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 computation platforms are often provided with a programming language that allows a developer to write privacy-preserving applications and hides away the underlying cryptographic details. The control flow of these programs is expensive to hide, hence branching on private values is often disallowed. 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 [1–3], as well as secure computation platforms implementing these methods [4–7]. 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, 8–11] or in a subset of some general language, e.g. C, possibly with extra privacy annotations [12, 13]. Operations with private values are compiled to protocols transforming the representations of inputs of these operations to the representation of the output.

In secure multiparty computation (SMC) protocol sets based on secret sharing [2, 3, 14, 15], the involved parties are usually partitioned into input, computation, and output parties [16]. The computation parties are holding the private values in secret-shared form between them, and are performing the bulk of computation and communication. In this setting, `if`- and `switch`-statements with *private* conditions are among unsupported operations. Namely, the taken branch should not be revealed to anyone, but it is difficult to hide the control flow of the program. Instead of choosing the right branch, all the branches are executed, and the final values of all program variables are chosen obliviously from the outcomes of all branches [13,17]. This introduces a significant overhead. An obvious optimization idea, which has not received much attention so far except for [18] 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.

In this work, we consider a simple imperative language with variables typed "public" and "private", invoking secure protocols to process private data. The use of expressions typed "private" is allowed in the conditions of `if` and `switch` statements. We translate a program written in this language into a computational circuit and optimize it, trying to fuse together the sets of operations, where the outcome of at most one of them is used in any concrete execution. Our optimization is based on *mixed integer linear programming*, but some greedy heuristics are proposed as well for better performance. Our optimization is very generic and can be applied on the program level, without decomposing high-level operations to arithmetic or boolean circuits. We do the optimization for some simple programs with private conditionals, and evaluate them top of the Sharemind SMC platform [14], showing that the optimization is indeed useful in practice.

2 Preliminaries

Secure computation There is a computing party (or several parties) whose task is to compute some function on secret inputs, without being allowed to infer any information about the inputs and/or the outputs. The inputs for such a function may be either encrypted on secret shared among several computing parties.

Languages for secure computation A privacy-preserving application is often described as a higher level functionality, without taking into account the underlying cryptographic protocols. Existing platforms usually come with a language [4,13,14,19] to program such applications. 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.

Mixed integer linear programming [20] A *mixed integer linear programming* is an optimization task stated as

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

where $\mathbf{x} \in \mathbb{R}^n$ is a vector of variables that are optimized, and the quantities $A \in \mathbb{R}^{m \times n}$, $\mathbf{b} \in \mathbb{R}^n$, $\mathbf{c} \in \mathbb{R}^n, \mathcal{I} \subseteq \{1, \dots, n\}$ are the parameters defining the task.

3 Related Work

There are a number of languages for specifying privacy-preserving applications to be run on top of secure computation platforms. These may be either domain-specific languages [4, 8, 10] 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 [9], as well as in newer languages and frameworks, such as PICCO [13], Obliv-C [21], Wysteria [22], SCVM [23], or the DSL embedded in Haskell by Mitchell et al. [11]. 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 branch is updated by selecting its value obliviously. Planul and Mitchell [17] 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.

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 [18], 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.

4 Programs and Circuits

The imperative language with private conditionals We start from a simple imperative language, 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 special operation `declassify` 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 [13, 17]. The syntax c denotes compile-time constants.

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

During the execution of a program on top of a secure computation platform, public values are known by all computation parties, while private values are either encrypted or secret-shared among them [8]. 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 [14].

Computational circuits Due to the existence of private conditionals, the programs written in this language need to be translated into computational circuits before execution. 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 [13].

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 obtain their values externally.

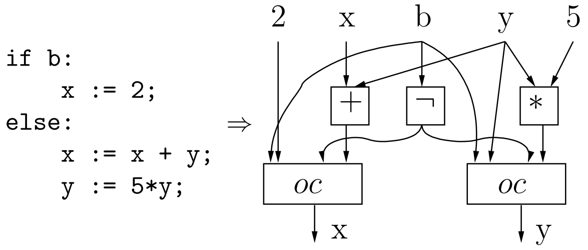
Definition 1. *Let Var be the set of program variables. A computational circuit is defined as a set of gates $G = \{g_1, \ldots, g_m\}$ for some $m \in \mathbb{N}$, where each gate $g \in G$ is defined as $g = (op, [v_1, \ldots, v_n])$:*

- *op is the operation that the gate computes (an arithmetic blackbox function of the secure computation platform);*
- *$[v_1, \ldots, v_n]$ for $v_i \in G \cup Var$ is the list of the arguments to which the operation op is applied when the gate is evaluated.*

For $g = (op, args) \in G$, we write $\mathsf{op}^G(g) = op$, and $\mathsf{args}^G(g) = args$.

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 a gate g has $\mathsf{op}^G(g) = oc$, $\mathsf{args}^G(g) = [b_1, v_1, \ldots, b_n, v_n]$), and it returns the output of v_i iff the output of b_i is 1. If there is no such b_i, then it outputs 0. It works on the assumption that at most one gate b_i outputs 1.

Transforming programs to circuits 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 $x_1, \ldots, x_n$. A sequence of assignments is put together into a single circuit using circuit composition.



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

Fig. 1. Example of program transformation

If the program statement is not an assignment, but a private conditional statement, all its branches are first transformed to independent circuits G_i. Let g_i^y be the gate of G_i that outputs the value of y. The value of each variable y is then selected obliviously amongst the output vertices $g_i^y \in G_i$, where the choice bit b_i of g_i^y is the condition of executing the i-th branch. So far, the transformation is similar to the related work [13,17]. An example of transforming a conditional statement to a circuit is given in Fig.1.

5 Optimizing the Circuit

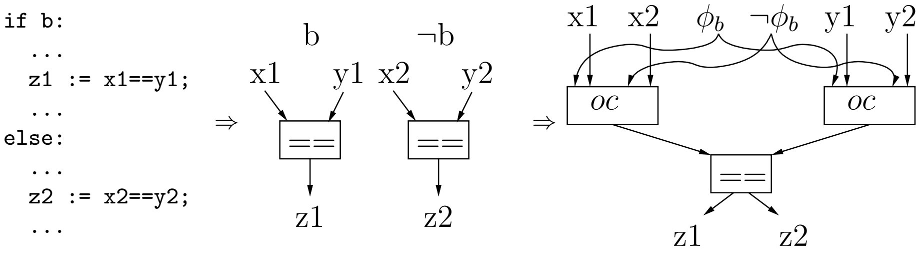
Let G be a computational circuit. Without loss of generality, let $G = \{1, \ldots, n\}$. The *weakest precondition* ϕ_g^G of evaluating a gate $g \in G$ is a boolean expression over the conditional variables, such that $\phi_g^G = 1$ iff the gate g is evaluated for the given valuation of conditional variables.

The main idea of our optimization is the following. Let $g_1, \ldots, g_k \in G$ be the gates such that $\mathsf{args}^G(g_i) = [x_1^i, \ldots, x_n^i]$, and $\mathsf{op}^G(g_i) = op$ for all i. Let $\phi_{g_1}^G, \ldots, \phi_{g_k}^G$ 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. An example of optimizing a circuit of two branches is given in Fig. 2.

As discussed in Sec. 4, private branches are not allowed to assign to public variables, so we are fusing only private gates. Hence such a transformation does not affect any public variables through which some additional data might have leaked, and it does not modify the privacy of the initial program.

5.1 High-Level Overview

Preprocessing First, we look for the pairs of mutually exclusive gates. Find the weakest precondition ϕ_i^G of each gate i. For $i, j \in G$, define $\mathsf{mex}^G(i, j) = 1$ iff $(i = j) \vee (\phi_i^G \wedge \phi_j^G$ is unsatisfiable). For a correct (but not necessarily optimal) solution, it suffices to find only a subset of mutually excluding pairs.



```
if b:
  ...
  z1 := x1==y1;
  ...
else:
  ...
  z2 := x2==y2;
  ...
```

Fig. 2. An example of gate fusing

Since fusing forces all the gate arguments to become chosen obliviously, all the inputs of a fused gate in general are treated as private. Depending on the secure computation platform and the particular operation, this may formally change the gate operation, possibly becoming more expensive, or even unsupported. We define $\mathsf{mex}^G(i, j) = 0$ for the gates i or j that have any public inputs, and whose cost may change if these inputs become private.

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

- $\forall i, j \in C_k : \;\; i \neq j \implies \mathsf{mex}^G(i,j) = 1$: we fuse only mutually exclusive gates, so that indeed at most one gate of C_k will actually be evaluated.
- $\forall i, j \in C_k : \; \mathsf{op}^G(i) = \mathsf{op}^G(j)$: only the same operation gates are fused.
- Let $E := \{(C_i, C_j) \mid \exists k, \ell : k \in C_i, \ell \in C_j, \ell \in \mathsf{args}^G(k)\}$. The relation $(C_i, C_j) \in E$ denotes that C_j should be evaluated strictly before C_i. We require that the graph $(\{C_k\}_k, E)$ should be acyclic.

If we treat the gates G as vertices, and the relation $\mathsf{mex}^G(i,j)$ as edges, we get that C_k are disjoint cliques on this graph. A possible fusing of gates into a clique is shown in Fig. 3, where the gray lines connect the gates for which $\mathsf{mex}^G(i,j)$ holds, and the shaded gates are treated as a single clique.

To ensure that the optimized circuit is acyclic, define the following predicates, that can be easily derived from the initial circuit:

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

The predicate $\mathsf{pred}^G(i,k)$ is true just if k is an immediate predecessor of i in G. The predicate $\mathsf{cpred}^G(i,k)$ 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 choice of the arguments of i, and in this case k has to be computed strictly before i. We call k a *conditional* predecessor of i.

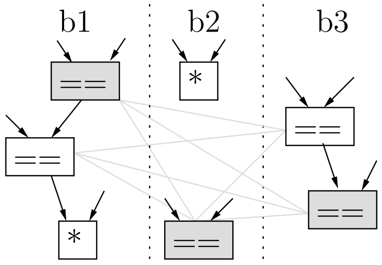


Fig. 3. Fusing gates into cliques

The number of cliques may vary between 1 and $|G|$. 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, which is the only gate of C_j left after the fusing.

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. Any gate in the rest of the circuit that has used some g_i as an input should now use g instead.

The Cost In a privacy-preserving application, each gate operation corresponds to some cryptographic protocol. In SMC platforms, such protocols require communication between the parties. We choose the total number of communicated bits as the cost. This metric is additive w.r.t the cost of individual gates. We note that introducing new oblivious choices may increase the number of rounds.

Greedy Construction of Cliques First, we propose some simple heuristic optimizations of the cost. The gates G are partitioned into subsets, grouped by their operation. These subsets are sorted according to the cost of their operation, so that more expensive gates come first. 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 $i, j \in C_k$ satisfy $\mathsf{mex}^G(i, j) = 1$;
- no $i \in C_k$ has already been included into some other clique;
- C_k does not introduce cycles.

We use two main greedy strategies for forming a set of cliques for a particular subset of gates. The first one merges the largest possible clique first. The second one merges the cliques pairwise, trying to form as many cliques as possible.

5.2 Reduction to an integer programming task

As an alternative to greedy algorithms, we reduce the gate fusing task to an integer program and solve it using an external solver (such as [24]).

We consider mixed integer programs of the form (1), defined as a tuple $(A, \mathbf{b}, \mathbf{c}, \mathcal{I})$. We describe how these quantities are constructed from G.

Variables The core of our optimization are the variables that affect the cost of the transformed circuit. We also need some variables that help to avoid cycles. For all $i, j \in G$, we define the following variables:

- $g_i^j \in \{0, 1\}$, $g_i^j = 1$ iff g_i belongs to the clique C_j.
  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.
- $\ell_j \in \mathbb{R}$ is the circuit topological level on which the gate j is evaluated. All the gates with the same level are evaluated simultaneously. Each gate must have a strictly larger level than all its predecessors.
- $c_j \in \{0, 1\}$, $c_j = 1$ iff the gate g_j is fused with some other gate.
  Each gate should have a strictly larger level than all its *conditional* predecessors iff it it is merged with some other gate.

Cost function We minimize the value $\sum_{j=1}^{|G|} cost(\mathsf{op}^G(j)) \cdot g_j^j$, which is the total cost of the gates left after fusing, except the new oblivious choice gates, and the new boolean operations possibly introduced for the weakest preconditions. This is sufficient as far as the cost of the oblivious choice and the bit operations is smaller than the cost of the gate operation being merged.

Inequality constraints The constraints $A\mathbf{x} \leq \mathbf{b}$ state the relations between the variables. 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 We describe how some logical statements are encoded as sets of constraints. Their correctness can be easily verified by case distinction.

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. We denote this set of constraints $\mathcal{P}(C, x, y, z)$.
- $C \cdot x + y - z \leq C$;
- $C \cdot x - y + z \leq C$;
- $C \cdot x - z \geq 0$.

Threshold. $y = 1$ if $\sum_{x \in \mathcal{X}} x \geq A$, and $y = 0$ otherwise, for $\forall x \in \mathcal{X} : x \in \{0, 1\}$, $y \in \mathbb{R}$, some constant A. We denote this set of constraints $\mathcal{F}(A, \mathcal{X}, y)$.
- $\mathcal{P}(1, y, x, z_x)$ for all $x \in \mathcal{X}$, where z_x are fresh variable names;
- $A \cdot y - \sum_{x \in \mathcal{X}} z_x \leq 0$;
- $\sum_{x \in \mathcal{X}} x - \sum_{x \in \mathcal{X}} z_x + (A - 1)y \geq (A - 1)$.

Implication of 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. We denote this constraint by $\mathcal{G}(C, A, x, y, z)$.
- $(C + A) \cdot z + y - x \geq C$.

Structural Constraints These ensure that the fusing forms a correct graph.

1. Only mutually exclusive gates may belong to the same clique.
   $g_i^j + g_k^j \leq 1$ for $i, k \in G$, $\neg\mathsf{mex}^G(i, k)$.
2. Each gate belongs to exactly one clique.
   $\sum_{j=1}^{|G|} g_i^j = 1$ for all $i \in G$.
3. If the clique C_j is non-empty, then it contains the gate j. This makes gate j the representative of C_j.
   $g_j^j - g_i^j \geq 0$ for all $i, j \in G$.
4. We assign an operation to each clique, based on its representative: $\mathsf{op}^G(C_j) = \mathsf{op}^G(j)$. The gate i may to belong to C_j only if $\mathsf{op}^G(C_j) = \mathsf{op}^G(i)$.
   $g_i^j = 0$ if $\mathsf{op}^G(i) \neq \mathsf{op}^G(j)$.
5. Prevent from fusing gates with cost 0, reducing the search space.
   $g_j^j = 1$ for all j such that $cost(\mathsf{op}^G(j)) = 0$.
6. The cliques are not allowed to form cycles. We assign a level ℓ_i to each gate i. If k is a predecessor of i, then $\ell_k < \ell_i$. To avoid degenerate solutions to the ILP, we introduce some difference between the levels: $\ell_i - \ell_k \geq 1$. If a gate i belongs to the clique C_j, then $\ell_i = \ell_j$. We take $|G|$ as the maximal value for ℓ_i, since we need at most $|G|$ distinct levels.
   (a) $\ell_i - \ell_k \geq 1$ for all $i, k \in G, \mathsf{pred}^G(i, k)$;
   (b) $\mathcal{G}(|G|, 0, \ell_i, \ell_j, g_i^j)$, $\mathcal{G}(|G|, 0, \ell_j, \ell_i, g_i^j)$ for all $i, j \in G$;
   (c) $\ell_i \geq 0, \ell_i \leq |G|$.
   We need to take into account the conditional predecessors. Let $c_j = 1$ iff the gate j is fused with some other gate. That is, either $g_j^j = 0$ (j belongs to some other clique), or $\sum_{i \in G, i \neq j} g_i^j \geq 1$ (there is some other gate in C_j).
   (d) $d_j = (1 - g_j^j)$ for all $j \in G$;
   (e) $\mathcal{F}(1, \{d_j\} \cup \{g_i^j \mid i \in G, i \neq j\}, c_j)$ for all $j \in G$;
   Finally, if $c_i = 1$ (i is fused with some other gate), then $l_k - l_i \geq 1$ (i is computed strictly before its conditional predecessor k).
   (f) $\mathcal{G}(|G|, 1, \ell_i, \ell_k, c_i)$ for all $i, k \in G$, $\mathsf{cpred}^G(i, k)$;

Binary constraints Dealing with a mixed integer program, we need to state explicitly that $g_i^j \in \{0,1\}$ for all $i,j \in G$. The condition $c_j \in \{0,1\}$ may remain implicit, as it follows from $g_i^j \in \{0,1\}$ and the inequality constraints for c_j.

6 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 [25]. The ILP is solved externally by the GLPK solver [24].

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.

• loan (31 gates, integer): A simple binary decision tree, which decides whether a person should be given a loan, based on its background. Only the comparisons that are needed for making decisions are fused.

• sqrt (123 gates, integer): 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.

• driver (53 gates, floating point [26]): 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 [27], assuming a privacy-preserving setting of this task. Some decisions require more complex operations, such as logarithms and inverses.

• stats (68 gates, floating point): choosing a particular statistical test 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 program chooses amongst the Student t-test, the Wilcoxon test, the Welch test, and the χ^2 test, whose privacy-preserving implementations are taken from [28].

• erf (335 gates, integer): The program evaluates the error function of a floating point number, which is represented as a triple (sign, significand, exponent) of integers [26]. The implementation is taken from [29]. The method chosen to compute the answer depends on the range in which the initial input is located.

All our programs are vectorized. During optimization, we treated vector operations as single gates. 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 optimization times are given in the Table 1. The rows correspond to different strategies, where lp_1 is the ILP approach described in Sec. 5.2, and lp_2 is an extended ILP that takes into account the new oblivious choice gates, and the gates computing weakest preconditions. The columns are the programs, where the numbers after the program name (for stats and erf) specify the depths of the subcircuits into which the gates were merged before optimization, being treated as single gates. In general, there was little variation in optimization times for depths of 2 or more. The greatest difference was noted for stats and erf, where the

Table 1. Optimization times

	times (ms)					times (s)	
	driver	sqrt	loan	stats, 0	stats, 1-5	erf, 0	erf, 1-7
greed	75-125	495-571	41-53	103-123	181-185	43.5	2.5-8.5
lp$_1$	118-181	584-780	83-97	142-170	292	47	2.9-8.2
lp$_2$	171-381	593-1300	115-121	214-262	16291	47.6	15.3-52.5

Table 2. Execution times

$n = 10$	driver	sqrt	loan	erf	stats
time (ms)	156-193	71-77	12-15	85-121	1700-1750
w/o opt.	156	73	16	91	1760
best strat.	greed$_1$	depth 0	depth 0	depth 1,5+	lp$_1$, lp$_2$
worst strat.	greed$_1$ depth 0	depth 1	greed$_2$ depth 0	depth 2-4	greed$_1$,greed$_2$

$n = 10^3$	driver	sqrt	loan	erf	
time (ms)	588-809	249-291	32-41	275-334	
w/o opt.	705	283	51	316	
best strat.	lp$_1$, lp$_2$, greed$_1$	depth 0	depth 0	depth 1,4+	
worst strat.	greed$_1$ depth 0	greed$_1$ depth 3	greed$_2$ depth 0	depth 2,3	

$n = 10^6$	driver	sqrt	loan	erf	stats, $n = 100$
time (s)	200-336	97-120	10-14	95-111	136-146
w/o opt.	256	121	19.5	108	148
best strat.	lp$_1$, lp$_2$, greed$_1$	depth 0	depth 0	depth 1,4+	no preference
worst strat.	greed$_1$ depth 0	depth 2+	greed$_2$ depth 0	depth 2,3	no preference

optimization time was significantly larger without constructing the subcircuits first. The overhead comes mainly from looking at all possible pairs of mutually exclusive gates. The details of lp$_2$, as well as the construction of subcircuits from gates, are given in the full version of this paper [30].

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. Each test was run 100 times on $n = 10$, $n = 10^3$ inputs, and 10 times on $n = 10^6$ inputs. The summary of the results is given in Table. 2. For each program, we give the runtime range of its optimized versions, the runtime of the non-optimized version, and which strategies have been the best and the worst. Here greed$_1$ is the strategy that chooses the largest clique first, and greed$_2$ fuses the gates pairwise.

Since the runtime depends also on the number of rounds that we did not optimize, our results are not good for small inputs. However, as the total amount of communication and computation increases, our optimized programs are becoming more advantageous. While greedy approaches may outperform ILP approaches for smaller inputs, ILP is more stable for large 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.

7 Conclusion

We have presented a generic optimization for programs written in imperative languages for privacy-preserving computation platforms. We have benchmarked some programs on Sharemind, and see that we indeed can obtain better runtimes. As future work, we might consider decomposing blackbox operations deeper into subprotocols, allowing to partially fuse different blackbox operations.

Acknowledgements. Supported by Estonian Research Council, grant IUT27-1.

References

1. Yao, A.C.: Protocols for secure computations (extended abstract). In: *CSF 1982*, pp. 160–164. IEEE Computer Society (1982)
2. Goldreich, O., Micali, S., Wigderson, A.: How to play any mental game or A completeness theorem for protocols with honest majority. In: Aho, A.V. (ed.), *STOC 1987*, pp. 218–229. ACM (1987)
3. Cramer, R., Damgård, I., Maurer, U.M.: General secure multi-party computation from any linear secret-sharing scheme. In: Preneel, B. (ed.), *EUROCRYPT*, vol. 1807 of *LNCS*, pp. 316–334. Springer (2000)
4. Malkhi, D., Nisan, N., Pinkas, B., Sella, Y.: Fairplay - a secure two-party computation system. In: *SSYM 2004, USENIX Security Symposium*, pp. 287–302, Berkeley, CA, USA. USENIX Association (2004)
5. Bogdanov, D., Laur, D., Willemson, J.: Sharemind: A framework for fast privacy-preserving computations. In: Jajodia, S., López, J. (eds.), *ESORICS*, vol. 5283 of *LNCS*, pp. 192–206. Springer (2008)
6. Burkhart, M., Strasser, M., Many, D., Dimitropoulos, X.: SEPIA: Privacy-preserving aggregation of multi-domain network events and statistics. In: *SSYM 2010, USENIX Security Symposium*, pp. 223–239, Washington, DC, USA. USENIX Association (2010).
7. Demmler, D., Schneider, T., Zohner, M.: ABY - A framework for efficient mixed-protocol secure two-party computation. In: *NDSS 2015*. The Internet Society (2015)
8. Bogdanov, D., Laud, P., Randmets, J.: Domain-polymorphic programming of privacy-preserving applications. In: Russo, A., Tripp, O. (eds.), *PLAS@ECOOP 2014*, page 53. ACM (2014)
9. Nielsen, J.D., Schwartzbach, M.I.: A domain-specific programming language for secure multiparty computation. In: Hicks, M.W. (ed.), *PLAS 2007*, pp. 21–30. ACM (2007)
10. Schröpfer, A., Kerschbaum, F., Müller, G.: L1 - an intermediate language for mixed-protocol secure computation. In: *COMPSAC 2011*, pp. 298–307. IEEE Computer Society (2011)
11. Mitchell, J.C., Sharma, R., Stefan, D., Zimmerman, J.: Information-flow control for programming on encrypted data. In: Chong, S. (ed.), *CSF 2012*, pp. 45–60. IEEE Computer Society (2012)

12. Franz, M., Holzer, A., Katzenbeisser, S., Schallhart, C., Veith, H.: CBMC-GC: an ANSI C compiler for secure two-party computations. In: Cohen, A. (ed.), *CC 2014*, vol. 8409 of *LNCS*, pp. 244–249. Springer (2014)
13. Zhang, Y., Steele, A., Blanton, M.: PICCO: a general-purpose compiler for private distributed computation. In: Sadeghi, A-R., Gligor V.D., Yung, M. (eds.), *CCS 2013*, pp. 813–826. ACM, 2013.
14. Bogdanov, D., Niitsoo, M., Toft, T., Willemson, J.: High-performance secure multi-party computation for data mining applications. *International Journal of Information Security*, pp. 403–418 (2012)
15. Damgård, I., Pastro, V., Smart, N.P., Zakarias, S.: Multiparty computation from somewhat homomorphic encryption. In: Safavi-Naini, R., Canetti, R. (eds.), *CRYPTO*, vol. 7417 of *LNCS*, pp. 643–662. Springer (2012)
16. Pruulmann-Vengerfeldt, P., Kamm, L., Talviste, R., Laud, P., Bogdanov, D.: Capability Model, March 2012. UaESMC Deliverable 1.1.
17. Planul, J., Mitchell, J.C: Oblivious program execution and path-sensitive non-interference. In: *CSF 2013*, pp. 66–80. IEEE (2013)
18. Kennedy, W.S., Kolesnikov, V., Wilfong, G.: Overlaying circuit clauses for secure computation. Cryptology ePrint Archive, Report 2016/685 (2016). `http://eprint.iacr.org/2016/685`.
19. Damgård, I., Geisler, M., Krøigaard, M., Nielsen, J.B.: Asynchronous multiparty computation: Theory and implementation. In: Jarecki, S., Tsudik, G. (eds.), *PKC 2009*, vol. 5443 of *LNCS*, pp. 160–179. Springer (2009)
20. Schrijver, A.: *Theory of Linear and Integer Programming*. Wiley Series in Discrete Mathematics & Optimization. John Wiley & Sons (1998)
21. Zahur, S., Evans, D.: Obliv-c: A language for extensible data-oblivious computation. Cryptology ePrint Archive, Report 2015/1153 (2015). `http://eprint.iacr.org/2015/1153`.
22. Rastogi, A., Hammer, M.A., Hicks, M.W.: Wysteria: A programming language for generic, mixed-mode multiparty computations. In: *SP 2014*, pp. 655–670, IEEE Computer Society (2014)
23. Liu, C., Huang, Y., Shi, E., Katz, J., Hicks, M.W.: Automating efficient ram-model secure computation. In: *SP 2014*, pp. 623–638, IEEE Computer Society (2014)
24. GLPK. GNU Linear Programming Kit. `http://www.gnu.org/software/glpk`.
25. Wielemaker, J., Schrijvers, T., Triska, M., Lager, T.: SWI-Prolog. *Theory and Practice of Logic Programming*, pp. 67–96 (2012)
26. Kamm, L., Willemson, J.: Secure floating point arithmetic and private satellite collision analysis. *International Journal of Information Security*, pp. 1–18 (2014)
27. Liu, N-H., Chiang, C-Y., Hsu, H-M.: Improving driver alertness through music selection using a mobile eeg to detect brainwaves. *Sensors*, pp. 8199–8221 (2013)
28. Bogdanov, D., Kamm, L., Laur, S., Sokk, V.: Rmind: a tool for cryptographically secure statistical analysis. Cryptology ePrint Archive, Report 2014/512 (2014). `http://eprint.iacr.org/2014/512`.
29. Krips, T., Willemson, J.: Hybrid model of fixed and floating point numbers in secure multiparty computations. In: *ISC 2014*, pp. 179–197. Springer (2014)
30. Laud, P., Pankova, A.: Optimizing Secure Computation Programs with Private Conditionals (full version). Cryptology ePrint Archive, Report 2016/942 (2016). `http://eprint.iacr.org/2016/942`.