=1\implies(x-y)\geq A$.

Proof: The correctness and completeness of the constraint $(C+A)\cdot z+(y-x)\leq C$ can be easily verified by case distinction on z for any $0\leq x,y\leq C$.

1. Substitute $z=0$: get $y-x\leq C$, which is always true for any $0\leq x,y\leq C$.
2. Substitute $z=1$: get $(C+A)+y-x\leq C$, which is equivalent to $A+y-x\leq 0$, or $x-y\geq A$. □

Lemma 8. *For $(A,\mathbf{b},\mathbf{c},\mathcal{I})=T_{ILP}^{\rightarrow}(G,X,Y)$, the following variables of any feasible solution of $(A,\mathbf{b},\mathbf{c},\mathcal{I})$ are binary:*

g_i^j	*for $j,i\in G$;*
b_j c_j, u_j,	*for $j\in G$;*
fx_ℓ^{jk}	*for $j\in G$, $k\in V(\mathsf{G})$, $\ell\in[\mathsf{arity}^G(j)]$;*
$e_{i\ell}^{jk}$	*for $j,i\in G$, $k\in V(\mathsf{G})$, $\ell\in[\mathsf{arity}^G(j)]$;*
fg^{ji}, fg_ℓ^{ji}	*for $j,i\in G$, $\ell\in[\mathsf{arity}^G(j)]$;*
s_ℓ^j	*for $j\in G$ $\ell\in[\mathsf{arity}^G(j)]$.*

Proof: By Lemma 5, if $x\in\{0,1\}$ and $y\leq C$, then $\mathcal{P}(C,x,y,z)$ ensures $z\in\{0,1\}$. By Lemma 6, if $\forall x\in\mathcal{X}:\ x\in\{0,1\}$, then $\mathcal{F}(A,\mathcal{X},y)$ ensures $y\in\{0,1\}$. We use these properties to propagate binariness.

We will make the proof for all types of variables one by one.

- The condition $g_i^j\in\{0,1\}$ is stated explicitly in the MIP description (the set $\mathcal{I}$).
- By constraints (6f), $c_i\in\{0,1\}$ since $g_i^j\in\{0,1\}$, and $d_j=(1-g_j^j)\in\{0,1\}$.
- By constraints (7a), $fx_\ell^{jk}\in\{0,1\}$ since $g_i^j\in\{0,1\}$.
- By constraints (7b), $e_{i\ell}^{jk}\in\{0,1\}$ since $g_i^j\in\{0,1\}$ and $fx_\ell^{jk}\in\{0,1\}$.
- By constraints (7c), for $k\in I(\mathsf{G})$, $fg_\ell^{ji}\in\{0,1\}$ since $e_{i\ell}^{jk}\in\{0,1\}$.
- By constraints (7d), for $k\notin I(\mathsf{G})$, $fg_\ell^{ji}=fx_\ell^{jk}$ and hence $\in\{0,1\}$
- By constraints (7e), $s_\ell^j\in\{0,1\}$ since $fg_\ell^{jk}\in\{0,1\}$.
- By constraint (8a), $fg^{jk}\in\{0,1\}$ since $fg_\ell^{jk}\in\{0,1\}$.
- By constraints (8b), $u^j\in\{0,1\}$ since $fg^{jk}\in\{0,1\}$.
- By constraints (9a), $t^j\in\{0,1\}$ since $s_\ell^j\in\{0,1\}$. Hence, by constraints (9b-9c), $t_i^j\in\{0,1\}$. Note that, for a fixed i, exactly one $g_i^j=1$ due to constraints (2). By definition of $\mathcal{P}$, we have $t_i^j=g_i^j\cdot t^j$, and hence at most one $t_i^j=1$. By constraints (9d), $b_i=\sum_{j\in G}t_i^j$, and so $b_i\in\{0,1\}$. □

G.2 Proof of feasibility of the integer programming task (Theorem 1)

Let $(A, \mathbf{b}, \mathbf{c}, \mathcal{I})$ be the mixed integer linear programming task. It has a solution iff the system $A\mathbf{x} \leq \mathbf{b}$ has at least one solution, assuming that $\forall i \in \mathcal{I}: \ x_i \in \{0,1\}$. We show that any solution in which $g_j^j = 1$ for all $j \in G$ and $g_i^j = 0$ for all $j, i \in G$, $i \neq j$, is feasible. Intuitively, this means that it is always possible not to fuse any gates, leaving the circuit as it is.

By Lemma 5 and Lemma 6 the constraints $\mathcal{P}(C,x,y,z)$ and $\mathcal{F}(A,\mathcal{X},z)$ are always satisfied if z is a new variable that has not been present in any other constraints before at this point.

Let $\forall j \in G: \ g_j^j = 1$, and $\forall j, i \in G, i \neq j: \ g_i^j = 0$. We show one by one, that all the constraints are satisfied.

1. $g_i^j + g_k^j \leq 1$ for $i,k \in G$, $\neg\mathsf{fusable}^{\mathsf{G}}(i,k)$.
   Since $\mathsf{fusable}^{\mathsf{G}}(i,i)$ holds for all i, here we have $i \neq k$, and never get the case $g_j^j + g_j^j \leq 1$. For all the other $g_i^j + g_k^j$ at least one term is 0.
2. $\sum_{j=1}^{|G|} g_i^j = 1$ for all $i \in G$.
   For any $i \in G$, the only j such that $g_i^j = 1$ is $j = i$.
3. $g_i^j = 0$ if $\mathsf{op}^G(i) \neq \mathsf{op}^G(j)$.
   We have $g_i^j = 1$ only if $i = j$, but then $\mathsf{op}^G(i) = \mathsf{op}^G(j)$.
4. $g_j^j - g_i^j \geq 0$ for all $i \in G$, $j \in G$.
   This is true since $g_j^j = 1$ and all g_i^j are binary by Lemma 8.
5. $g_j^j = 1$ for all j such that $cost(\mathsf{op}^G(j)) = 0$.
   All $g_j^j = 1$ anyway.
6. We show that a possible evaluation of ℓ_i is the topological ordering of gates in the initial circuit.
   (a) $\ell_i - \ell_k \geq 1$ for all $i,k \in G, \mathsf{pred}^{\mathsf{G}}(i,k)$;
   the constraint is satisfied by definition of $\mathsf{pred}^{\mathsf{G}}(i,k)$ and the fact that we use topological ordering which assigns a strictly smaller level to the gate predecessors.
   (b) The constraints $\mathcal{G}(|G|, 0, \ell_i, \ell_j, g_i^j)$ and $\mathcal{G}(|G|, 0, \ell_j, \ell_i, g_i^j)$ are satisfied since we only have $g_j^j = 1$, and $\ell_j = \ell_j$ is trivially satisfied.
   (c) $\ell_i \geq 0, \ell_i \leq |G|$.
   This holds by definition of topological order: it assigns a unique number to each gate.
   (d) $d_j = (1 - g_j^j)$;
   Satisfied since d_j is a newly introduced variable.
   (e) $\mathcal{F}(1, \{d_j\} \cup \{g_i^j \mid i \in G, i \neq j\}, c_j)$;
   Satisfied since c_j is a newly introduced variable. Namely, since $g_j^j = 1$, and $g_i^j = 0$ for $i \neq j$, we have $c_j = 0$ for all j.
   (f) $\mathcal{G}(|G|, 1, \ell_i, \ell_k, c_i)$ for $\mathsf{cpred}^{\mathsf{G}}(i,k)$.
   By Lemma 7, $c_i = 1$ implies $\ell_i - \ell_j \geq 0$. Since we have $c_i = 0$, the implication is trivially true.
7. The constraints (7) are satisfied due to introducing new variables fx_ℓ^{jk}, $e_{i\ell}^{jk}$, fg_ℓ^{ji}, s_ℓ^j, and sc_ℓ^j.
8. The constraints (8) are satisfied due to introducing new variables fg^{jk}, u^j, and uc^j.
9. The constraints (9) are satisfied due to introducing new variables t^j, t_i^j, and b_i. □

H Circuit Transformation

In this section, we give an algorithm that performs the transformation of the circuit according to the ILP solution. We prove its correctness, estimate the cost, and show that the same algorithm can be used with any of the greedy algorithms of App. F.

H.1 Transformation Correctness

The work of the transformation function $T^{\leftarrow}_{ILP}$ is given in Alg. 7. Let sol be the dictionary mapping ILP variables to their valuations. After evaluating sol by solving the ILP problem on line 1, the circuit is constructed sequentially, starting from an empty set of gates initialized on line 2.

The loop of line 3 iterates through all the cliques C_j. We assume that the cliques are sorted topologically, according to their level ℓ_j, so that the arguments of the clique are processed before the clique itself. The clique C_j is defined on line 4 as the set of all gates belonging to it. The arguments of C_j are processed one by one by the loop on line 5. On line 6 all the ℓ-th arguments of C_j and their weakest preconditions are collected into the set $\mathcal{B}^j_\ell$. Since we have computed the weakest preconditions ϕ^{G}_i in the initial graph G, some variables of ϕ^{G}_i may be unavailable in G' due to gate fusing. Hence each gate of ϕ^{G}_i is substituted with the corresponding clique representative that is left in G' after the fusing.

A fresh name v^j_ℓ is created for the new oc gate on line 7. Then, Alg. 8 is called on line 8, and it actually decides if an oc gate is needed. On line 1 of Alg. 8, all the values from which to choose are collected into the set $\mathcal{K}$. If $|\mathcal{K}| > 1$, then there are at least 2 choice candidates, and hence an oblivious choice needs to be introduced. The new node vb_k is needed to construct the choice of the argument k, which may be chosen by several different mutually exclusive choices. Hence the condition of choosing k is the sum of all the conditions b such that $(b,k) \in \mathcal{B}$ (here we are allowed to use addition instead of $\vee$ since the gates are mutually exclusive). T_C transforms the boolean expression to a set of gates. The oc gate itself is formally constructed on line 6 of Alg. 8. If $|\mathcal{K}| \leq 1$, then the new oc gate is not needed, and the the only element of $|\mathcal{K}|$ can be used straightforwardly (we substitute oc with id to make the presentation simpler).

After the inputs of C_j are handled, if $\mathsf{op}^{\mathsf{G}}(j) \neq oc$, the representative of C_j is included into G on line 15. If $\mathsf{op}^{\mathsf{G}}(j) = oc$, the algorithm collects the choices and their conditions directly from the arguments of j, and then calls Alg. 8 to check if it remains an oc gate, or becomes an id. This happens on lines 10-13 of Alg. 7.

Some variables of Y may point to improper output wires if the corresponding gates have been fused into cliques. These references are rearranged on line 16.

We now prove the correctness of $T^{\leftarrow}_{ILP}$. We prove that the semantics of the transformed circuit do not change.

Theorem 4. *Let $(G,X,Y) \in \mathcal{G}$. Let* solve *be an arbitrary integer linear programming solving algorithm. Let $(A,\mathbf{b},\mathbf{c},\mathcal{I}) = T^{\rightarrow}_{ILP}(G,X,Y)$. Then $[\![\mathtt{eval}(G,X,Y)]\!]\,s = [\![\mathtt{eval}(T^{\leftarrow}_{ILP}(\mathsf{solve}(A,\mathbf{b},\mathbf{c},\mathcal{I}),(G,X,Y)))]\!]\,s$ for any $s \in State$.*

Proof Let $\mathsf{G} = (G,X,Y)$ be the initial circuit. Let $\mathsf{G}' = (G',X',Y')$ be the transformed circuit. We show that $[\![\mathtt{eval}(G,X,Y)]\!]\,s = [\![\mathtt{eval}(G',X',Y')]\!]\,s$.

In order to make the proof easier, let us rewrite the expressions according to their definitions:

- $[\![\mathtt{eval}(G,X,Y)]\!]\,s = \mathtt{upd}(Y \circ [\![G]\!](s \circ X), s)$;
- $[\![\mathtt{eval}(G',X',Y')]\!]\,s = \mathtt{upd}(Y' \circ [\![G]\!](s \circ X'), s)$.

Since Alg. 7 defines $Y' \leftarrow Y$ on line 2, and only the range of Y' is modified on line 16, we have $dom(Y) = dom(Y') =: \mathcal{Y}$. It suffices to prove that

$$\forall y \in \mathcal{Y}:\ [\![G]\!](s \circ X)(Y(y)) = [\![G']\!](s \circ X')(Y'(y))\ \ .$$

For all j, we have defined $Y' = Y[i \leftarrow j \mid g^j_i = 1]$ on line 16. Since there is exactly one j such that $g^j_i = 1$ (by constraints (2) of Sec. 6.5), we should actually prove

$$\forall i,j \in G, i,j \in \mathcal{Y}:\ (g^j_i = 1) \implies [\![G]\!](s \circ X)(i) = [\![G']\!](s \circ X')(j)\ \ . \tag{3}$$

Since all the output wires of (G,X,Y) are evaluated in any case (by circuit definition), for each i such that $i \in \mathcal{Y}$ we may add an additional assumption $[\![\phi^{\mathsf{G}}_i]\!]\,[\![G]\!](s \circ X) = 1$, where ϕ^{G}_i is the weakest precondition of evaluating i. This will be useful during the proof by induction. We may now replace the assumption $i,j \in \mathcal{Y}$ with $[\![\phi^{\mathsf{G}}_i]\!]\,[\![G]\!](s \circ X) = 1$ in (3), proving a less general result

$$\forall i,j \in G:\ ([\![\phi^{\mathsf{G}}_i]\!]\,[\![G]\!](s \circ X) = 1 \wedge g^j_i = 1) \implies [\![G]\!](s \circ X)(i) = [\![G']\!](s \circ X')(j)\ \ . \tag{4}$$

We prove this statement by induction on the number of the first j topologically ordered cliques that have already been processed. More precisely, for the clique ordering we use the variables ℓ_i from the constraints (6).

First, we prove that the gate ordering defined by ℓ_i creates no cycles. Namely, we prove that if a gate g_j is computed on the level ℓ_j, then each its argument has been computed on the level ℓ_k for $\ell_k < \ell_j$.

Algorithm 7. $T_{ILP}^{\leftarrow}$ reconstructs the circuit G according to the variables g_i^j

```
Data: A circuit $\mathsf{G} = (G, X, Y) \in \mathcal{G}$
Data: An ILP $(A, \mathbf{b}, \mathbf{c})$
Result: A transformed circuit $\mathsf{G}' = (G', X', Y')$
1  $\mathsf{sol} \leftarrow \mathsf{solve}(A, \mathbf{b}, \mathbf{c})$;
2  $G' \leftarrow \emptyset, X' \leftarrow X, Y' \leftarrow Y$;
3  foreach $j \in G$, $\mathsf{sol}(g_j^j) = 1$ do
4      $C_j \leftarrow \{i \mid \mathsf{sol}(g_i^j) = 1\}$;
5      foreach $\ell \in [\mathsf{arity}^G(j)]$ do
6          $\mathcal{B}_\ell^j \leftarrow \{(\phi_i^{\mathsf{G}}[i' \leftarrow j' \mid i' \in C_{j'}], k) \mid k \in I(\mathsf{G}), i \in C_j, k = \mathsf{args}^G(i)[\ell]\}$
               $\cup\{(\phi_i^{\mathsf{G}}[i' \leftarrow j' \mid i' \in C_{j'}], k) \mid k \notin I(\mathsf{G}), i \in C_j, \exists u:\ u = \mathsf{args}^G(i)[\ell], u \in C_k\}$;
7          $v_\ell^j \leftarrow \mathsf{fresh}()$;
8          $G_\ell^j \leftarrow \mathsf{ocSubgraph}(v_\ell^j, \mathcal{B}_\ell^j)$;
9          $G' \leftarrow G' \cup G_\ell^j$;
10     if $\mathsf{op}^G(j) = oc$ then
11         $\mathcal{B}^j \leftarrow \{(\mathsf{args}^G(j)[\ell - 1][i' \leftarrow j' \mid i' \in C_{j'}], v_\ell^j) \mid \ell \in \mathsf{arity}^G(j), \ell \in 2\mathbb{N}\}$;
12         $G^j \leftarrow \mathsf{ocSubgraph}(j, \mathcal{B}^j)$;
13         $G' \leftarrow G' \cup G^j$;
14     else
15         $G' \leftarrow G' \cup \{(j, \mathsf{op}^G(j), [v_1^j, \ldots, v_{\mathsf{arity}^G(j)}^j])\}$;
16 $Y' \leftarrow Y'[i \leftarrow j \mid i \in C_j]$;
17 return $(G', X', Y')$;
```

Algorithm 8: ocSubgraph constructs either an oc gate, or an id gate

```
Data: $\mathcal{B}$ – a set of condition and choice pairs $(b, k)$
Data: $v$ – the name of the wire that outputs the choice result
Result: A set of gates computing the oc and its conditions
1  $\mathcal{K} \leftarrow \{k \mid \exists b:\ (b, k) \in \mathcal{B}\}$;
2  if $|\mathcal{K}| > 1$ then
3      foreach $k \in \mathcal{K}$ do
4          $vb_k \leftarrow \mathsf{fresh}()$;
5          $(G_k, X_k, Y_k) \leftarrow T_C(vb_k := \sum_{(b,k) \in \mathcal{B}} b)$;
6      return$\{G_k\}_{k \in \mathcal{K}} \cup \{(v, oc, [vb_k, k]_{k \in \mathcal{K}})\}$;
7  else
8      $\{w\} \leftarrow \mathcal{K}$;
9      return $\{(v, id, w)\}$;
```

Lemma 9. *The following claims hold.*

- *For all $j \in G$, $k \in \mathsf{args}^G(j)$: $\ell_k < \ell_j$.*
- *For all $j, i \in G$: if $g_i^j = 1$, then $\ell_i = \ell_j$.*
- *For all $i \in G$, $k \in \mathsf{args}^G(\phi_i^{\mathsf{G}})$: if $g_i^j = 1$ for some $j \neq i$, then $\ell_k < \ell_i$.*

Proof: Recall the constraints 6:

(a) $\ell_i - \ell_k \geq 1$ for all $i, k \in G, \mathsf{pred}^{\mathsf{G}}(i,k)$;
since we define $\mathsf{pred}^{\mathsf{G}}(i,k)$ for all $k \in \mathsf{args}^G(i)$, this constraint ensures that $\forall j \in G, k \in \mathsf{args}^G(j) : \ell_k < \ell_j$.

(b-d) $\mathcal{G}(|G|, 0, \ell_i, \ell_j, g_i^j)$, $\mathcal{G}(|G|, 0, \ell_j, \ell_i, g_i^j)$, and $0 \leq \ell_i \leq |G|$ for all $i, j \in G$;
since g_i^j are binary variables, by Lemma 7 this ensures that, if $g_i^j = 1$, then $\ell_i \leq \ell_j$ and $\ell_j \leq \ell_i$, so $\ell_i = \ell_j$.

(e-f) $d_j = (1 - g_j^j)$ and $\mathcal{F}(1, \{d_j\} \cup \{g_i^j \mid i \in G, i \neq j\}, c_j)$ for all $j \in G$;
By Lemma 6, the constraints ensure that $c_j \in \{0,1\}$, and that $c_j = 1$ iff either $g_i^j = 1$ for some $j \neq i$, or there is k s.t $g_j^k = 1$.

(g) $\mathcal{G}(|G|, 1, \ell_i, \ell_k, c_i)$ for all $i, k \in G$, $\mathsf{cpred}^{\mathsf{G}}(i,k)$;
By definition, $\mathsf{cpred}^{\mathsf{G}}(i,k) = 1$ iff $k \in \mathsf{args}^G(\phi_i^{\mathsf{G}})$. By Lemma 7, since c_i is a binary variable, $c_i = 1$ implies $\ell_i - \ell_k \geq 1$, which is $\ell_k < \ell_i$, and $c_i = 1$ is implied by $g_i^j = 1$, $i \neq j$ according to the previous constraints. □

Let G_j be the subcircuit of G consisting just of the gates belonging to the first j cliques ordered by ℓ_k (i.e $G_j = \{C_k \mid \ell_k \leq \ell_j\}$ where $C_k = \{i \mid g_i^k = 1\}$). Let G'_j be the subcircuit of G' obtained after processing the first j cliques by Alg. 7. In this way, if there are m cliques in total, then $G' = G'_m$, and $G = G_m$.

Base: $G_0 = \emptyset$, and the statement (4) is trivially true.

Step: Suppose that we are adding the clique C_j to the subcircuit G_{j-1}. By induction hypothesis, the statement (4) already holds for all $i, j \in G \setminus C_j$, so it suffices to prove that

$$\forall i \in C_j : \; ([\![\phi_i^{\mathsf{G}}]\!]\,[\![G]\!](s \circ X) = 1 \wedge g_i^j = 1) \implies [\![G_j]\!](s \circ X)(i) = [\![G'_j]\!](s \circ X')(j) \; . \tag{5}$$

Since by definition of C_j we have $\forall i \in C_j : \; g_i^j = 1$, we may simplify (5) and prove

$$\forall i \in C_j : \; [\![\phi_i^{\mathsf{G}}]\!]\,[\![G]\!](s \circ X) = 1 \implies [\![G_j]\!](s \circ X)(i) = [\![G'_j]\!](s \circ X')(j) \; . \tag{6}$$

Let $i \in C_j$. Let $t = \mathsf{arity}^G(j)$. Before the transformation, for all i s.t $g_i^j = 1$, due to the constraint (2), which states that a gate belongs to exactly one clique, there should have been exactly one gate $(i, \mathsf{op}^G(i), [k_1, \ldots, k_t])$ in C_j, and, due to constraints (3), which state that a gate has the same operation as the clique representative, $\mathsf{op}^G(i) = \mathsf{op}^G(j)$.

First, let $\mathsf{op}^G(j) \neq oc$. In this case, new the input wires of j are exactly the new variables v_ℓ^j. By definition,

$$[\![G'_j]\!](s \circ X')(j) = [\![\mathsf{op}^{G'}(j)]\!]([\![G'_j]\!](s \circ X')(v_1^j), \ldots, [\![G'_j]\!](s \circ X')(v_t^j)) \; ,$$

and

$$[\![G_j]\!](s \circ X)(i) = [\![\mathsf{op}^G(j)]\!]([\![G_j]\!](s \circ X)(k_1), \ldots, [\![G_j]\!](s \circ X)(k_t)) \; .$$

Since $\mathsf{op}^G(j) \neq oc$, we have $\mathsf{op}^G(j) = \mathsf{op}^{G'}(j)$, and it suffices to show that the arguments of $[\![\mathsf{op}^G(j)]\!]$ in both cases are the same, i.e

$$\forall \ell \in [t] : \; ([\![\phi_i^{\mathsf{G}}]\!] s = 1) \implies [\![G_j]\!](s \circ X)(k_\ell) = [\![G'_j]\!](s \circ X')(v_\ell^j) \; . \tag{7}$$

Let $\mathcal{K}_\ell^j$ denote the set $\mathcal{K}$ formed by Alg. 8 when called by Alg. 7 with the input $\mathcal{B}_\ell^j$. We will prove statement (7) for different cases of $|\mathcal{K}_\ell^j|$.

1. If $|\mathcal{K}_\ell^j| \leq 1$, then Alg. 8 creates a gate (v_ℓ^j, id, w_ℓ^j) for $\{w_\ell^j\} \leftarrow \mathcal{K}_\ell^j$. Since a non-empty clique C_j has at least one ℓ-th input (the one belonging to the gate j), we have $|\mathcal{K}_\ell^j| = 1$, and so such w_ℓ^j exists, and the definition of v_ℓ^j is correct. Since w_ℓ^j is the only ℓ-th input of C_j, and k_ℓ is the ℓ-th input of some gate of C_j by definition, we have $w_\ell^j = k_\ell$, and since id does not modify the value, we have $[\![G'_j]\!](s \circ X')(v_\ell^j) = [\![G'_j]\!](s \circ X')(w_\ell^j)$.

(a) If $w^j_\ell \in I(\mathsf{G})$, then trivially $w^j_\ell \in G'_i$ since Alg. 7 keeps all the input gates of the initial circuit on line 2.

(b) If $w^j_\ell \notin I(\mathsf{G})$, then there is $u \in C_i$ such that $u = \mathsf{args}^G(j)[\ell]$. We have $\mathsf{pred}^{\mathsf{G}}(j,u)$, and by Lemma 9, $\ell_u < \ell_j$. Hence $w^j_\ell \in G'_u \subseteq G'_j$.

As the result, the values w^j_ℓ chosen by Alg. 7 satisfy $w^j_\ell \in G'_j$. Since $w^j_\ell = k_\ell$, we have $[\![G_j]\!](s\circ X)(k_\ell) = [\![G'_j]\!](s\circ X')(w^j_\ell) = [\![G'_j]\!](s\circ X')(v^j_\ell)$.

2. If $|\mathcal{K}^j_\ell| > 1$, Alg. 8 defines a subcircuit $(G^{jk}_\ell, X^{jk}_\ell, Y^{jk}_\ell)$ computing $b^{jk}_\ell := \sum_{i\in\mathcal{I}^{jk}_\ell} \phi^{\mathsf{G}}_i[i' \leftarrow j' \mid g^{j'}_{i'} = 1)]$, where, by definition of $\mathcal{B}^j_\ell$,

$$\mathcal{I}^{jk}_\ell = \{i \mid k \in I(\mathsf{G}), k = \mathsf{args}^{\mathsf{G}}(i)[\ell]\} \\ \cup\{i \mid k \notin I(\mathsf{G}), k \in C_u, u = \mathsf{args}^{\mathsf{G}}(i)[\ell]\} \ .$$

The gate names of $(G^{jk}_\ell, X^{jk}_\ell, Y^{jk}_\ell)$ are fresh, so there are no name conflicts with the gate names of G'_{j-1}. Since all the gates are new and do not belong to G, we do not need to prove statement (6) for them.

The following lemma proves the correctness of computing the conditions of the newly introduced *oc* gates. By definition, we collect all the gates that use the k as the ℓ-th argument into $\mathcal{I}^{jk}_\ell$, and hence the condition for choosing $i \in \mathcal{I}^{jk}_\ell$ should be $\sum_{i\in\mathcal{I}^{jk}_\ell} \phi^{\mathsf{G}}_i$.

Lemma 10. *Let* $|\mathcal{K}^j_\ell| > 1$. *If* $[\![\phi^{\mathsf{G}}_k]\!]\,[\![G]\!](s\circ X) = 1$, *then*

$$[\![G'_j]\!](s\circ X')b^{jk}_\ell = [\![\sum_{i\in\mathcal{I}^{jk}_\ell} \phi^{\mathsf{G}}_i]\!]\,[\![G_j]\!](s\circ X) \ .$$

Proof: Since $b^{jk}_\ell = \sum_{i\in\mathcal{I}^{jk}_\ell} \phi^{\mathsf{G}}_i[i' \leftarrow j' \mid g^{j'}_{i'} = 1]$, it suffices to show that

$$[\![\phi^{\mathsf{G}}_i[i' \leftarrow j' \mid g^{j'}_{i'} = 1]]\!]\,[\![G'_j]\!](s\circ X') = [\![\phi^{\mathsf{G}}_i]\!]\,[\![G_j]\!](s\circ X) \ .$$

We can rewrite it as

$$[\![\phi^{\mathsf{G}}_i]\!]\,[\![G'_j]\!](s\circ X')[i' \leftarrow j' \mid g^{j'}_{i'} = 1] = [\![\phi^{\mathsf{G}}_i]\!]\,[\![G_j]\!](s\circ X) \ .$$

For all variables i' of ϕ^{G}_i, we have $\mathsf{cpred}^{\mathsf{G}}(i,i')$. By Lemma 9, $\ell_{i'} < \ell_i$, and by induction hypothesis, $[\![\phi^{\mathsf{G}}_{i'}]\!]\,[\![G]\!](s\circ X) = 1 \implies [\![G_j]\!](s\circ X)(i') = [\![G'_j]\!](s\circ X')(j')$.

In order to apply the hypothesis, we need $[\![\phi^{\mathsf{G}}_{i'}]\!]\,[\![G]\!](s\circ X) = 1$ to hold, but we have $[\![\phi^{\mathsf{G}}_i]\!]\,[\![G]\!](s\circ X) = 1$. Since each i' is involved in the computation of i and is its predecessor, we have

$$[\![\phi^{\mathsf{G}}_{\phi^{\mathsf{G}}_i}]\!]\,[\![G]\!](s\circ X) = 1 \implies [\![\phi^{\mathsf{G}}_{i'}]\!]\,[\![G]\!](s\circ X) = 1 \ .$$

By assumption, $[\![\phi^{\mathsf{G}}_i]\!]\,[\![G]\!](s\circ X) = 1$, and in order that ϕ^{G}_i could be evaluated, it should be $[\![\phi^{\mathsf{G}}_{\phi^{\mathsf{G}}_i}]\!]\,[\![G]\!](s\circ X) = 1$. We get an implication

$$[\![\phi^{\mathsf{G}}_i]\!]\,[\![G]\!](s\circ X) = 1 \implies [\![\phi^{\mathsf{G}}_{i'}]\!]\,[\![G]\!](s\circ X) = 1 \ .$$

We get $[\![G_j]\!](s\circ X)(i') = [\![G'_j]\!](s\circ X')(j')$ for all i' that define the value of ϕ^{G}_i. Hence

$$[\![\phi^{\mathsf{G}}_i[i' \leftarrow j' \mid g^{j'}_{i'} = 1]]\!]\,[\![G'_j]\!](s\circ X') = [\![\phi^{\mathsf{G}}_i]\!]\,[\![G_j]\!](s\circ X) \ .$$

□

Alg. 7 then defines a new gate $(v^j_\ell, oc, [b^{jk}_\ell, k]_{k\in\mathcal{K}^j_\ell})$ for a new variable v^j_ℓ that has not been used anywhere else before. We have

$$[\![G_j]\!](s\circ X)(v^j_\ell) = [\![oc]\!]([b^{jk}_\ell, k]_{k\in\mathcal{K}^j_\ell}) \ .$$

By constraints (1) and the definition of $\mathsf{fusable}^{\mathsf{G}}$, the weakest preconditions of the gates inside one clique are mutually exclusive, and hence for any s, at most one of $[\![G'_j]\!](s \circ X')(b^{jk}_\ell)$ is 1, so this is a valid instance of oc.

We need to prove the equality $[\![G'_j]\!](s \circ X')(v^j_\ell) = [\![G_j]\!](s \circ X)(k_\ell)$, where k_ℓ is the ℓ-th input of i, on the assumption $[\![\phi^{\mathsf{G}}_i]\!]\,[\![G_j]\!](s \circ X) = 1$. Since $k_\ell \in \mathcal{K}^j_\ell$, there is a pair $(b^{jk_\ell}_\ell, k_\ell)$ in the oblivious choice.

By Lemma 10 we have

$$[\![G'_j]\!](s \circ X')(b^{jk_\ell}_\ell) = [\![\sum_{i' \in \mathcal{I}^{jk_\ell}_\ell} \phi^{\mathsf{G}}_{i'}]\!]\,[\![G_j]\!](s \circ X) \enspace .$$

By definition of $\mathcal{I}^{jk_\ell}_\ell$, k_ℓ is a predecessor of all $i' \in \mathcal{I}^{jk_\ell}_\ell$, so we have $([\![\phi^{\mathsf{G}}_{i'}]\!]\,[\![G_j]\!](s \circ X) = 1) \implies ([\![\phi^{\mathsf{G}}_{k_\ell}]\!]\,[\![G_j]\!](s \circ X) = 1)$ for all $i' \in \mathcal{I}^{jk_\ell}_\ell$ and since $([\![\phi^{\mathsf{G}}_{k_\ell}]\!]\,[\![G_j]\!](s \circ X) = 1) \implies ([\![b^{jk_\ell}_\ell]\!]\,s = 1)$, we also have $([\![\phi^{\mathsf{G}}_{i'}]\!]\,[\![G_j]\!](s \circ X) = 1) \implies ([\![b^{jk_\ell}_\ell]\!]\,s = 1)$, in particular $([\![\phi^{\mathsf{G}}_{i}]\!]\,[\![G_j]\!](s \circ X) = 1) \implies ([\![b^{jk_\ell}_\ell]\!]\,s = 1)$.

Hence it suffices to prove $\forall \ell : ([\![b^{ji_\ell}_\ell]\!]\,s = 1) \implies [\![G'_j]\!](s \circ X')(v^j_\ell) = [\![G_j]\!](s \circ X)(k_\ell)$. If $[\![b^{jk_\ell}_\ell]\!]\,s = 1$, then $[\![G'_j]\!](s \circ X')(v^j_\ell) = [\![G'_j]\!](s \circ X')(k_\ell)$ by definition of oc. Since $k_\ell \in C_{i'}$ for some $i' < j$, we have $[\![G'_j]\!](s \circ X')(k_\ell) = [\![G'_{j-1}]\!](s \circ X')(k_\ell)$, and due to $([\![\phi^{\mathsf{G}}_i]\!]\,s = 1) \implies ([\![\phi^{\mathsf{G}}_{k_\ell}]\!]\,s = 1)$, by induction hypothesis equal to $[\![G_{j-1}]\!](s \circ X)(k_\ell) = [\![G_j]\!](s \circ X)(k_\ell)$.

If $\mathsf{op}^G(j) \neq oc$, then a gate $(j, \mathsf{op}^G(j), [v^j_1, \ldots, v^j_t])$ is added to the clique C_j. By the constraints (4) that define the clique representative, if the clique C_j is non-empty (there exists at least one $g^j_i = 1$), then definitely $j \in C_j$ ($g^j_j = 1$), and $\forall i \in G:\ g^i_j = 0$ due to constraints (2). Hence it is the only gate with the name j, so there are no name conflicts.

If $\mathsf{op}^G(j) = oc$, then it may happen that $\mathsf{op}^{G'}(j)$ changes, and moreover, the inputs v^j_ℓ may be rearranged by the call to Alg. 8. We need to prove $[\![G_j]\!](s \circ X)(i) = [\![G'_j]\!](s \circ X')(j)$ straightforwardly. The proof is analogous to the proof that $[\![G_j]\!](s \circ X)(k_\ell) = [\![G'_j]\!](s \circ X')(v^j_\ell)$, as we call the same Alg. 8 here, and even a bit simpler since we the conditions for the choices are the inputs of j, and not the weakest preconditions. □

H.2 The Cost of the Transformed Circuit

First, we will prove some relations between different variables that are defined by the constraints. Let $\mathcal{K}^j_\ell$ and $\mathcal{K}^j$ denote the sets $\mathcal{K}$ formed by Alg. 8 when called by Alg. 7 with the inputs $\mathcal{B}^j_\ell$ and $\mathcal{B}^j$ respectively.

Lemma 11. *For all $j \in G$, $\ell \in \mathsf{arity}^G(j)$, $k \in V(\mathsf{G})$, we have:*

- *$\mathcal{K}^j_\ell = \{k \mid fg^{jk}_\ell = 1\}$;*
- *$\mathcal{K}^j = \{k \mid fg^{jk} = 1\}$.*

Proof: The proof is based on the fact that all the variables are binary (proven in Lemma 8), and on the definition of constraints $\mathcal{P}$ and $\mathcal{F}$ for binary inputs (proven in Lemma 5 and Lemma 6).

1. By definition of $\mathcal{F}$, since $g^j_i \in \{0,1\}$, $fx^{jk}_\ell \in \{0,1\}$, and $fx^{jk}_\ell = 1$ iff at least one $g^j_i = 1$ s.t $k = \mathsf{args}^G(i)[\ell]$. Since $g^j_i = 1$ denotes $i \in C_j$, we get $fx^{jk}_\ell = 1$ iff $\exists i \in C_j :\ k = \mathsf{args}^G(i)[\ell]$.
2. By definition of $\mathcal{P}$, since fx^{jk}_ℓ and g^i_k are binary, $e^{jk}_{i\ell} \in \{0,1\}$, and $e^{jk}_{i\ell} = 1$ iff $fx^{jk}_\ell \cdot g^i_k$. Since $g^i_k = 1$ denotes $k \in C_i$, we get $e^{jk}_{i\ell} = 1$ iff $k \in C_i$ and $k \in \mathsf{args}^G(C_j)[\ell]$.
3. By definition of $\mathcal{F}$, for $i \in G$, since $e^{jk}_{i\ell}$ are binary, $fg^{jk}_\ell \in \{0,1\}$, and $fg^{ji}_\ell = 1$ iff there exists $k \in I(\mathsf{G})$ s.t $e^{jk}_{i\ell} = 1$, so there is some k that causes $k \in C_i$ and $k \in \mathsf{args}^G(j)[\ell]$. We get $fg^{jk}_\ell = 1$ iff $\exists i \in C_j, u \in C_k : u = \mathsf{args}^G(i)[\ell]$.
4. For $k \in I(\mathsf{G})$, define $fg^{jk}_\ell = fx^{jk}_\ell$. We get $fg^{jk}_\ell = 1$ iff $\exists i \in C_j :\ k = \mathsf{args}^G(i)[\ell]$ for $k \in I(\mathsf{G})$, and $fg^{jk}_\ell = 1$ iff $\exists i \in C_j, u \in C_k : u = \mathsf{args}^G(i)[\ell]$ for $k \notin I(\mathsf{G})$. By definition of $\mathcal{B}^j_\ell$ and $\mathcal{K}$, we have $fg^{jk}_\ell = 1$ iff $k \in \mathcal{K}^j_\ell$, so $\mathcal{K}^j_\ell = \{k \mid fg^{jk}_\ell = 1\}$.
5. By definition of $\mathcal{F}$, since $fg^{jk}_\ell \in \{0,1\}$, also $fg^{jk} \in \{0,1\}$, and $fg^{jk} = 1$ iff at least one $fg^{jk}_\ell = 1$ for $\ell \in 2\mathbb{N}$. We get $fg^{jk} = 1$ iff $\exists i \in C_j : k \in \mathsf{args}^G(i)[\ell]$ for some $\ell \in 2\mathbb{N}$. By definition of $\mathcal{B}^j$ and $\mathcal{K}$, we have $fg^{jk} = 1$ iff $k \in \mathcal{K}^j$, so $\mathcal{K}^j = \{k \mid fg^{jk} = 1\}$. □

The following lemma intuitively proves that s^j_ℓ denotes if there are at least 2 choices for the ℓ-th input of C_j, and that these choices are captured by the set $\mathcal{K}^j_\ell$. Similarly, u^j denotes if there are at least 2 choices left for C_j in the case $\mathsf{op}^{\mathsf{G}}(j) = oc$, and that these choices are captured by $\mathcal{K}^j$.

Lemma 12. *If $g^j_j = 1$, then:*

1. $sc^j_\ell = |\mathcal{K}^j_\ell| - 1$;
2. $s^j_\ell = 0$ *iff* $|\mathcal{K}^j_\ell| = 1$;
3. $uc^j_\ell = |\mathcal{K}^j| - 1$;
4. $u^j = 0$ *iff* $|\mathcal{K}^j| = 1$.

Proof: The proof is based on the fact that all the variables are binary (proven in Lemma 8), and on the definition of constraints $\mathcal{P}$ and $\mathcal{F}$ for binary inputs (proven in Lemma 5 and Lemma 6).

By constraints (7d), $sc^j_\ell = \sum_{k \in V(\mathsf{G})} fg^{jk}_\ell - g^j_j$, and $s^j_\ell = 1$ iff at least two of fg^{jk}_ℓ are 1. By Lemma 11 we have $\mathcal{K}^j_\ell = \{k \mid fg^{jk}_\ell = 1\}$. It immediately follows that $sc^j_\ell = |\mathcal{K}^j_\ell| - 1$. If $s^j_\ell = 1$, we get $1 \neq |\mathcal{K}^j_\ell| > 2$.

If $s^j_\ell = 0$, there is at most one k such that $fg^{jk}_\ell = 1$, and it remains to prove that there is at least one such k. Suppose by contrary that there is no such k, and $|\mathcal{K}^j_\ell| = \emptyset$. Then it should be $\mathsf{args}^G(C_j)[\ell] = \emptyset$. By definition, $\mathsf{args}^G(C_j)[\ell] = \{k \mid i \in C_j, g^j_i = 1, k = \mathsf{args}^{\mathsf{G}}(i)[\ell]\}$. By assumption, we have $g^j_j = 1$, and since $j \in C_j$ and we have taken $\ell \in [\mathsf{arity}^G(j)]$, at least $k = \mathsf{args}^G(j)[\ell]$ is a suitable candidate.

The proof is analogous for u^j. ☐

The following lemma shows the relations of s^j_ℓ and b_i.

Lemma 13. *For all $i \in G$, if there exist $j \in G$, $\ell \in \mathsf{arity}^G(j)$ s.t $s^j_\ell = 1$ and $g^j_i = 1$ for $j \neq i$, then $b_i = 1$.*

Proof: The proof is based on the fact that all the variables are binary (proven in Lemma 8), and on the definition of constraints $\mathcal{P}$ and $\mathcal{F}$ for binary inputs (proven in Lemma 5 and Lemma 6).

- By constraints (9a), if $s^j_\ell = 1$, then $t^j = 1$.
- By constraints (9b), if $t^j = 1$ and $g^j_i = 1$, for $j \neq i$, then $t^j_i = 1$.
- By constraints (9d), if $t^j_i = 1$ for $j \in G$, then $b_i = 1$. ☐

Proof of the Cost of the Transformed Circuit

Theorem 5. *Let $(G, X, Y) \in \mathcal{G}$. Let solve be an arbitrary integer linear programming solving algorithm. Let $(A, \mathbf{b}, \mathbf{c}, \mathcal{I}) = T^{\rightarrow}_{ILP}(G, X, Y)$. Then*
$cost(T^{\leftarrow}_{ILP}(\mathsf{solve}(A, \mathbf{b}, \mathbf{c}, \mathcal{I}), (G, X, Y))) = \mathbf{c}^{\mathrm{T}} \cdot \mathsf{solve}(A, \mathbf{b}, \mathbf{c})$.

We prove the cost for different kinds of gates, one by one.

- **Old non-*oc* gates** Alg. 7 adds to G' exactly those gates j for which $g^j_j = 1$. While defining $\mathsf{fusable}^{\mathsf{G}}$, we agreed not to fuse the the gates whose complexity changes if their public inputs become private. Hence their total cost is $\mathsf{C}_g = \sum_{j=1, \mathsf{op}^{\mathsf{G}}(j) \neq oc}^{|G|} cost(\mathsf{op}^G(j)) \cdot g^j_j$.
- **Old *oc* gates** An old *oc* gate is replaced with an *id* by Alg. 7 if $|\mathcal{K}^j| = 1$.
  - Let for $u^j = 1$. By Lemma 12, $u^j = 1$ implies $|\mathcal{K}^j| > 1$, causing Alg. 7 to leave the *oc* into G'.
  - Let for $u^j = 0$. By Lemma 12, $u^j = 0$ implies $|\mathcal{K}^j| = 1$, and Alg. 7 does not construct an *oc* gate.

  By Lemma 12, we have $uc^j = |\mathcal{K}^j| - g^j_j$, which is the number of choices that the old *oc* gate makes in the transformed graph. We have defined the cost of an *oc* gate as $cost(oc_{base}) + cost(oc_{step}) \cdot n$, where n is the number of choices that the *oc* gate makes. Hence the total cost of the new *oc* gates is $\mathsf{C}_{oc} = \sum_{j,\ell=1,1}^{|G|, \mathsf{arity}^G(j)} cost(oc_{base}) \cdot u^j_\ell + cost(oc_{step}) \cdot uc^j_\ell$.
- **New *oc* gates** An *oc* gate is created by Alg. 7 if $|\mathcal{K}^j_\ell| > 1$.

- Let for $s_\ell^j = 1$. By Lemma 12, $s_\ell^j = 1$ implies $|\mathcal{K}_\ell^j| > 1$, causing Alg. 7 to construct a new *oc* gate v_ℓ^j that is included into G'.
- Let for $s_\ell^j = 0$. By Lemma 12, $s_\ell^j = 0$ implies $|\mathcal{K}_\ell^j| = 1$, and Alg. 7 does not construct an *oc* gate.

By Lemma 12, we have $sc_\ell^j = |\mathcal{K}_\ell^j| - 1$, which is the number of choices that the new *oc* gate makes. The total cost of the new *oc* gates is $\mathsf{C}_{oc} = \sum_{j,\ell=1,1}^{|G|,\mathsf{arity}^G(j)} cost(oc_{base}) \cdot s_\ell^j + cost(oc_{step}) \cdot sc_\ell^j$.

– **New G_k gates** These gates are computing the conditions vb_k. For the old *oc* gates, vb_k are just some wire names, and the only other introduced operation is addition Hence these gates may have some cost only if they are composed for a new *oc* gate. Let us assume that Alg. 8 has been called by Alg. 7 on some $\mathcal{B}_\ell^j$.
For simplicity, we defined Alg. 8 in such a way that it does not assign special default choices and uses only the weakest preconditions straightforwardly. However, since *oc* gates are correctly defined (by Thm .4), all the vb_k arguments sum up either to 0 or 1. If they sum up to 0, then the weakest preconditions of all the choices are 0, and hence the output of *oc* does not matter anyway, so we may as well make one of the choices 1. If they do sum up to 1, then one of the vb_k arguments of the *oc* is actually a linear combination of the other its vb_k arguments. We may let Alg. 7 to choose any k to be the default one. Without loss of generality, let it be $k = \mathsf{args}^G(j)[\ell]$. For $g_j^j = 1$, we have $b_j = 0$ due to constraints (9a-9d). For $k \neq \mathsf{args}^G(j)[\ell]$, Alg. 7 does include G_k into G'.
We need to show that including G_k for $k \neq \mathsf{args}^G(j)[\ell]$ into G' indeed implies including all ϕ_i^{G} for $i \neq j$. Suppose by contrary that there is some $i \neq j$ that only occurs in G_k for $k' = \mathsf{args}^G(j)[\ell]$, regardless of the choice of ℓ. In other words, each ℓ-th input of i is the ℓ-th input of j, and hence $\mathsf{args}^G(i) = \mathsf{args}^G(j)$. Since $\mathsf{op}^G(i) = \mathsf{op}^G(i)$, and the gates are unique, we have $i = j$. This contradicts the assumption $i \neq j$, so ϕ_i^{G} should have been used in at least one vb_k for $k \neq \mathsf{args}^G(j)[\ell]$.
We get that the gates of ϕ_i^{G} are included into G' iff $b_j = 1$. For all $k \in C_j$, Alg. 7 takes

$$vb_k = \sum_{(\phi_i^{\mathsf{G}}[i' \leftarrow j' \mid g_{i'}^{j'}=1],k)\in\mathcal{B}} \phi_i^{\mathsf{G}}[i' \leftarrow j' \mid g_{i'}^{j'} = 1] \ .$$

Substituting i' with j' does not affect the cost of ϕ_i^{G}, since by definition, $cost(\phi_i^{\mathsf{G}})$ only counts the $\vee$ and $\wedge$ operations of ϕ_i^{G}. The costs of j' are already included in C_g, so they do not have to be computed again. Without merging the repeating subexpressions that may occur in different ϕ_i^{G}, the cost of G_k is $\mathsf{C}_b = \sum_{j=1}^{|G|} cost(\phi_j^{\mathsf{G}}) \cdot b_j$.

In total, we get the cost $\mathsf{C}_g + \mathsf{C}_{oc1} + \mathsf{C}_{oc2} + \mathsf{C}_b$. □

H.3 Applying back the greedy algorithm solution

We show that we can use Alg. 2 for constructing the optimized circuit from the set of cliques returned by a greedy algorithm.

Theorem 6. *Let $(G,X,Y) \in \mathcal{G}$. Let* greed *be the function of Alg. 2 that returns the set of cliques of gates of G. There exists a transformation T_G^C such that $[\![\mathtt{eval}(G,X,Y)]\!](s) = [\![\mathtt{eval}(T_{ILP}^{\leftarrow}(T_G^C(\mathsf{greed}(G),X,Y)))]\!](s)$ for any $s \in State$.*

Proof By Lemma 4, Alg. 2 terminates. Let Cs be the set of cliques returned by the greedy algorithm. We now may reduce the cliques to an IP solution as follows. First, sort each $C \in Cs$ by the gate indices, so that the first index is the smallest one. For all $C \in Cs$, $j = C[0]$, assign $g_i^j = 1$ for all $i \in C$. We show that this assignment satisfies IP constraints.

– In Alg. 2, the gates are first of all sorted by types, and only the gates of the same type are fused. This satisfies the constraint (3) of IP.
– As soon as a gate has been taken into a clique, this is not added to any other clique. If any gate is left, it is treated as a singleton clique. This satisfies the constraint (2).
– We have taken $g_i^j = 1$ for $j = C[0]$, so j is the unique clique representative, and (4) is satisfied.
– Alg. 2 stops when it reaches gates of cost 0 and puts all of them into separate cliques, so that (5) is satisfied.
– Whatever greedy strategy we take, each of them accepts a clique iff the function goodClique returns true. For any gates i, j that belong to the same clique C, $\mathsf{fusable}^{\mathsf{G}}(i,j)$ holds. This satisfies the constraint (1) of IP. In addition, goodClique assigns levels to the gates, which can be put into one-to-one correspondence with the variables ℓ_i of the constraint (6). For $\mathsf{pred}^{\mathsf{G}}(i,k)$, we have $\mathsf{level}(i) > \mathsf{level}(k)$, and for $\mathsf{cpred}^{\mathsf{G}}(i,k)$ we have $\mathsf{level}(i) > \mathsf{level}(k)$ iff the size of the clique to which i belongs is at least 2. The function goodClique checks it on the fly, and since if a clique has already been accepted, it will never be updated anymore, the condition will not be broken. □

I The Sample Programs

I.1 driver

```
def mean(x):
    return floatMult(floatSum(x), inv(length(x)));

def entropy (x):
    return floatNeg(floatSum(floatMult(x, ln(x))));

def divide (x1, x2):
    return floatMult (x1, inv(x2));

def AP (v):
    return mean (v);

def AD (d):
    return mean (d);

def PE (n, t):
    return entropy (divide (n, t));

def PIE (i, ti):
    return entropy (divide (i, ti));

def main:
    private v, d, n, i, t, ti;
    input v, d, n, i, t, ti;

    private ap := AP(v);
    private ad := AD(d);
    private pe := PE(n,t);
    private pie := PIE(i,ti);

    if ap <= 900:
        if ad <= 13:
            if pe <= -5:
                y := 6
            else:
                y := 7;
        else:
            if pie <= -6:
                y := 5
            else
                y := 9;
    else:
        if ad <= 21:
            if pe <= -3:
                y := 8
            else:
                y := 7;
        else:
            if pie <= -4:
                y := 4
            else:
                y := 3;
    return y;
```

I.2 sqrt

```
def main:
    private a, x, y, mid;
    private answer := -1;
    input a;
    x := 0;
    y := a;
    mid := a >> 1;
    for i in range(10):
        if (mid * mid > a):
            y := mid;
            mid := (x + mid) >> 1;
        else if (mid * mid < a):
            x := mid;
            mid := (mid + y) >> 1;
        answer := mid;
    return answer;
```

I.3 loan

```
def main:
    private age, num_of_parents, num_of_children,
            income, answer;
    input age, num_of_parents, num_of_children,
            income;

    if age < 18:
        answer := 0
    else if age < 65:
        if num_of_children == 0:
            if income > 20:
                answer := 1
            else:
                answer := 0
        else if num_of_parents == 1:
            if income > 25:
                answer := 1
            else:
                answer := 0
        else:
            if income > 30:
                answer := 1
            else:
                answer := 0
    else if income > 40:
        answer := 1
    else:
        answer := 0;
    return answer;
```

I.4 stats

```
def mean (x):
    return floatMult (floatSum(x), inv(length(x)));

def variance (x):
    private w := floatMult (floatSquare (floatSum(x));
    private z1 := floatNeg (w, inv(n)));
    private z2 := floatSum (floatSquare(x));
    return floatMult (floatAdd (z1,z2), inv(n - 1));

def sdev (x):
    return sqrt (variance (x));

def sdev (x, y):
    public nxy := length(x) + length(y) - 2;
    public nx := length(x) - 1;
    public ny := length(y) - 1;
    private vx := floatMult (variance (x), nx);
    private vy := floatMult (variance (y), ny);
    return sqrt (floatMult (floatAdd (vx,vy), inv(nxy)));

def studenttest (x, y):
    private mx := mean (x);
    private my := mean (y);
    private z1 := floatAdd (mx, floatNeg(my));

    public nx := inv(length(x));
    public ny := inv(length(y));
    private sxy := sdev (x, y);
    public w := sqrt (floatAdd (n1, n2));
    private z2 := floatMult (sxy, w);

    return floatMult (z1,inv (z2));

def welchtest (x, y):
    private mx := mean (x);
    private my := mean (y);
    private z1 := floatAdd (mx, floatNeg(my));

    public nx := inv(length(x));
    public ny := inv(length(y));
    private vx := floatMult (variance (x), nx);
    private vy := floatMult (variance (y), ny);
    private z2 := sqrt (floatAdd (vx,vy));
    return floatMult (z1, inv(z2));

def wilcoxontest (x, y):
    private d := floatAdd (x, floatNeg(y));
    private s := floatSign (d);
    private dpr := floatAbs (d);

    s := sort (dpr, s);
    private r := rank0 (s);
    return floatSum (floatMult (s, r));

def contingencytable (x, y, c):
    private ct_x := floatOuterEqualitySums (x,c);
    private ct_y := floatOuterEqualitySums (y,c);
    return (ct_x, ct_y);

def chisquaretest (x, y, c);
    private ninv := inv (length(x));

    private (ct_x,ct_y) := contingencytable (x, y, c);
    private rx := floatSum (ct_x);
    private ry := floatSum (ct_y);
    private p  := floatAdd (ct_x,ct_y);

    private ex := floatMult (floatMult(p,rx), ninv);
    private ey := floatMult (floatMult(p,ry), ninv);

    private nex := floatNeg(ex);
    private ney := floatNeg(ey);
    private z1 := floatSquare (floatAdd (ct_x, nex));
    private z2 := floatSquare (floatAdd (ct_y, ney));

    private w1 := floatSum (floatMult (z1,inv(ex)));
    private w2 := floatSum (floatMult (z2,inv(ey)));
    return floatAdd (w1,w2);

def main:
    private result;
    private b1;  #is the distribution normal
    private b2;  #are the stdev and the mean known
    private b3;  #is it ordinal data

    input b1, b2, b3;

    # The first dataset
    private x;
    input x;

    # The second dataset
    private y;
    input y;
    # The set of possible classes for chi-squared test
    private c;
    input c;

    if b1 == 1:
        if b2 == 1:
            result := studenttest (x, y);
        else:
            result := welchtest (x, y);
    else:
        if b3 == 1:
            result := wilcoxontest (x, y);
        else:
            result := chisquaretest (x, y, c);

    return result;
```

I.5 erf

```
#fixpoint to floating point
def fix_to_float (y,t,n,q):
    private u := getbit (y, t);
    private s := 1;
    private e;
    private f;
    if u == 1:
        e := t + q + 1;
        f := y * (1 << (n-t-1));
    else
        e := t + q;
        f := y * (1 << (n-t));
    return (s,e,f);

#Multiply two floating point numbers
def float_mult (s1,e1,f1,s2,e2,f2,n):

    private lambda;
    s := (s1 == s2);
    e := e1 + e2;
    f := f1 * f2;
    #here --> and <-- are ring conversion operations
    f := ((f1 --> (2*n))*(f2 --> (2*n))) <-- n;
    lambda := f >> (n-1);
    if lambda == 0:
        f := f << 1;
        e := e - 1;
    return (s,e,f);

#Evaluate a polynomial of degree <= 12
def eval (x0,s,c):
    private x[13];
    x[0] := 1;
    x[1] := x0;
    x[2] := x0 * x0;
    x[3] := x[2] * x[1];
    x[4] := x[2] * x[2];
    x[5] := x[4] * x[1];
    x[6] := x[4] * x[2];
    x[7] := x[4] * x[3];
    x[8] := x[4] * x[4];
    x[9] := x[8] * x[1];
    x[10] := x[9] * x[2];
    x[11] := x[8] * x[3];
    x[12] := x[8] * x[4];

    private z1[13] := 0;
    private z2[13] := 0;

    for i in range(13):
        if s[i] == 1:
           z1[i] := z1[i] + c[i] * x[i];
        else if s[i] == -1:
           z2[i] := z2[i] + c[i] * x[i];

    return z1 - z2;


def gaussian_poly_0 (x):
    return eval (x,[0,1,-1,-1,-1,1,-1,-1,-1,1,-1,-1,1],
               [0, 37862129, 89, 12620065, 3115, 3797002, 27323,
                850652, 68415, 238867, 35736, 22843, 6588]);

def gaussian_poly_1 (x):
    return eval (x,[1,1,1,-1,1,-1,-1,1,1,1,1,1,1],
                [945472, 31405311, 18236798, 40079935, 23153761,
                5984925, 599861, 0, 0, 0, 0, 0, 0]);

def gaussian_poly_2 (x):
    return eval (x,[-1,1,-1,1,1,1,-1,1,1,1,1,1,1],
               [31613609, 134982639, 119986495, 59088711, 17266836,
               2930966, 247133, 3236, 636, 0, 0, 0, 0]);

def gaussian_poly_3 (x):
    return eval (x,[1,1,-1,1,-1,1,-1,1,1,1,1,1,1],
               [28778930, 7535740, 4967310, 1750656, 347929,
                36972, 1641, 0, 0, 0, 0, 0, 0]);

def main:
    #the input
    private s, e, f;
    input s, e, f;
```

```
public q := (1 << 14) - 1;
public n := 32;
public m := 25;

public shift0 := n - m + 0 - 2;
public shift1 := n - m + 1 - 2;
public shift2 := n - m + 2 - 2;
public shift3 := n - m + 3 - 2;
public shift4 := n - m + 4 - 2;

private f0 := f >> shift0;
private f1 := f >> shift1;
private f2 := f >> shift2;
private f3 := f >> shift3;
private f4 := f >> shift4;

private g0, g_1, g_1_0, g_1_1;

g0 := gaussian_poly_1 (f0);
g_1_0 := gaussian_poly_2 (f0);
g_1_1 := gaussian_poly_3 (f0);

private u := f <- m;
if u == 1:
    g_1 := g_1_1
else
    g_1 := g_1_0;

public t_1 := 0;
public t0 := 0;
public t1 := 2 - 1;
public t2 := 2 - 2;
public t3 := 2 - 3;
public t4 := 2 - 4;

if e <= q - 4:
    e := q - 4;
if q + 3 <= e:
    e := q + 3;

private s_pr, e_pr, f_pr;
private index := e - q;

if index == -2:
    (s_pr,e_pr,f_pr) := float_mult(1,25,21361415,s,e,f,n);
else if index == -1:
    (s_pr,e_pr,f_pr) := fix_to_float (g_1,t_1,n,q);
else if index == 0:
    (s_pr,e_pr,f_pr) := fix_to_float (g0,t0,n,q);
else if index == 1:
    private g1 := gaussian_poly_0 (f1);
    (s_pr,e_pr,f_pr) := fix_to_float (g1,t1,n,q);
else if index == 2:
    private g2 := gaussian_poly_0 (f2);
    (s_pr,e_pr,f_pr) := fix_to_float (g2,t2,n,q);
else if index == 3:
    private g3 := gaussian_poly_0 (f3);
    (s_pr,e_pr,f_pr) := fix_to_float (g3,t3,n,q);
else if index == 4:
    private g4 := gaussian_poly_0 (f4);
    (s_pr,e_pr,f_pr) := fix_to_float (g4,t4,n,q);
else
    (s_pr,e_pr,f_pr) :=  (1, 0, 1);

return (s_pr, e_pr, f_pr);
```