Automatically Optimizing Secure Computation

Florian Kerschbaum
SAP Research
Karlsruhe, Germany
florian.kerschbaum@sap.com

ABSTRACT

On the one hand, compilers for secure computation protocols, such as FairPlay or FairPlayMP, have significantly simplified the development of such protocols. On the other hand, optimized protocols with high performance for special problems demand manual development and security verification. The question considered in this paper is: Can we construct a compiler that produces optimized protocols? We present an optimization technique based on logic inference about what is known from input and output. Using the example of median computation we can show that our program analysis and rewriting technique translates a FairPlay program into an equivalent – in functionality and security – program that corresponds to the protocol by Aggarwal et al. Nevertheless our technique is general and can be applied to optimize a wide variety of secure computation protocols.

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 (Multiparty) computation protocols [4, 7, 15, 31] allow a number of parties the computation of a function on joint input without disclosing anything except what can be inferred by one party's input and output. This offers an intriguing solution to many real-world problems where collaboration is prevented by the reluctance to disclose one's data. Privacy-preserving data mining [2, 21] is just one example of many.

Secure computation protocols are notoriously difficult to develop, since they require a high degree of domain expertise. Compilers, such as FairPlay [23] or FairPlayMP [6], have significantly simplified their development. They implement general techniques – the protocols of Yao [31] and Beaver et al. [4], respectively – in a cryptographic protocol layer and translate a program describing the ideal functionality into a circuit that is then interpreted by this layer. This reduces the development of the protocol to programming the ideal functionality.

Unfortunately, the compiler approach can be quite inefficient and is currently not amenable to large problem instances. Fortunately, a large body of work exists on optimized secure computation protocols for special problems. Besides the specification of the protocol itself, optimized protocols usually require a hand-crafted proof of security. These proofs are difficult and can be omitted in the compiler approach, since they are implicit in the cryptographic protocol layer.

Naturally, the question arises if a compiler could also produce optimized protocols. In programming languages the optimizing techniques of compilers have already overtaken the capabilities of even the most skilled assembler programmers. In this paper we present an optimization technique for a secure computation compiler. To the best of our knowledge this is the first such technique. Our optimizer takes as input a FairPlay program in the *Secure Function Definition Language* (SFDL). It outputs a Java program with several calls to FairPlay as a subroutine.

The basic idea is as follows: Everything that can be inferred from one's input and output is known. All known information can be revealed during the computation and does not have to be computed securely. Via logic inference we (under-) estimate the knowledge of each agent about a program's variables. Then we extract all operations on known data to the Java program of the local agent. This can significantly reduce the functionality that needs to be computed securely and the size of the circuit(s).

This paper contributes

- *logic inference rules* about an agent's knowledge in a SFDL program.
- a *program transformation* based on the result of the inference.
- an *evaluation* based on example applications of median and weighted average.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CCS'11, October 17–21, 2011, Chicago, Illinois, USA.
Copyright 2011 ACM 978-1-4503-0948-6/11/10 ...$10.00.

We present our optimization technique using a running example of median computation. In Section 2 we present this example in detail with program code for SFDL, optimized version as suggested by Aggarwal et al. [1] and performance comparison. Then, based on this example we show our inference rules and transformation technique in Section 3. We report our evaluation including the examples of oblivious transfer and weighted average in Section 4. Finally, we describe related work in Section 5 and conclude the paper in Section 6.

2. RUNNING EXAMPLE

As a running example in this paper we consider the joint computation of the median. We consider two parties – Alice and Bob – and each party has a set of n (distinct) integers. Alice and Bob want to jointly compute the median element of the union of their sets without disclosing anything else.

We implement the distributed algorithm by Aggarwal et al. [1]. Let each party's set be sorted in ascending order. The algorithm proceeds in several rounds. In each round each party chooses the median element of its set and then the parties compare these two elements. The party with the lower median disposes all elements below its median and the party with the higher median disposes all elements above its median. They begin a new round with these newly formed sets until each set has size 1 in which case they choose the lower one as the joint median.

We implemented this algorithm in SFDL of the FairPlay system [23]. This is not straightforward, since efficiently accessing the median of a variably sized array requires division which is not available in SFDL. This problem can be avoided by unrolling the loops and precomputing the division. In order not to complicate the exposition we choose $n = 2$. Furthermore, to simplify the analysis to be presented in this paper we replace arrays with structures of scalar variables. The resulting program is depicted in Listing 1.

Aggarwal et al. [1] not only present a distributed algorithm, but a distributed, secure protocol. The key insight of their protocol is that given the output of the joint median, the result of each comparison can be inferred. Then, in order to implement an efficient, secure algorithm it is only necessary to implement the comparisons using secure computation and all other operations can be implemented locally at the party's site. In Listing 1 the two relevant comparisons are depicted in lines 17 and 25.

The protocol by Aggarwal et al. can no longer be implemented as a single FairPlay program. Instead we implemented it using the L1 language and system for mixed-protocol secure computation [30]. The resulting program is depicted in Listing 2.

In L1 the same program is executed by both parties; except lines preceded by numbers are only executed by the parties with the respective identifiers. These identifiers are also returned by the call to the built-in function `id()`. The function call `comp32` in lines 9 and 17 invokes a secure computation of a comparison using the local input x, i.e. each party submits its variable x to a joint secure computation protocol.

The similarity between the two programs is not surprising, since they implement the same algorithm. Nevertheless, they compile into two very different protocols. While the FairPlay program implements Yao's protocol, the L1 program invokes secure computation only for the comparisons.

```
program Median {
type Elem           = Int<32>;
type AliceInput     = struct {Elem x,
                              Elem y};
type AliceOutput    = Elem;
type BobInput       = struct {Elem x,
                              Elem y};
type BobOutput      = Elem;
type Input   = struct {AliceInput   alice,
                        BobInput     bob};
type Output = struct {AliceOutput alice,
                        BobOutput    bob};

function Output out (Input in) {
   var Elem a, b;

   if (in.alice.x <= in.bob.x) {
      a = in.alice.y;
      b = in.bob.x;
   } else {
      a = in.alice.x;
      b = in.bob.y;
   }

   if (a <= b) {
      out.alice = a;
      out.bob   = a;
   } else {
      out.alice = b;
      out.bob   = b;
   }
}
}
```

Listing 1: Median Computation in FairPlay

```
include "utilYaoOTs.l1";
include "compareYao.l1";

int Median() {
   int x = loadInt("input_x" + id());
   int y = loadInt("input_y" + id());
   int a;

   if (comp32(x) == 1) {
      1: a = y;
      2: a = x
   } else {
      1: a = x;
      2: a = y;
   }

   if (comp32(a) == 1) {
      1: send(2, "a", a);
      2: a = readInt("a");
   } else {
      1: a = readInt("a");
      2: send(1, "a", a);
   }

   return a;
}

Median();
```

Listing 2: Median Computation in L1

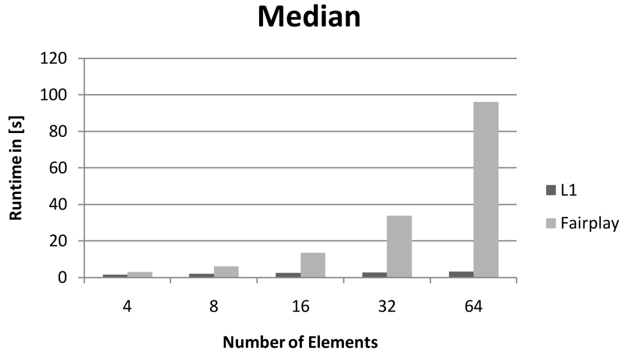

Figure 1: Secure Median

The result is a striking difference in performance. We ran experiments with both protocols for different joint set sizes $2n = 4, 8, 16, 32, 64$. The performance measurements are depicted in Figure 1. For only 64 elements in the joint set the L1 implementation already outperforms the FairPlay implementation by more than a factor of 30.

The research question of this paper is: Given the similarity between the two programs can we compile the FairPlay program into the protocol of the L1 program? Furthermore, can we design an optimization algorithm that is not only applicable to Aggarwal et al.'s median computation, but is generic?

3. TRANSFORMATION

3.1 Semi-Honest Security

We consider secure computation protocols secure in the semi-honest model [14]. Loosely speaking, an adversary in the semi-honest model adheres to the protocol, but may keep a record of the interaction and later try to infer additional information from it. A protocol secure in the semi-honest model, keeps everything about a party's input confidential that cannot be inferred from one's input and output.

Goldreich [14] defines security in the semi-honest model. The view $VIEW^{\Pi}(x, y)$ of a party during protocol Π 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 Π 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 $VIEW^{\Pi}(x, y)$:*

$$S(x, f(x, y)) \stackrel{c}{=} VIEW^{\Pi}(x, y)$$

FairPlay programs are secure in the semi-honest model by construction. L1 programs need to be proven secure manually, but can also implement more efficient protocols. Aggarwal et al.'s protocol is secure, if (correctly) implemented in L1.

Semi-honest security ensures confidentiality except what can be inferred from one's input and output. This inference is the basis of our optimization. We construct a program analysis technique for FairPlay programs that infers what is known from input and output. If our optimized protocol reveals this additional information – but nothing else –, then this does not violate semi-honest security, since it can be constructed by the simulator. We need to be very careful that our analysis is safe and always underestimates the possible inferences from input and output.

3.2 Algorithm

In our algorithm to transform a SFDL program we use labeling of variables. Each variable in the program is assigned any of two labels: $\{A\}$ or $\{B\}$. Labels are non-exclusive, i.e. a variable may be assigned none, one or two labels. The contents of a variable labeled $\{A\}$ is (always) known to Alice and the contents of a variable labeled $\{B\}$ is (always) known to Bob. A variable without labels is secret and can only be implemented using secure computation while a variable with both labels is public information and can be freely shared. The program text is known to both parties and so are all constants in the program. Our algorithm for optimizing a SFDL program proceeds as follows:

1. We convert the SFDL program into single-static assignment (SSA), 3-operand code form.
2. We use our inference algorithm to infer the labels of each variable.
3. We segment the program into different protocols. Each statement of which all variables are known to any party are executed locally by that party. Only statements with operands whose intersection of labels is empty will be executed using a secure computation as in FairPlay.

We give two examples of our segmentation of a program into protocols. Consider the addition expression in Listing 3. We have written the labels of the variables in subscript.

$$a_{A} = b_{A} + c_{A\ B};$$

Listing 3: Local Computation

This statement can be executed locally at Alice's site, since she knows all variables and the result `a` can be used as input in subsequent computations. Now, consider the comparison expression in Listing 4.

$$a_{A\ B} = b_{A} < c_{B};$$

Listing 4: Secure Computation

This statement must implemented as a secure computation. It is the program representation of Yao's millionaires' problem [31] where two parties compare private numbers.

3.3 Static Single Assignment

We convert the SFDL program into static single assignment (SSA) [9] form. In SSA each variable is assigned at most once and never changed afterwards. If a variable is changed in the original program, a new variable is introduced in SSA.

```
a = b ? c : d;
```

Listing 5: Conditional Assignment

```
program Median {

function Output out (Input in) {
    var Boolean a, d;
    var Elem     b, c;

    a = in.alice.x <= in.bob.x;
    b = a ? in.alice.y : in.alice.x;
    c = a ? in.bob.x   : in.bob.y;

    d = b <= c;
    out.alice = d ? b : c;
    out.bob   = d ? b : c;
}
}
```

Listing 6: Single Static Assignment Form

We also unroll all loops – in SFDL loops have a constant number of iterations –, inline all functions – in SFDL there is no recursion –, and transform all if statements to conditional assignments. A conditional assignment has the form of Listing 5 where the variable `a` is assigned either the value of `c` or `d` depending on the truth value of the condition `b`.

Furthermore, we resolve complex expressions and transform all expressions into 3-operand code (except conditional assignments which have four operands). Each statement takes two operands (and an operator) as input and assigns the result (a new) operand as output.

The result of our SSA transform is depicted in Listing 6. We have omitted the type declarations for brevity as they are the same as in Listing 1. The challenge for our inference is now to determine that both Boolean variables, `a` and `d`, are (always) known to both, Alice and Bob, if they know their respective input and output.

3.4 Inference

Our inference algorithm (under-)estimates the knowledge about variables given one's input and output. In the notion of information flow, we try to determine whether there is an *inevitable* flow of information from the input to that variable or from that variable to the output. We differ from language-based information flow [29] which tries to detect and prevent *possible* information flows. One particular challenge in our inference is therefore that the flow of information may occur differently as possible flows depending on the state of program, but does occur inevitably in all possible states. We address this challenge by using a more powerful logic.

Furthermore, all (sensible) secure computations involve a declassification (release) of private information. Assuming that information flow is admissible, our inference algorithm indicates the earliest possible declassification operation in order to later increase the performance of the protocol. We present the details in Section 3.5.

3.4.1 Epistemic Modal Logic

We use epistemic modal logic (EML) [16] to reason about the knowledge of the protocol participants. Let p be a proposition which can be either true (p) or false ($\neg p$). We consider a single agent (either Alice or Bob) and use the modal operator K to denominate its knowledge. When we write

$$Kp$$

we mean that the agent knows p. EML only considers truthful knowledge, i.e.

$$Kp \Rightarrow p$$

Although the agent's knowledge is certain, the agent might consider different *possible worlds*. A possible world is an interpretation of propositions assigning them a truth value. One of the possible worlds is the real world, but the agent cannot differentiate between them. For example, let $\mathcal{W}_1$ and $\mathcal{W}_2$ be two different, possible worlds. A proposition p might be true in world $\mathcal{W}_1$, but false in world $\mathcal{W}_2$. We write

$$\mathcal{W}_1 \vDash p$$
$$\mathcal{W}_2 \vDash \neg p$$

As already mentioned we consider one agent at a time and using the possible worlds we can define the meaning of knowledge. Let there be n possible worlds $\mathcal{W}_1, \dots, \mathcal{W}_n$. If and only if the proposition p is true in all possible worlds, then the agent knows p. We write

$$\mathcal{W}_j \vDash Kp \Leftrightarrow \forall (1 \leq i \leq n) \mathcal{W}_i \vDash p$$

This implements one of the simplest Kripke structures [20]. Introducing "logical and" for complex propositions ϕ and ψ we can summarize our simplified EML variant as

$\mathcal{W} \vDash p$	if p is assigned true in $\mathcal{W}$
$\mathcal{W} \vDash \neg\phi$	if $\mathcal{W} \nvDash \phi$
$\mathcal{W} \vDash \phi \wedge \psi$	if $\mathcal{W} \vDash \phi$ and $\mathcal{W} \vDash \psi$
$\mathcal{W} \vDash K\phi$	if for all $\mathcal{W}_i \vDash \phi$

3.4.2 Traces

In the remainder of this section we will consider wlog Alice as the agent. We create possible worlds by tracing the program. Each possible world contains one trace of the program. The trace is a set of propositions where each proposition is generated by a possible assignment in the program.

For integers we follow the assumption by Aggarwal et al. and assume that each input is unique. This assumption is not essential in our inference and can be easily lifted by adapting the algorithm to generate the traces. No further changes are necessary, but Aggarwal et al.'s protocol is then insecure and our running example no longer works. We therefore trace each integer variable in Alice's input as a unique identifier from the set $\{A1, \dots, An\}$. We trace each integer variable in Bob's input as the identifier B. Boolean variables may be traced as either $true$ or $false$.

```
a = in.alice.x + in.bob.y
```

Listing 7: Integer Assignment

An assignment creates a new entry in the trace. An assigned integer variable is added to the trace with the intersection of the identifiers of its operands (with $\perp$ marking the empty set). In Listing 7 we add to the trace the proposition

$$\mathcal{W} \vDash a = \perp$$

```
a = in.alice.x < in.bob.y
```

Listing 8: Boolean Assignment

Each Boolean assignment with integer operands doubles the number of possible worlds. For each existing world $\mathcal{W}$

$$\mathcal{W} \models a = false$$
$$\mathcal{W} \models b = A1$$
$$\mathcal{W} \models c = B$$
$$\mathcal{W} \models d = false$$
$$\mathcal{W} \models out.alice = B$$

Figure 2: Trace

we create a new world $\mathcal{W}'$. In the example of Listing 8 we add to the trace of world $\mathcal{W}$ the proposition

$$\mathcal{W} \models a = true$$

In world $\mathcal{W}'$ we copy all of the previous trace (except, of course, the trace of `a`) of the world $\mathcal{W}$ and then add the proposition

$$\mathcal{W}' \models a = false$$

For Boolean assignments with Boolean operands we optimize this to a single proposition in all possible worlds by evaluating the expression.

Conditional assignments as in Listing 5 interpret the truth assignment of the condition. In worlds with the proposition $b = true$ we add a proposition for `a` with the trace of `c` and in worlds with the proposition $b = false$ we add a proposition for `a` with the trace of `d`.

In our example of the median computation (Listing 6) we get four possible worlds: one for each trace of `a` and `d`. We show one of the four traces in Figure 2.

Each variable is a proposition. The proposition is true in a possible world, if the contents of a variable is known to Alice in this world, i.e. under the assumption of a specific trace. If a variable `a` is known to Alice in all worlds, i.e. proposition a is true, then then the proposition Ka is true and we add the label $\{A\}$ to the variable.

It is important to note that we consider all possible traces. This may include invalid traces of unreachable code. Since we perform static analysis, we must either over- or underestimate the reachable code. By considering all unreachable code our analysis is safe. An additional, invalid trace can only reduce the set of labels of a variable, since even in the invalid trace the variable must be known to Alice in order to result in a label $\{A\}$ for the variable.

3.4.3 Rules

Initially only input and output are known to Alice (in all worlds). Further knowledge needs to be inferred via rules. We present these rules in this section.

Rules are logical deductions of propositions. Our rules are true in all worlds, although they might make a reference to a trace proposition. Let ϕ, ψ and χ be propositions. We write

$$\phi \wedge \psi \Rightarrow \chi$$

or

$$\frac{\phi \quad \psi}{\chi}$$

as a shortcut for

$$\neg((\phi \wedge \psi) \wedge \neg\chi)$$

We use the SSA form of the program and for each statement we add several rules. As we try to infer additional knowledge about program variables from input and output, we can distinguish two types of rules. Forward rules try to infer knowledge about the result of a statement from knowledge about its input. Backward rules try to infer knowledge about the input of a statement from knowledge about its result and maybe partial input. Forward and backward rules are true and applied concurrently during the logical inference. Only their combination enables us to make the necessary inferences. We have rules for expressions (3 operand code), conditional assignments, symbolic computation and previous knowledge.

Expressions.

```
a = b + c
d = e * f
```

Listing 9: Expression Assignment

The forward rule for expressions is simple. If both operands are known then, then the assigned variable is known. For the example in Listing 9 we write

$$\frac{b \quad c}{a} \qquad \frac{e \quad f}{d}$$

The backward rules for expressions are already more complex. Whether there is a backward rule depends on the operator. For addition (+) and subtraction (−) there are backward rules.

$$\frac{a \quad c}{b} \qquad \frac{a \quad b}{c}$$

But for multiplication ($*$) and (integer) division (/) there are none. This is for safety reasons. While in (modular) addition and subtraction it is always possible to compute the inverse, this is not always the case for multiplication and division. Consider multiplication by 0: the result is 0 independent of the value of the other operand.

Conditional Assignment.

The forward rule for conditional assignments is similar. Nevertheless we distinguish between the two possible worlds by making a reference to the trace proposition. Consider the example in Listing 5: we construct the following forward rules

$$\frac{b \quad b = true \quad c}{a} \qquad \frac{b \quad b = false \quad d}{a}$$

For conditional assignments there are two different backward rules. One for the assigned operand and one for the condition. The backward rules for the operands are as follows.

$$\frac{b = true \quad a}{c} \qquad \frac{b = false \quad a}{d}$$

We emphasize that these rules do not require knowledge of the condition. This is safe (as we will show at the example of oblivious transfer in Section 4.1), since the conclusion is only true in half of the possible worlds and therefore no further inference on labels is feasible.

The backward rule for the condition is our most complex rule. For its understanding we need to consider a specific world (trace). For example, consider the trace of Figure 2 and the conditional assignment in line 12 of the corresponding Listing 6. It is the final output assignment for Alice of the median computation.

Our intuition is the following: If the agent Alice knows the assigned variable ($out.alice$) and the trace of the variable ($out.alice = B$) only occurs in possible worlds with a specific condition ($d = false$), then the agent knows the condition. In other words: Alice can infer the condition from observing the known and unique output for that condition. This rule combines our tracing, backward rules and EML. We can express the uniqueness of the trace as Alice's knowledge

$$K(out.alice = B \Rightarrow d = false)$$

Recall that the modal operator K ranges over all possible worlds. The entire rule for the example is as follows

$$\frac{out.alice \quad out.alice = B \quad K(out.alice = B \Rightarrow d = false)}{d}$$

In the median example this means that Alice knows that in the last comparison her value was larger, because the output is a value of Bob. Recall that she can distinguish her and Bob's values, because all inputs are distinct.

Symbolic Computation.

In addition to the evaluation of statements we need to perform some symbolic computation. We can infer additional knowledge about the program's state by manipulating its symbols. Particularly, we introduce propositions for the lower-than relation into our inference.

Let `a` and `b` be two variables. Then the (true) proposition lt_a_b means that (in this possible world)

$$lt_a_b \Leftrightarrow a < b$$

```
c = z ? y : x;
a = b <= c;
d = e ? f : c;
```

Listing 10: Comparison Example

First, we need to tie the symbolic computation to the program. Consider the example in Listing 10. We deduce a proposition lt_c_b, if $a = false$ is in the trace and the variable `a` is known to the agent Alice.

$$\frac{a \quad a = false}{lt_c_b}$$

We also deduce knowledge about program variables from symbolic knowledge. We can omit the trace proposition, since our deduction of symbolic knowledge is always correct and safe.

$$\frac{lt_c_b}{a}$$

We need to take care of conditional assignments and track variable renaming. Variable tracking works, again, backward and forward. In the example of Listing 10 we write

$$\frac{e = false \quad lt_c_b}{lt_d_b} \qquad \frac{z = false \quad lt_c_b}{lt_x_b}$$

There are many such rules, one for each direction, condition truth value, front or back position of the variable in the proposition and integer variable that is not `c`.

Second, we implement mathematical axioms, such as negation and transitivity. Since all inputs are distinct, the lower-than relation can be inverted for negation.

$$\frac{\neg lt_a_b}{lt_b_a}$$

The lower-than relation is also transitive.

$$\frac{lt_a_b \quad lt_b_c}{lt_a_c}$$

Unfortunately, propositions in EML are simple, such that when using symbolic computation on n variables the number of rules increases to $O(n^3)$ with only transitivity already. Also, each conditional assignment introduces $O(n)$ variable renaming rules. We must therefore take care to restrict symbolic computation to the minimum set of variables necessary. We use symbolic computation only for variables appearing as operands in comparison statements.

Nevertheless, in our running example of median computation symbolic computation is necessary. The agent Alice needs to infer that in the world of the trace in Figure 2 the condition $in.alice.x > in.bob.x$ in the first comparison is true, since $in.alice.x > in.bob.y$ from the second comparison and $in.bob.y > in.bob.x$ in all of its inputs.

Assumptions.

Of course, we need to start out with the propositions that input and output are known.

$$K in.alice.x$$
$$K in.alice.y$$
$$K out.alice$$

But as seen in the previous example we need to also encode the assumptions about the input. Following the assumption of Aggarwal et al. we start out with the propositions

$$K lt_in.alice.x_in.alice.y$$
$$K lt_in.bob.x_in.bob.y$$

Note that these assumptions could also be encoded in the program. If the program would start with a sorting routine of the inputs, then the subsequent inference would remain. The sorting routines would also be marked as local by our optimization technique, since they operate on local input only. They would then be executed at the local site's before the secure computation. This shows another advantage of our technique[1].

3.4.4 Deduction

We can now enter all possible worlds (traces) and rules into a theorem prover. If it can deduce knowledge about the variables, we can assign the labels. Let ϕ be the propositions for all possible worlds and ψ all rules derived from the program. We then enter the following theorem for each variable, e.g. `a`

$$\phi \wedge \psi \Rightarrow Ka$$

If the theorem can be proven, we assign the label $\{A\}$ to the variable `a`. We create one such theorem for each variable. If the theorem cannot be proven, we do not assign a label.

We then repeat the process with Bob as the agent. The derivation algorithm for the rules and traces remains unchanged, but some propositions in the rules made in Section 3.4.3 "Assumptions" change, since now Bob's input and output are known instead of Alice's. Also, the propositions in the traces change, since Bob can distinguish its integer input variables, but not Alice's. We also create one theorem for each variable for Bob.

[1]We have omitted this option in order to stay truthful to the example of Aggarwal et al.

```
program Median {

function Output out (Input in) {
    var Boolean a A B, d A B;
    var Elem    b A, c B;

    a A B = in.alice.x A <= in.bob.x B;
    b A=a A B? in.alice.y A: in.alice.x A;
    c B=a A B? in.bob.x B  : in.bob.y   B;

    d A B = b A <= c B;
    out.alice A = d A B ? b A: c B;
    out.bob   B = d A B ? b A: c B;
}
}
```

Listing 11: Assigned Labels

Finally, we assign the labels. The result of the median example is depicted in Listing 11. As we have pointed out in the beginning of the paper, the challenge is to infer the labels on the variables `a` and `d`. We show a complete deduction of Ka for Alice in Appendix A.

3.5 Segmentation

We can now segment the program of the median computation and perform selected computations at the local sites. The first statement in line 7 needs to be performed as a secure computation, since the intersection of labels is empty. Nevertheless, the result of the secure computation can be made available to both, Alice and Bob. This computation is an instance of Yao's millionaires' problem. Line 8 can be performed solely by Alice and line 9 solely by Bob. Line 11 is a secure computation again.

For lines 12 and 13 we need to implement a special routine. Although the intersection of labels is empty, Alice (or Bob) needs to learn the output of the computation. Therefore for output variables assigned in conditional statements, in case the operand is only known by the other party, we need to implement a send and receive message operation. Care must be taken and this routine only applies, if the condition variable is public and known to both, Alice and Bob, and not in our further example of oblivious transfer where the condition is only known to Alice and which must be implemented as a secure computation. The transmission operations are also implemented locally at Alice's and Bob's site, respectively.

If you compare this segmentation with the L1 program of Listing 2 the program transform is completely successful (except some syntactical constructs). We encode our segmented program as a Java program that calls FairPlay as a subroutine for its secure computation. We have transformed a (relatively) inefficient FairPlay program using program analysis into an efficient protocol similar to the hand-crafted one by Aggarwal et al.

4. FURTHER EXAMPLES

In this section we consider further examples in order to validate our optimization technique. First, we consider 1-out-of-2 oblivious transfer [11] and show that our inference is safe, i.e. the optimized protocol still implements secure oblivious transfer. Second, we use our optimization to reduce joint lot size computation [3] to weighted average computation.

```
program Median {
type Elem        = Int<32>;
type AliceInput  = Boolean;
type AliceOutput = Elem;
type BobInput    = struct {Elem x,
                           Elem y};
type Input  = struct {AliceInput  alice,
                      BobInput    bob};
type Output = struct {AliceOutput alice};

function Output out (Input in) {
out.alice = in.alice ? in.bob.x : in.bob.y;
}
}
```

Listing 12: Oblivious Transfer

4.1 Oblivious Transfer

Oblivious transfer (OT) can be implemented with the SFDL program shown in Listing 12. Of course, it is not best practice to implement OT using SFDL, since FairPlay uses an optimized OT protocol to implement Yao's protocol. It is much more efficient to use such an optimized protocol directly. We can nevertheless test if our optimization technique is safe.

OT is implemented as a single conditional assignment without intermediate variables. Therefore only traces and the rules for conditional assignments apply. There are four possible worlds – two for Alice and two for Bob – depending on the condition `in.alice`.

First, in Alice's case the backward rule for the operand applies. In one world she learns `in.bob.x` and in the other `in.bob.y`. Then also the forward rule applies, but she already knows `out.alice`. Alice cannot distinguish Bob's input, therefore the backward rule for the condition does not apply. Since Alice only learns either of Bob's inputs in a single world, she does not gain knowledge in all worlds and no new labels are created.

Second, in Bob's case no rule applies. Neither backward rule applies, since he does not know `out.alice`. The forward rule also does not apply, since he does not know the condition `in.alice`.

Therefore no optimization is possible using our inference algorithm. This is good news, since OT is an elementary cryptographic primitive and any "optimization" at the language layer would show that our inference algorithm is unsafe. We show next that a slight deviation in just one rule could already lead to such unsafe inferences.

Assume we would change the forward rule by omitting the proposition *in.alice* of the condition, as we have done in the backward rule, resulting in the rules below

$$\frac{in.alice = true \quad in.bob.x}{out.alice} \qquad \frac{in.alice = false \quad in.bob.y}{out.alice}$$

Then, Bob would learn `out.alice` in both worlds. Furthermore, the backward rule for the condition would apply and Bob would also learn `in.alice`. A complete breakdown of the safety of the optimization. It is therefore necessary to include the condition in the forward rule, but not the backward rule for the operand.

```
program Median {
type Elem          = Int<32>;
type AliceInput    = struct {Elem d,
                              Elem fV,
                              Elem hV,
                              Elem c};
type AliceOutput = Elem;
type BobInput      = struct {Elem d,
                              Elem fB,
                              Elem hB};
type BobOutput     = Elem;
type Input  = struct {AliceInput  alice,
                      BobInput    bob};
type Output = struct {AliceOutput alice,
                      BobOutput   bob};

function Output out (Input in) {
   var Elem a, b, c, d, e, f, g, h, i;

   a = 2 * in.alice.d;
   b = a * in.alice.fV;
   c = in.alice.d / in.alice.c;
   d = c * in.alice.hV;

   e = 2 * in.bob.d;
   f = e * in.bob.fB;

   g = f + b;
   h = in.bob.hB + d;
   i = g / h;

   out.alice = i ** 2;
   out.bob   = i ** 2;
}
}
```

Listing 13: Joint Economic Lot Size

4.2 Weighted Average

We now consider an example from business administration. When placing an order buyer and vendor need to agree on an order quantity or lot size. Both, buyer and vendor, know the (yearly) demand d of the buyer as part of their contract, but they also have private information. The buyer has holding costs h_B per item and fixed ordering costs f_B per order. The vendor has holding costs h_V per item, fixed setup costs f_V per order and capacity c. They can jointly compute the optimal lot size q using Banerjee's formula [3] below

$$q = \sqrt{\frac{2d(f_B + f_V)}{h_B + \frac{d}{c}h_V}}$$

Since information such as costs and capacities is very sensitive, it is sensible to implement the computation securely. Buyer and vendor can do so using the (extended) SFDL program in Listing 13 where Alice is the vendor and Bob the buyer. We extend SFDL with multiplication ($*$) and integer division (/) which are parsed, but not implemented in FairPlay. This has been fixed in FairPlayMP. Furthermore, we introduce the operator ($**$) for integer roots which cannot be parsed by FairPlay. Therefore we need to execute the resulting protocols using the L1 system.

The program is entirely constructed from assignment expressions. Therefore only its forward and backward rules apply. Recall that constants are public and known to both, Alice and Bob.

As a result of our analysis intermediate variables `a`, `b`, `c`, `d` and `i` are known to Alice. Variables `e`, `f` and `i` are known to Bob. Therefore the first segment of the program (lines 20 - 23) can be executed locally at Alice's site and the second segment (lines 25 and 26) can be executed locally at Bob's site. Only the middle segment (lines 28 - 30) need to be executed as a secure computation. The final segments can be executed locally again, at either party's site.

Thus, we have optimized the protocol significantly. Instead of securely computing a square root – amongst other arithmetic operations –, we have reduced the problem to securely computing a weighted average. This significantly increases the efficiency of the protocol by locally computing several expensive operations and shows another successful application of our optimization technique. All our optimizations (including segmentation) were performed automatically.

We stress that we prepared the SFDL program, such that the optimizer yields the best result. We structured the arithmetic in order to form groups of operations that can be optimized into local computations. We nevertheless anticipate that this optimization can also be performed automatically in the future. Techniques, such as term rewriting, offer excellent capabilities for such transforms.

5. RELATED WORK

There are several systems for implementing secure computation. We classify them into systems specifying the ideal functionality and systems specifying the protocol description. Furthermore, we look at the examples used in our paper.

The positive effect on performance of performing as many computations as possible locally has been confirmed by [18]. [26] also tries to deduct local computations, but only considers the parties' input and not also their output as we do.

5.1 Ideal Functionality Specification

FairPlay [23] provides a generic system for secure two-party computation. The FairPlay system comprises a procedural language, called *Secure Function Definition Language* (SDFL), a compiler that translates SDFL programs into one-pass Boolean circuits that can be securely evaluated using the protocol suggested by [31]. Originally, i.e., as presented in [23], the FairPlay system supported only secure two-party computations written in SDFL 1. This system served as a basis for the development of the FairPlayMP system [6]. FairPlayMP supports the secure evaluation of multi-party computations written in SFDL 2. Some circuit optimization techniques for FairPlay are presented in [28].

There are other compilers [12, 32] which produce safe code for a program according to an information flow policy in a distributed environment. Nevertheless, they do not allow to specify any secure computation and offer no secure implementation for the resulting information flow.

Just FairPlay and FairPlayMP are instances of systems which only describe the ideal functionality of a secure computation, i.e. *what* is to be implemented by the protocol. The burden of finding the ideal protocol is with the compiler. To the best of our knowledge we present the first technique that automatically optimizes such a program.

5.2 Protocol Description Specification

Next to FairPlay and its specification of the ideal functionality, there are several secure computation implementations where the programmer can give the compiler hints on how to optimize. In the Secure Multiparty Computation Language (SMCL) [25] the programmer specifies the ideal functionality, but may specify also the visibility of variables as secret, private or public. This is similar to our (party-specific) labels, but we infer the labels automatically via program analysis and do not burden the programmer. The TASTY compiler and its TASTYL language [17] allow implementing mixed-protocol implementations using not only Yao's protocol, but also (additively) homomorphic encryption. The programmer can specify in TASTYL which protocol is to be used for which operation. This results in significantly more efficient, yet always provably secure protocols. The VIFF framework [10] extends the Python language using a library. It offers the programmer several primitives for secure computation, but the protocol description is up to the programmer. The Sharemind framework [8] offers an interpreter for an assembler-like language implementing secure protocols. The programmer has to specify the protocol description using the primitives. The L1 system [30] allows specifying protocol descriptions with many primitives including Yao's protocol and homomorphic encryption. It also offers a library for network communication.

All of these languages, compilers or frameworks are instances of systems where the programmer can – at least partially – specify *how* the protocol is implemented. This approach leads to significantly more efficient protocols, but puts an additional burden on the programmer. In this paper we have chosen a different approach. We use program analysis to infer the optimal protocol from the ideal functionality, but, of course, our techniques are also applicable in combination with other, programmer specified optimizations. To the best of our knowledge we are the first to present a fully automatic technique.

5.3 Examples

We use several optimized protocols as our examples. Our running example of the median computation by Aggarwal et al. [1] was one of the first instances of a (hand-)optimized secure computation protocol. Our optimization technique was able to even slightly exceed the manual optimization.

Oblivious transfer (OT) is a long known primitive in cryptography. 1-out-of-2 OT was introduced by Even et al. [11]. Our optimization was not able to improve its programming language specification. On the one hand, this is good, because our optimization is safe. After all, Kilian has shown that cryptography can be founded on OT [19]. On the other hand, it shows that our optimization technique cannot optimize the cryptographic implementation. This is not surprising, since it only consider the ideal functionality. There are several (cryptographically) optimized protocols for OT, e.g. [13, 22, 24].

Our last example is joint lot size computation as weighted average. This has been introduced in [27]. It shows the manifold applications of secure computation. Our optimization technique was able to match the manual optimization, if the program was accordingly structured.

6. CONCLUSIONS

We have presented the first technique to automatically optimize a secure computation program. Our program analysis infers what a party knows from input and output. This additional information is then used to perform several computations locally instead of as a secure protocol. We have shown using the running example of median computation that our optimization technique can match (and even exceed) those performed manually by the programmer. Using our implementation we were able to compile and execute an SFDL program as Aggarwal et al.'s protocol.

As the performers of an automated analysis related to Aggarwal et al.'s protocol we would like to make a few comments on this protocol. First, our analysis underpins that the protocol is *secure*. Our analysis is crafted carefully to be safe and resulted in a slightly more revealing protocol than the manually designed one. Second, our analysis shows that protocol is very slightly *inefficient*. Our analysis concludes – and it is very easy to verify manually – that the last comparison can also be implemented as a Yao's millionaire's protocol. Aggarwal et al. recommend to use a minimum computation that outputs the minimum element. This is very slightly less efficient, but avoids sending the output by one party. Our compiled protocol is therefore slightly more efficient.

For future work we see two open problems. First, our analysis works on protocols secure in the semi-honest model. We would like to extend this to protocols secure in the malicious model. One can adapt the compiler from the semi-honest to the malicious model by Goldreich [14], but this could be too inefficient. Second, our prototypical implementation uses the ModLeanTab theorem prover [5]. We can perform the compilation of the median example in roughly one minute, but the majority of the time is spent on proving the theorems. A more efficient algorithm that scales to large programs and maybe avoids theorem proving is desirable.

7. ACKNOWLEDGEMENTS

We are grateful to Axel Schröpfer for implementing the L1 examples and performing the experiments and to Achim Brucker and Marcel Jünemann for implementing SSA and the transformation.

8. REFERENCES

[1] G. Aggarwal, N. Mishra, and B. Pinkas. Secure computation of the k-th ranked element. In *EUROCRYPT'04: Advances in Cryptology*, 2004.

[2] R. Agrawal and R. Srikant. Privacy-preserving data mining. *ACM SIGMOD Record*, 29(2), 2000.

[3] A. Banerjee. A joint economic-lot-size model for buyer and supplier. *Decision Sciences*, 17, 1986.

[4] D. Beaver, S. Micali, and P. Rogaway. The round complexity of secure protocols. In *STOC'90: Proceedings of the 22nd ACM Symposium on Theory of Computing*, 1990.

[5] B. Beckert and R. Gore. In *CADE'98: Proceedings of the International Conference on Automated Deduction*, 1998.

[6] A. Ben-David, N. Nisan, and B. Pinkas. Fairplaymp: a system for secure multi-party computation. In *CCS'08: Proceedings of the 15th ACM Conference on Computer and Communications Security*, 2008.

[7] M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *STOC'88: Proceedings of the 20th ACM Symposium on Theory of Computing*, 1988.

[8] D. Bogdanov, S. Laur, and J. Willemson. Sharemind: a framework for fast privacy-preserving computations. In *ESORICS'08: Proceedings of the 13th European Symposium on Research in Computer Security*, 2008.

[9] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions Programming Languages and Systems*, 13(4), 1991.

[10] I. Damgård, M. Geisler, M. Krøigaard, and J. B. Nielsen. Asynchronous multiparty computation: theory and implementation. In *PKC'09: Proceedings of the 12th International Conference on Practice and Theory in Public Key Cryptography*, 2009.

[11] S. Even, O. Goldreich, and A. Lempel. A randomized protocol for signing contracts. *Communications of the ACM*, 28(8), 1985.

[12] C. Fournet, G. L. Guernic, and T. Rezk. A security-preserving compiler for distributed programs: from information-flow policies to cryptographic mechanisms. In *CCS'09: Proceedings of the 16th ACM Conference on Computer and Communications Security*, 2009.

[13] C. Gentry and Z. Ramzan. Single-database private information retrieval with constant communication rate. In *ICALP'05: Proceedings of the 32nd International Colloquium on Automata, Languages and Programming*, 2005.

[14] O. Goldreich. *Foundations of Cryptography*, volume 2. Cambridge University Press, 2004.

[15] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. In *STOC'87: Proceedings of the 19th ACM Symposium on Theory of Computing*, 1987.

[16] J. Y. Halpern. Reasoning about knowledge: a survey. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 4. Oxford University Press, 1995.

[17] W. Henecka, S. Kögl, A.-R. Sadeghi, T. Schneider, and I. Wehrenberg. Tasty: tool for automating secure two-party computations. In *CCS'10: Proceedings of the 17th ACM Conference on Computer and Communications Security*, 2010.

[18] Y. Huang, D. Evans, J. Katz, and L. Malka. Faster secure two-party computation using garbled circuits. In *Proceedings of the 20th USENIX Security Symposium*, 2011.

[19] J. Kilian. Founding crytpography on oblivious transfer. In *STOC'88: Proceedings of the 20th ACM Symposium on Theory of Computing*, 1988.

[20] S. Kripke. A semantic analysis of modal logic i: normal modal propositional calculi. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 9, 1963.

[21] Y. Lindell and B. Pinkas. Privacy-preserving data mining. In *CRYPTO'00: Advances in Cryptology*, 2000.

[22] H. Lipmaa. An oblivious transfer protocol with log-squared communication. In *ISC'05: Proceedings of the 8th International Conference on Information Security*, 2005.

[23] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay—a secure two-party computation system. In *Proceedings of the 13th USENIX Security Symposium*, 2004.

[24] M. Naor and B. Pinkas. Efficient oblivious transfer protocols. In *SODA'01: Proceedings of the 12th ACM-SIAM Symposium on Discrete Algorithms*, 2001.

[25] J. D. Nielsen and M. I. Schwartzbach. A domain-specific programming language for secure multiparty computation. In *PLAS'07: Proceedings of the ACM Workshop on Programming Languages and Analysis for Security*, 2007.

[26] A. Paus, A.-R. Sadeghi, and T. Schneider. Practical secure evaluation of semi-private functions. In *ACNS'09: Proceedings of the 7th International Conference on Applied Cryptography and Network Security*, 2009.

[27] R. Pibernik, Y. Zhang, F. Kerschbaum, and A. Schröpfer. Secure collaborative supply chain planning and inverse optimization - the jels model. *European Journal of Operational Research*, 208(1), 2011.

[28] B. Pinkas, T. Schneider, N. P. Smart, and S. C. Williams. Secure two-party computation is practical. In *ASIACRYPT'09: Advances in Cryptology*, 2009.

[29] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), 2003.

[30] A. Schröpfer, F. Kerschbaum, and G. Müller. L1 – an intermediate language for mixed-protocol secure computation. In *COMPSAC'11: Proceedings of the 35th IEEE Computer Software and Applications Conference*, 2011.

[31] A. C.-C. Yao. How to generate and exchange secrets. In *FOCS'86: Proceedings of the 27th IEEE Symposium on Foundations of Computer Science*, 1986.

[32] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers. Untrusted hosts and confidentiality: Secure program partitioning. In *SOSP'01: Proceedings of the 18th ACM Symposium on Operating System Principles*, 2001.

APPENDIX

A. MEDIAN EXAMPLE

Assumptions:

$$\begin{array}{l} K in.alice.x \\ K in.alice.y \\ K out.alice \\ K lt_in.alice.x_in.alice.y \\ K lt_in.bob.x_in.bob.y \end{array}$$

Trace of world $\mathcal{W}_1$:

$$\begin{array}{l} \mathcal{W}_1 \models a = true \\ \mathcal{W}_1 \models b = A2 \\ \mathcal{W}_1 \models c = B \\ \mathcal{W}_1 \models d = true \\ \mathcal{W}_1 \models out.alice = A2 \end{array}$$

Inference in world $\mathcal{W}_1$:

$$\frac{\dfrac{d = true \quad out.alice}{b} \qquad b = A2 \quad K(b = A2 \Rightarrow a = true)}{\mathcal{W}_1 \models a}$$

Trace of world $\mathcal{W}_2$:

$$\begin{array}{l} \mathcal{W}_2 \models a = false \\ \mathcal{W}_2 \models b = A1 \\ \mathcal{W}_2 \models c = B \\ \mathcal{W}_2 \models d = true \\ \mathcal{W}_2 \models out.alice = A1 \end{array}$$

Inference in world $\mathcal{W}_2$:

$$\frac{\dfrac{d = true \quad out.alice}{b} \qquad b = A1 \quad K(b = A1 \Rightarrow a = false)}{\mathcal{W}_2 \models a}$$

Trace of world $\mathcal{W}_3$:

$$\begin{array}{l} \mathcal{W}_3 \models a = true \\ \mathcal{W}_3 \models b = A2 \\ \mathcal{W}_3 \models c = B \\ \mathcal{W}_3 \models d = false \\ \mathcal{W}_3 \models out.alice = B \end{array}$$

Inference in world $\mathcal{W}_3$:

$$\frac{in.alice.x \qquad \dfrac{a = true \qquad \dfrac{d = false \quad out.alice}{c}}{in.bob.x}}{\mathcal{W}_3 \models a}$$

Trace of world $\mathcal{W}_4$:

$$\begin{array}{l} \mathcal{W}_4 \models a = false \\ \mathcal{W}_4 \models b = A1 \\ \mathcal{W}_4 \models c = B \\ \mathcal{W}_4 \models d = false \\ \mathcal{W}_4 \models out.alice = B \end{array}$$

Inference in world $\mathcal{W}_4$:

$$\frac{\dfrac{lt_in.bob.x_in.bob.y \qquad \dfrac{a = false \qquad \dfrac{a = false \qquad \dfrac{\dfrac{out.alice \quad out.alice = B \quad K(out.alice = B \Rightarrow d = false)}{d} \qquad d = false}{lt_c_b}}{lt_c_in.alice.x}}{lt_in.bob.y_in.alice.x}}{lt_in.bob.x_in.alice.x}}{\mathcal{W}_4 \models a}$$

Global inference:

$$\frac{\mathcal{W}_1 \models a \quad \mathcal{W}_2 \models a \quad \mathcal{W}_3 \models a \quad \mathcal{W}_4 \models a}{K a}$$