ABSTRACT

In theory secure computation offers a solution for privacy in many collaborative applications. However, in practice poor efficiency of the protocols prevents their use. Hand-crafted protocols are more efficient than those implemented in compilers, but they require significantly more development effort in programming and verification. Recently, Kerschbaum introduced an automatic compiler optimization technique for secure computations that can make compilers as efficient as hand-crafted protocols. This optimization relies on the structure of the secure computation program. The programmer has to implement the program in such a way, such that the optimization can yield the optimal performance. In this paper we present an algorithm that rewrites the program most notably its expressions – optimizing their efficiency in secure computation protocols. We give a heuristic for whole-program optimization and show the resulting performance gains using examples from the literature.

Categories and Subject Descriptors

D.3.4 [Programming Languages]: Processors—Optimization; D.4.6 [Operating Systems]: Security and Protection—Cryptographic controls

General Terms

Security, Programming Languages

Keywords

Secure Two-Party Computation, Programming, Optimization

1. INTRODUCTION

Secure (two-party) computation [39] allows two parties to compute a function \( f \) over their joint, private inputs \( x \) and \( y \), respectively. No party can infer anything about the other party’s input (e.g. \( y \)) except what can be inferred from one’s own input (e.g. \( x \)) and output (e.g. \( f(x, y) \)). Secure computation has many applications, e.g. in the financial sector, and has been successfully deployed in commercial and industrial settings [8, 9, 27].

Secure computation notoriously suffers from poor efficiency (compared to non-secure computations). Already in 1997 Goldwasser suggested manually optimized, specialized protocols for important problems [17]. We have seen a huge number of manually optimized protocols in the literature and there is growing adoption in industrial practice [8, 9, 27]. Therefore it is foreseeable that this approach will no longer scale.

A number of compilers, e.g. FairPlay [31], and programming environments [5, 7, 14, 19, 21, 38] exist that intend to remedy this problem. Nevertheless, these compilers suffer from worse performance than the manually optimized protocols. A recent automatic compiler optimization for secure computations [24] addresses this problem. Using this technique a compiler is capable of transforming a FairPlay program into a secure computation protocol that is (in many cases) as efficient as a manually optimized one.

However, this optimization technique relies on the structure of the program, such that the programmer has to adapt the program in order to achieve optimal performance. This can be a very complicated task, since the optimization might not be obvious to the programmer. The optimizer may therefore fail to yield an efficient protocol albeit optimization is feasible.

In this paper we consider restructuring the program, such that it is amenable to optimization. We rewrite the program’s expressions, such that the optimizer will output a more efficient protocol. We emphasize that the effectiveness of local rewriting can only be judged in the context of the global secure computation. For example, a program can be more efficient as a secure computation, but less efficient as a non-secure computation. Therefore incremental rewriting techniques no longer apply; such as common sub-expression elimination [11], where the effectiveness of each rewriting can be judged independently. Nevertheless, it is generally undecidable to find the optimal program. Hence, we apply a cost-driven heuristic and show its effectiveness using a number of examples drawn from the literature.

In summary, this paper’s contributions are

- Expression rewriting rules that yield more efficient secure computation protocols.
- A cost-driven heuristic to apply these rules, such that the resulting protocol will be at least as efficient as the protocol compiled from the initial program.
• An evaluation of examples drawn from the literature to demonstrate the effectiveness of the rewriting rules.

The remainder of this paper is structured as follows. First, we give a motivating example and problem description in Section 2. Then, we briefly describe the secure computation optimization from [24] in Section 3. We explain our rewriting rules in detail in Section 4, before presenting the heuristic applying them in Section 5. In Section 6 evaluate its effectiveness using several examples. We review related work in Section 7 and present our conclusions in 8.

2. PROBLEM STATEMENT

Consider the following secure computation: Alice and Bob, each have \( n \) data values \( a_i \) and \( b_i \) \((0 \leq i < n)\), respectively. They want to compute the mean \( \mu \) and variance \( \sigma^2 \) of the joint \( 2n \) values.

\[
\mu = \frac{1}{2n} \sum_{i=0}^{n-1} (a_i + b_i)
\]

\[
\sigma^2 = \frac{1}{2n-1} \sum_{i=0}^{n-1} ((a_i - \mu)^2 + (b_i - \mu)^2)
\]

In a single secure computation protocol, e.g. using Yao’s garbled circuit protocol [39], this protocol would have \( 2n \) inputs and 2 outputs. Using the FairPlay compiler [31], the above formulas can be straightforwardly implemented as in Listing 1. Note that the input size \( n \) is known to both parties.

Nevertheless, this secure computation protocol is not optimal. Using the ideas of [26] we can construct the following protocol. First, Alice and Bob each locally add their values to an intermediate sum \( a \) and \( b \), respectively. Since these intermediate sums can be inferred from one’s input and output of the statistics computation, they can simply exchange them and compute (and output) the mean locally as \( \mu = \frac{\sum_{i=0}^{n-1} (a_i + b_i)}{2n} \). This constitutes a protocol with identical security in the semi-honest model of secure computation. Similarly, since they now already know the mean, they can sum the square of the differences of their values locally to intermediate sums \( a’ \) and \( b’ \), respectively. They again exchange those intermediate sums and compute the variance as \( \sigma^2 = \frac{\sum_{i=0}^{n-1} ((a_i - \mu)^2 + (b_i - \mu)^2)}{2n-1} \).

We can implement this protocol in other languages for secure computation, e.g. L1 [38]. We omit depicting the code, since it is straightforward. This protocol is orders of magnitude faster than the FairPlay program. It does not even use one cryptographic operation, such as encryption or oblivious transfer. Furthermore, it reduces the complexity of the secure computation from \( O(n^2) \) (computing \( 2n \) squares for the variance) to \( O(1) \) (exchanging two values).

We compared our L1 implementation to an implementation of Yao’s protocol [20]. The results are depicted in Figure 1. Even against this best-in-class implementation, the efficiency gain of our optimized protocol is such that its performance bar (dark blue on the left) is barely visible in the figure.

1The protocol does not run, since multiplication and division have not yet been implemented in FairPlay. This has been fixed in FairPlayMP [5].
the program and since the programmer has not foreseen the necessary structure of localizing the sums, the optimizer is unable to detect it.

We therefore need to rewrite the program, such that the optimizer can be effective. Essentially, we have to untangle the loops, such that values from either Alice or Bob are first summed within their respective sets. Listing 3 in Section 6 displays such a rewritten program.

The optimizer would successfully deduce the efficient protocol from this FairPlay program. The research question of this paper is whether we can automatically rewrite the program from Listing 1 to the program from Listing 3. Furthermore, there are many more examples (see Section 6) which should be also covered. We therefore search for a generic technique that can be applied to many, if not all, secure computation protocols.

3. OPTIMIZATION

3.1 Labels

We only briefly describe the optimization algorithm of [24]. The basic idea is to infer program variables that can always be learnt from input and output. Each variable – whether input, output or intermediary – that is known to a party carries a label. If a variable is known to Alice, it carries the label A; if it is known to Alice and Bob, it carries the label AB.

The optimization of [24] infers labels of variables using epistemic modal logic. Consider the statement in Listing 2. If the variables b and c are known, then so is a. This is called forward inference. There is also backward inference. If the variables a and b are known, then so is c, but backward inference is tricky and depends on the operator. For a complete list of inference rules see [24].

3.2 Segmentation

Before applying the inference rules the optimizer transforms the program into an intermediate language. This intermediate language is a loop-free, single static assignment, three operand-code. If all three operands of a statement are known to at least one party, then this statement can be performed locally on that party and no longer needs to be computed in the secure computation protocol.

Using this localization of statements, the optimizer segments the program into possibly several secure computation protocols and local programs. Only the statements that actually need to be performed securely are implemented in a secure computation protocol. Particularly local segments at the beginning and at the end can be commonly exploited. In this paper, we attempt to enable this segmentation, even if the program description as such does not. We therefore extend the optimization technique of [24] by rewriting rules that transform the program into one more amenable to optimization.

3.3 Security

We assume the semi-honest or honest-but-curious model as defined by Goldreich [16]. The adversary in the semi-honest model is assumed to follow the protocol as specified, but may keep a record of the interaction and try to infer as much information as possible. A protocol secure in the semi-honest model ensures that such an adversary learns nothing except what can be inferred from his input and output. This guarantee is captured by Definition 1 of Goldreich using a simulator of the adversary view. The view VIEW\(^I\)(x, y) of a party during a protocol II on this party’s input x and the other party’s input y is its input x, the outcome of its coin tosses and the messages received during the execution of the protocol.

DEFINITION 1. We say a protocol II computing f(x, y) is secure in the semi-honest model, if for each party there exist a polynomial-time simulator S given the party’s input and output is computationally indistinguishable from the party’s view VIEW\(^II\)(x, y):

\[ S(x, f(x, y)) \approx VIEW^II(x, y) \]

Functions implemented in Yao’s garbled circuit protocol [39], e.g. using the FairPlay compiler [31], are always secure in the semi-honest model [30]. In this paper, our rewriting rules also assume the semi-honest model, but do not actually emit a protocol. This protocol is compiled by the optimizer of [24], such that if it preserves semi-honest security, then the optimizations in this paper preserve semi-honest security. All our transformations are performed on the intermediate code of the secure computation protocol. We therefore can omit any further security analysis.

4. REWRITING RULES

4.1 Abstract Syntax Tree

\[
\begin{align*}
\text{expr} &::= \text{bracket}\text{expon}\text{multi}\text{add}\text{compare}\text{simple} \\
\text{bracket} &::= (\text{expr}) \\
\text{expon} &::= \text{expr} \land \text{number} | 1/\text{number} \\
\text{multi} &::= \text{expr} \cdot | /\text{expr} \\
\text{add} &::= \text{expr} + | -\text{expr} \\
\text{compare} &::= \text{expr} < | <= | == | != | > | >= \text{expr} \\
\text{simple} &::= \text{var} - \text{var}\text{number} \\
\text{var} &::= a|b|\ldots \\
\text{number} &::= 0|1|2\ldots
\end{align*}
\]

Figure 2: Expression grammar

We consider a language for expressions with the grammar as in Figure 2. For simplicity of the exposition, we consider only one type of operand: fixed-length (e.g. 32 bit), signed fixed-point numbers. Fixed-point numbers are scaled integers where a fixed number of bits is reserved for decimal digits. They can be effectively used to implement division or roots and can be efficiently implemented in secure computation [10]. Each arithmetic operand implements the usual semantics on these numbers. Comparison operations return either 0 for false or 1 for true.
Instead of representing an expression in the same intermediate language as in [24], we represent it in an abstract syntax tree. The nodes of the tree are the operators (+, ·, ∧, etc.) and the leaves are the simple expressions (numbers and variables). The root of the tree is the last operator to be evaluated. We use the same operator precedence rules as standard programming languages, such as C or Java, indicated by the order of grammar expressions. We resolve bracket expressions into a position in this syntax tree in the usual way. The expression \((2 + (3 \times 4))\) has an abstract syntax tree with a root operator + whereas the expression \((2 + 3) \times 4\) has an abstract syntax tree with a root operator \(\times\).

The leaves are assigned labels regarding their origin, i.e. variables input by Alice are assigned label \(A\) and variables by Bob are assigned \(B\). Constants (numbers) are assigned label \(\text{AB}\), since they are known to both – Alice and Bob. We include the case of public input variables which are also known to Alice and Bob. Only intermediate values can be secret and carry no label.

Forward inference is straightforward. Since in our abstract syntax tree there are no intermediate values, we assign labels to the nodes. For forward inference a node is assigned the intersection of labels of its children.

We only perform a simplified version of backward inference compared to [24]. The rewritten expression is then analyzed in order to further optimize its performance. We named our backward inference result pruning. We assume that the final expression result is revealed as an output of the secure computation to both parties. Then each operation at the root of the tree with an operand known to both parties can be pruned. If we encounter such an operation, then we prune it from the tree.

At last, the optimization is performed. A label at a node means that this intermediate value is known to a party and may be computed locally at that party. Nodes with no labels need to be computed using a secure computation protocol.

4.2 Rule 1: Associative and Commutative Law

For an operator \(\oplus\) the associate law holds if and only if
\[
a \oplus (b \oplus c) = (a \oplus b) \oplus c
\]
The commutative law holds if and only if
\[
a \oplus b = b \oplus a
\]
Both laws hold for the operators + (addition) and · (multiplication). We can rewrite the operators − (subtraction) and / (division) for labeled, second operands into + and · respectively. In the expression \(a \oplus b\) the second operand is \(b\). An operand is labeled, if it carries one of the labels \(A\), \(B\) or \(\text{AB}\). We rewrite the operand with its inverse in the operation, i.e. \(-b\) or \(1/b\). Since inversion is an operation of the neutral element (a constant) which is always labeled \(\text{AB}\) and the second operand, the inversion operator is assigned the intersection, i.e. the label of the second operand. Therefore we rewrite a labeled operand with a labeled operation. Then we can rewrite the operator of the expression to either + or ·.

Our Rule 1 processes operators for which both - associative and commutative - laws hold in two steps: merging and sorting. In the merging step, we merge adjacent nodes in the syntax tree into a combined node with three (or more) operands. Due to the associative law we do not change the expression result.

![Figure 4: Application of associative and commutative laws](image)

Then, in the sorting step, we sort each child according to its label in the following order of labels \(\text{AB}, A, B\) and no label. Due to the commutative law we again do not change the expression’s result. Figure 4 shows an exemplary application of this rewriting rule.

![Figure 5: Output of merged and sorted multi-operand operators](image)

When we generate the secure computation protocol, e.g. the 3 operand code of [24] or the circuit description of [31], we create the operands in sequential order of the labels. This means, first public operands (label \(\text{AB}\)), second Alice’s operands (label \(A\)), and so on. Finally, we add the operators with different labels, again in order. This ensures that the maximum possible number of operators are assigned the labels of the operands. The only remaining operators in the secure computation are then the ones without label. All other operations are performed locally at one party. Figure 5 shows the generated expression (as an abstract syntax tree) of our example.

4.3 Distributive Law

For a pair \(\oplus, \odot\) of operators the distributive law holds if and only if
\[
(a \oplus b) \odot (a \oplus c) = a \odot (b \oplus c)
\]
The distributive law holds for the pairs of operators + or · and · or /, respectively.

4.3.1 Rule 2 and 3: Forward

The most straightforward way to apply the distributive law is to reduce the number of operations in case of common sub-expressions. We emphasis that this step does not supersede common sub-expression elimination (CSE) [11]. On the one hand, CSE identifies and optimizes common sub-expressions even in remote parts of the program or protocol whereas our rewriting rule only uses local operands. On the other hand, our rewriting rule actually changes the order of operator evaluation and thereby eliminates the need for temporary storage.

Let be sub-expressions with arbitrary labels. Let be an arbitrary – potentially empty – security label. Let indicate that there is an empty label and if a sub-expression has no label, then it may carry any label Let indicate that there is an empty label and if a sub-expression carries a label, it does not (noticeably) decrease performance either. It only changes the order of the operators – potentially allowing the application of further associate and commutative rewriting rules. We show an example of the combination of both rules in Figure 7.

4.3.2 Rule 4: Backward

A not so obvious way to apply the distributive law is backward. This may seem counter-intuitive at first, since it increases the number of operations. This is an example where the cost of a local computation increases, but due to the higher cost of a secure computation, the overall cost decreases. We therefore need to take precautions, that this optimization yields an improvement in efficiency. If we ensure that the “distributed” operand is labeled, then the number of operations to be performed as secure computations is at least likely to stay constant. Furthermore, if this operand is labeled as known to both parties or in combination with the rewriting rules for the associate and commutative laws, it may actually decrease the number of operations in the secure computation protocol.

Again, let be a non-empty label. Rule 4 rewrites the expression to

![Figure 6: Forward application of distributive law](image)

Note that at least one operator must not carry a label. It is then ensured that the number of operations in the secure computation decreases. Eleven more analogous rewriting rules exist for every position pair of the sub-expression and every combination of unlabeled operators in the initial expression. Figure 6 shows an exemplary transformation of the syntax tree of the first rewriting rule.

We can furthermore create common sub-expressions. This may be useful in combination with our other rewriting techniques, such as the rules for the associative and commutative law. Let be a non-empty label, such that and have at least one party in common, i.e. . Let be the inverse of sub-expression according to operator . Then using Rule 3 we rewrite the expression to

![Figure 7: Creation of common sub-expression](image)

In a second step we apply the above forward rewriting rule for the distributive law.

![Figure 8: Backward application of distributive law](image)

We show an example in Figure 8. As mentioned above, it is not guaranteed that this rewriting rule decreases the number of operations in the secure computation protocol, but it is possible. We show an example taken from the literature in Section 6.2. We also show how to apply the rewriting rules, such that a performance increase is guaranteed in Section 5. If this rewriting rule is effective, i.e. it decreases the number of operations in the secure computation, then both operators must get a label after the rewriting – potentially after applying further rewriting rules. Furthermore, if this is the case, then this rule will not be undone by the forward rewriting rule for the distributive law, since this rule requires at least one operator to be unlabeled.
4.4 Rule 5: Inverse Balancing

For comparison operators there are no algebraic laws that can be used for optimization. Nevertheless, there is one more rewriting rule. Let $\odot$ be a comparison operator, e.g. $\prec$. Let $\odot$ and $\ominus$ be inverse operators. Furthermore, let neither change the result of the comparison. In our language $+$ and $-$ always build such a pair, whereas $\cdot$ and $/$ do not for inequality operators, since multiplication with a negative number inverts the result of the comparison. Rule 5 rewrites the expression

$$e \odot - f \ominus - g$$

to

$$e \odot g \ominus f$$

Figure 9: Application of inverse balancing

We also maintain rewriting rules for the other operator $\odot$, the other operand $f$ and the other side of the comparison in the initial expression. Figure 9 shows an exemplary application of the above rule.

5. Algorithm

Even with such a simple language as arithmetic expressions we cannot hope to achieve optimal performance, since the program equivalency problem is recursively unsolvable [11]. We therefore apply a heuristic that results in secure computation protocols which are at least as efficient as the direct compilation of the initial expression. This heuristic considers all rewriting rules iteratively on an incrementally increasing program. Nevertheless, some rewriting rules may even result in less efficient protocols. We therefore need to carefully select the rules to apply. We guide our choice using a cost function for operators. This cost function ensures that the cost of the entire program constantly decreases and only rewriting rules that optimize the whole program (up to the current increment) are considered.

5.1 Cost Function

Predicting the performance of a secure computation protocol is a challenging task. An attempt for comparing homomorphic encryption and garbled circuits (Yao’s protocol) has been made in [37]. Nevertheless the main anticipated benefit of our optimization is to compute more operators locally, although in some cases we might also improve on the type of operators in the secure computation. Local compilation is an order of magnitude faster than any secure computation – whether using homomorphic encryption or garbled circuits. This has already been exploited in the manual optimization of [21]. We therefore conclude that the accuracy of our model for comparing operators in the secure computation protocol is of less importance.

We follow the argument of [37] that the performance of a garbled circuit secure computation protocol is linear in the number of gates of the operator. Let $\lambda$ be the bit length of an operand. Therefore we use the costs as in Table 1 for the operators.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Labeled</td>
<td>0</td>
</tr>
<tr>
<td>Unlabeled</td>
<td></td>
</tr>
<tr>
<td>Comparison</td>
<td>$\lambda$</td>
</tr>
<tr>
<td>Addition$^1$</td>
<td>$\lambda$</td>
</tr>
<tr>
<td>Multiplication$^1$</td>
<td>$\lambda^2$</td>
</tr>
<tr>
<td>Exponentiation</td>
<td>$\lambda^3$</td>
</tr>
</tbody>
</table>

Table 1: Operator costs

There are two distinct cost functions: one for operators $\text{cost}$ and one for trees $\text{tcost}$. The operator cost function $\text{cost}$ returns the costs as in Table 1. The tree cost function $\text{tcost}$ performs a preorder tree traversal and sums the cost of each encountered operator.

5.2 Rewriting

We perform the search for performance-improving rewriting rules using a reverse breadth-first search. The intuition is that in order to judge the effectiveness of a rewriting rule at the root of the tree each sub-tree must be optimal. We visit each sub-tree in order of incrementally increasing height. Then, we apply each of the five possible rewriting rules at the root and perform a subsequent optimization on each sub-tree. This repeated, subsequent optimization is necessary, since a change in the tree, e.g. a change of the order of operators, may have enabled additional rules in the sub-tree.

We emphasize that a rule that causes a cost-reduction cannot be undone by a subsequent rewriting rule. This is also true for the forward and backward rewriting rules for the distributive law due to their choice of labels on the operators. Algorithm 1 depicts the pseudocode for our heuristic. The function $\text{toptimize}$ performs the overall optimization whereas the function $\text{optimize}$ optimizes a sub-tree. We only keep results of rewriting rules that decrease cost. Therefore our overall heuristic deterministically does not increase cost, i.e. it either improves or maintains cost.

Let there be $n$ nodes in tree and let $H_n$ be the height of tree. Each node is visited at most $H_n$ times. Since $H_n = O(n)$, the worst-case complexity of our heuristic is $O(n^3)$. The expected height of an uniformly randomly chosen tree is $H_n = O(\log n)$ [15]. Therefore the expected average-case complexity of our heuristic is $O(n \log n)$.

6. Examples

On the one hand, we cannot expect to achieve optimal performance in all cases. On the other hand, our heuristic is not expected to decrease performance. We evaluate its effectiveness using a number of examples from the literature.

6.1 Statistics

Statistics computation is the motivating example from Section 2. We consider the program from Listing 1. We unroll the loops, since they have a fixed number of iterations, and build an abstract syntax tree of expressions for

$^1$For multi-operand operators we multiply by the number of unlabeled operators in the expanded tree.
Algorithm 1 Cost-driven heuristic

```plaintext
procedure OPTIMIZE(root)
    for 1 ≤ i ≤ HEIGHT(root) do
        for all t | HEIGHT(t) = i do
            OPTIMIZE(t)
        end for
    end for
end procedure

procedure OPTIMIZE(node)
    for all r ∈ Rules do
        if MATCHES(r, node) then
            new ← COPY(node)
            APPLY(r, new)
            for all c|c ∈ CHILDREN(new) do
                OPTIMIZE(c);
            end for
            if TCOST(new) < TCOST(node) then
                node ← new
            end if
        end if
    end for
end procedure
```

each output. Then we use our heuristic. First, the addition operators are merged and sorted into one multi-operand addition operator using the rewriting rules for the associative and commutative law (Rule 1). The sorting step leads to the main optimization possible. The division by 2n and 2n − 1, respectively, are pruned using result pruning. No more rewriting rules lead to cost reductions. The generated intermediate code corresponds to the intermediate code of the program in Listing 3. The optimizer can deduce the optimal protocol of exchanging only intermediate sums. Our compiler in this example is as efficient as the manually optimized protocol.

6.2 Joint Economic Lot Size

The joint economic lot size (JELs) is a two-party supply chain optimization following the model of [3]. Using a secure computation protocol has been proposed in [35] and demonstrated as a JavaScript implementation in [36]. We do not give the operations research background in this paper, but refer the reader to [35]. We extend the model by a safety stock s. Alice supplies inputs f_A, h_A and c. Bob supplies inputs f_B, h_B, s and d is a public input known to both. The following formula from [3] is computed securely and output

$$q = \sqrt{2 \cdot d \cdot f_A + f_B \cdot \frac{d \cdot h_A}{e} + h_B + s}.$$ 

We build the abstract syntax tree for the expression. The backward rewriting rule for the distributive law (Rule 4) is applied to the denominator. This immediately reduces the number of operations in the secure computation by a multiplication, since d is a public variable. This example underpins the necessity for the backward rewriting rule. The term d · f_A is labeled using the forward inference algorithm. The resulting expression in the secure computation is

$$q = \sqrt{ \frac{a + b}{a' + h_B} + h_B + s}$$

where \(a = 2 \cdot d \cdot f_A\), \(a' = d \cdot \frac{h_A}{e}\) and \(b = 2 \cdot d \cdot f_B\). A similar computation (without safety stock) has been proposed in [35, 36]. Again, our optimization algorithm is as efficient as the manually optimized protocol.

6.3 XML Transformation

Protocols applicable to XML transformation have been proposed in [23]. A type-safe and therefore provably secure implementation has been shown in [25]. These protocols implement basic string processing operations, such as concatenation, sub-string and find. The protocols are composed of several cryptographic techniques, such as secret sharing, homomorphic encryption, garbled circuits, and oblivious transfer.

The implementation of [25] contains a sub-step where a string length t is compared to a constant. The string length is additively secret shared, i.e. Alice has t_A and Bob has...
Our optimizer will immediately use the inverse balancing rule (Rule 5) to rewrite the expression to

\[ l_A < m - l_B \]

saving one secure addition. The manually optimized protocol description of [23] suggests this optimization as well. Nevertheless, due to type safety considerations it cannot be implemented in [25]. Our optimizer is able to remedy this and optimize the type-safe implementation. This example shows that different compiler techniques for security and performance can nicely interact to yield the best-performing, provably secure protocol.

7. RELATED WORK

This work is related to programming environments for secure computation [5, 7, 14, 19, 21, 31, 38], compiler language techniques for secure computation [24, 25, 33], and programming zero-knowledge proofs [1, 2, 32].

Programming environments for secure computation can be coarsely classified into those that implement their own compiler [5, 7, 19, 31] and those that build on top of existing programming languages [14, 21, 38]. The first programming environment – including its own compiler – for secure computation is FairPlay [31]. It implemented Yao’s two-party, garbled circuit protocol [39]. It has been later extended to multi-party computations in [5] based on the multi-party version of Yao’s protocol [4]. It introduced the SFDL programming language which provided an abstraction of the ideal functionality, i.e. the function computed by the secure protocol. Programs in SFDL are translated into a binary circuit which is interpreted using Yao’s protocol. Secure multi-party computations are also supported by ShareMind [7] based on the information-theoretically secure protocol of [6]. It has its own programming language SecureC [22] and compiler. All these single protocol based environments suffer from very poor performance of the compiled protocols. A first step towards improving performance by mixing protocols in one environment, such as Yao’s protocol and homomorphic encryption, was introduced in TASTY [19]. Nevertheless, they restricted themselves to one provably secure composition of [28] and do not perform program-based optimization.

Programming environments as extensions of existing, generic programming languages can implement any protocol, but usually provide built-in support for a few. The first such language is VIFF [14] based on the Python language. It is intended for the protocol of [12]. Similar extensions have been presented for the Java language [21, 38]. The framework of [21] is based on Yao’s protocol [39] and currently produces the most efficient protocols. A significant portion of its efficiency gain is based on optimizations by the programmer like manually localizing computations. The L1 language [38] again supports mixed protocol secure computations and outputs its protocols in Java.

The first programming language (domain-specific language) for secure computations was introduced in [33]. It contains a labeling concept similar to ours, but the labels are set by the programmer. We already introduced the first program-based optimization of [24]. It is the first proposal to use programming language techniques to enhance the performance of secure computations. Our work builds on its optimizing algorithm, but removes one of its major problems (for expressions): the dependence on the program structure. There are other optimizations of secure computations, such as the free XOR-technique [29]. This technique optimizes a specific secure computation protocol – Yao’s garbled circuits [39] – almost independent of the program. In [25] a type system for mixed protocol secure computation is introduced. It ensures that type-safe programs are protocols secure in the semi-honest model. It provides a security guarantee for the programmer, but does not perform any optimization.

Recently, first in [1] and independently in [32] and later in [2] the authors have proposed compilers for zero-knowledge proofs, which take as input a specification of the statement to be proved. This statement may contain expressions similar to the language considered in this paper and then it is compiled in to a zero-knowledge protocol. The cryptographic implementation of such protocol is either based on \( \Sigma \)-protocols [13] or a pairing-based scheme [18].

Note that optimization of expressions in non-secure programming languages has been considered decades ago, e.g. [11, 34].

8. CONCLUSIONS AND FUTURE WORK

In this paper we consider the automatic optimization of programs compiled into secure computation protocols. We build on the automatic optimization of [24] and present five rewriting rules that restructure expressions in order to yield more efficient protocols. We have successfully applied our optimization algorithm to multiple examples drawn from the literature.

Future work is to extend rewriting rules to conditional statements implementing branches. Furthermore, this work assumes the semi-honest model. Other models which are generally considered more secure, such as the malicious model, should be investigated.

9. REFERENCES


