ABSTRACT

Secure computation has high computational resource requirements during run-time. Secure computation optimization can lower these requirements, but has high computational resource requirements during compile-time. This prevents automatic optimization of most larger secure computations. In this paper we present an efficient optimization algorithm that does no longer require the use of a theorem prover. For a secure computation with \( m \) statements of which \( n \) are branching statements we lower the complexity from \( O(2^{n}m^{5}) \) to \( O(m^{5}2^{n}) \). Using an implementation of our algorithm we can extend automatic optimization to further examples such as the AES key schedule.

Categories and Subject Descriptors
D.4.6 [Operating Systems]: Security and Protection—Cryptographic controls; D.3.4 [Programming Languages]: Processors—Compilers

Keywords
Secure Two-Party Computation, Optimization, Programming, Domain-Specific Language, Theorem Proving

1. INTRODUCTION

Secure (two-party) computation [37] 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 [7, 8, 28].

Secure computation notoriously suffers from poor efficiency (compared to non-secure computations). Already in 1997 Goldwasser suggested manually optimized, specialized protocols for important problems [15]. A recent optimization algorithm for secure computations [23] automates this approach. Using this algorithm a compiler is capable of transforming a FairPlay [31] program into a secure computation protocol that is (in many cases) as efficient as a manually optimized one.

The key insight of this optimization algorithm is that information known from the input and output can be computed locally at a party’s site instead of securely. The algorithm infers this information and separates the program into local and secure parts. This inference is made using a theorem prover [4] for epistemic modal logic [16]. Later, the programming languages community picked up the approach and proved it sound and (partially) complete [34]. Let a secure computation have \( m \) (simple operation) statements of which \( n \) are Boolean assignments (branches), then the run-time of the theorem prover is \( O(2^{n}m^{5}) \) for one of \( m \) variables. In this paper we improve the efficiency of the optimization, but do not modify its output (and therefore soundness and completeness still hold).

The current run-time clearly prevents the inference on larger programs and the use of the algorithm in real-world compilers. In [23] the author presents the example of Aggarwal et al.’s median protocol with 2 elements at each party. This example ran in our implementation in roughly 58 minutes. We calculated that running the example with 4 elements at each party exceeds our lifetime. Using the implementation of the algorithm in this paper we ran the example with 4 elements in roughly 4 seconds. Furthermore, we automatically performed the previously manual optimization of the AES key schedule from [19] in 295 seconds.

This paper contributes an optimization algorithm for secure computation that

- performs the same optimization of secure computation as [23],
- extends that optimization to arrays,
- reduces the complexity from \( O(2^{n}m^{5}) \) to \( O(m^{5}2^{n}) \).

This algorithm completely eliminates the need for a theorem prover and instead uses its own data structures. Hence, it can reduce the complexity from exponential to polynomial (in the number of non-branching statements).

The remainder of the paper is structured as follows: In the next section we review related work. In Section 3 we give the necessary background to understand our algorithm. We briefly describe the optimization algorithm of [23]. Then, in Section 4 we show our novel, efficient optimization algorithm and present its application to three examples in Section 5.
2. RELATED WORK

Performance optimization of secure computation is subject to research for a long time. Initially most protocols were manually crafted and optimization was performed by skilled researchers. Goldwasser suggested this direction of research already in 1997 [15].

Manual optimization of secure computation can be difficult. A researcher might try to optimize the wrong parameters of a protocol. For example, [2] and [12] suggested manually optimized protocols for private sequence comparisons and set intersection, respectively, and were later superseded by clever implementations of generic protocols [22, 18].

With the prevalence of commercial applications of secure computation, such as [7, 8, 9, 26, 28], the necessity for rapid prototyping and testing for performance of secure computation arose. First, FairPlay [31] introduced a compiler for secure two-party computations. It has been later extended to multi-party computations in FairPlayMP [5] based on the protocol of [3]. FairPlay takes a specification of the ideal functionality of a secure computation as input and produces a garbled circuit for Yao’s protocol [37]. It frees the programmer from verifying security as in manual protocol development, since Yao’s protocol is provably secure for any functionality [30]. Nevertheless, the performance of the resulting protocols was still dissatisfying.

There are a number of further frameworks which allow the programmer to specify the secure computation protocol in more detail [6, 11, 36]. In all of these the programmer has to care about the security of the protocol. This approach is obviously error-prone.

Given the compiler approach of FairPlay there are still a number of approaches to improve performance. First, one can improve the implementation of Yao’s protocol. There are different protocol techniques, such as the Free XOR-technique [29] and extensions of OT [21], that improve the performance of Yao’s protocol. These have been integrated into a recent implementation [19]. It also uses pipelining to better parallelize the evaluation of the garbled circuit. This implementation achieved the best performance for a number of secure computations, such as private set intersection [18] and biometric identification [20].

Second, one can choose between different protocols for secure computation. Next to Yao’s protocol, there is also (partially) homomorphic encryption, e.g., Paillier’s encryption scheme [32], and Goldreich’s protocol [14]. They have different performance characteristics for different secure operations as shown by [35]. The framework of [17] first enabled the programmer to choose between Yao’s protocol and homomorphic encryption. It achieved this while preserving guaranteed security. In [38] it was shown that also Goldreich’s protocol can be competitive thereby further increasing the options for the protocol designer. All of these protocols can be integrated securely using a type system as shown by [25]. This choice can be quite difficult for the programmer and [27] introduced a compiler technique that automatically chooses the best protocol.

Third, as in [27] the compiler can perform the optimization. This has first been introduced in [23]. The algorithm presented in this paper significantly improves the (compile-time) performance of the algorithm proposed in [23] for optimization. This optimization technique identifies parts of the computation that do not need to be performed securely. Rastogi et al. proved it sound and (partially) complete [34]. Nevertheless, it depends on the specification of the program. Automatic restructuring of the program has been introduced in [24]. It rewrites arithmetic expressions, such that they yield better performance in a secure computation protocol.

3. BACKGROUND

In this section we briefly explain the optimization technique from [23]. We will not cover the inferences made for all different statements. The reader is referred to [23] for the details. Wlog we only consider the knowledge of one party – Alice – in the remainder of this paper. Bob’s knowledge can be computed analogously.

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. This model covers many real-life threats such as attacks by honest but curious insiders.

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^{HI}(x, y) of a party during 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^{HI}(x, y):

\[ S(x, f(x, y)) \approx VIEW^{HI}(x, y) \]

All FairPlay programs are secure in the semi-honest model by construction. Also all protocols inferred by our optimization algorithm are secure in the semi-honest model. This includes Aggarwal et al.’s median algorithm and the optimized, joint computation of AES.

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 are careful that our analysis is safe and always underestimates the possible inferences from input and output.

3.2 Intermediate Language

Our optimization algorithm takes as input a FairPlay program in the Secure Function Definition Language (SFDL). We first convert the SFDL program into an intermediate language. We convert to static single assignment (SSA) [10] form. In SSA each program variable is assigned at most once and never changed afterwards. If a program variable is
changed in the original program, a new program variable is introduced in SSA. Hence, there are as many program variables – namely $m$ – as there are statements in the program.

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 $a = b \ ? \ c : d$ where the program 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 program variables (and an operator) as input and assigns the result (a new) program variable as output.

### 3.3 Tracing

We create all possible traces of the program. A statement assigning an integer program variable using an expression creates a new entry in the trace. Such an entry consists of the program variable name, e.g. $a$, and the input source, e.g. $A$ (for Alice), which can also be empty. To the contrary, each Boolean program variable assignment creates a branch and thereby doubles the number of possible traces. It also creates an entry with the program variable name and the Boolean value, either $true$ or $false$. At the end of the tracing algorithm we have $2^n$ traces with $m$ entries each.

In epistemic modal logic there is the concept of a possible world. 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. Using the possible worlds we can define the meaning of knowledge. Let there be $2^n$ possible worlds $W_1, \ldots, W_{2^n}$. If and only if the proposition $p$ is true in all possible worlds, then the agent knows $p$. In this case we write $Kp$ using the modal operator $K$.

In [23] a one-to-one correspondence between possible worlds and traces is established. Inference is then performed over all possible worlds (traces).

### 3.4 Inference

Inference is performed using propositional variables – not to be confused with program variables. There are three types of propositional variables:

- **Knowledge variables** store whether a program variable is known to Alice. There is one knowledge variable for each of the $m$ program variables and for each of the $2^n$ traces.
- **Trace variables** store the trace of a program variable. In epistemic modal logic there is one trace variable for each of the $m$ program variables, for each of the $2^n$ traces and for each input source ($A$, $B$, $\perp$, etc.) or truth value. In our algorithm we store the input source in the trace variable, since there can be at most one source for each variable due to the SSA.
- **Relation variables** store the quantity relation between program variables, e.g. greater than. In epistemic modal logic there is one trace variable for each of the $\binom{m(m-1)}{2}$ ordered pairs of program variables, for each of the $2^n$ worlds and for each relation. In our algorithm we store the relation in the relation variable. We only store the strictest relation. Note that $a < b$ or $a = b$ both imply $a \leq b$. Similar implications hold for greater-than relations. For example, if we store the relation $a < b$ and then access the relation variable for $a \leq b$, we obtain also the result true from the access method. Furthermore, if we store the relation $a = b$ and set the relation $a \leq b$ to true, no update of the data structure is performed.

Inferences are made using rules formulated as implications. If the left-hand side of a statement is true (in one trace), the right-hand side is set to true (in that same trace). On the left-hand side there is a conjunction of at most four of any type of propositional variable. On the right-hand side there is either one knowledge variable or one relation variable.

An example is the following simple statement

$$a = b + c$$

This statement implies three rules: one for forward inference and two for backward inference. Let $a$, $b$ and $c$ be the knowledge variables for the program variables $a$, $b$ and $c$, respectively. The forward inference rule is

$$b \land c \rightarrow a$$

This means if Alice knows both inputs, then she also knows the output. The backward inference rules similarly infer one input from the other input and output, i.e., they infer input from output. For all inference rules the reader is referred to [23].

All forward and backward inference rules are valid concurrently and for all traces. Note that inferences may nevertheless proceed differently in different traces due to the trace variables.

The propositional variables can also be augmented with the modal operator $K$. This implies that the variable must be true in all traces. The modal operator appears only on the left-hand side except for a final inference step for each program variable. At the end of the optimization we check whether a knowledge variable is true in all traces and then mark this variable as known to the party Alice.

There are at most four rules per statement depending on the statement type, i.e., we have to consider $O(m)$ rules. For an overview of all rules we refer the interested reader to [23].

At the beginning of the inference all knowledge variables for input and output (of Alice) are set to true. Furthermore, all trace variables from the tracing algorithm (see Section 3.3) are set to true in the corresponding traces. For Aggarwal et al.’s median protocol it is further necessary to set the relation variables for the assumed input relations to true. Then, the inference algorithm can take place. In [23] this executes the theorem prover. In Section 4 we will replace this with our algorithm.

 Afterwards, we can segment the program and perform selected computations at the local sites. Each statement of which all program variables are known to either Alice or Bob are executed locally by Alice or Bob, respectively. Only statements with program variables not entirely known to one party will be executed using a secure computation as in FairPlay.

In [23] it has been shown that this algorithm can infer Aggarwal et al.’s median protocol from the respective FairPlay program. In this paper we extend this to an inference of the AES key schedule as done manually in [19].
4. ALGORITHM

4.1 Data Structures

We store each type of propositional variable in its own data structure. Knowledge variables are stored in an array of lists. For each program variable there is an array element – a list. This list contains the traces for which the knowledge variable is true. Consequently this data structure has an overall size of \(O(2^m)\) with an array size \(O(m)\) and list sizes \(O(2^n)\) each. We commonly access this data structure to read or write the (truth) value of an individual variable. We have at most \(O(m^22^n)\) read accesses during the interpretation of the rules. Hence, the sequential \(O(2^n)\) read access time does not increase overall complexity.

Trace variables are stored in a list of arrays. For each trace there is a list entry containing an array for each program variable. The array stores the input source (e.g. \(A, B, \bot\), etc.) for integer variables or truth value for Boolean variables of the variable in this trace. This data structure also has an overall size of \(O(m2^n)\) with list size \(O(2^n)\) and array sizes \(O(m)\) each. Our most common operation on this data structure is to compute a set of traces that contain a specific input source or truth value. We can therefore traverse the list sequentially.

Relation variables are stored in a two-dimensional array of lists. Similar to knowledge variables, the list contains the entries for traces if available. Relation variables store the quantity relation between two program variables. Therefore the array contains an entry for each pair. We store only the strictest relation. Less strict relations are inferred in the algorithm. We can reduce the array size by only storing only relations of integer program variables. If necessary, relations of Boolean variables can be computed from the values stored in their trace variables, since the assignment of each Boolean variable in each trace is known. This data structure has an overall size of \(O(m2^n)\) with an array size \(O(m)\) and list sizes \(O(2^n)\) each.

4.1.1 Array Handling

The algorithm in [23] does not handle arrays, but FairPlay allows using array program variables. We distinguish two types of access: static indexing and variable indexing. In static indexing the array index is a constant. In variable indexing the array index is another program variable.

We treat each array element as its own program variable, i.e., we introduce propositional variables for each array element. This works well for reading and writing using static indexing. The new variables can be used in the same way as variables for scalar program variables.

For variable indexing there are essentially two design alternatives: First, we can trace each integer variable with all possible values and reduce array access to the array element. Unfortunately this significantly increases the number of traces and quickly becomes intractable. We therefore chose the following second alternative.

When reading an array element using variable indexing we use the intersection of all propositional variables of the array. For example, when reading a knowledge variable for \(a[i]\) we use the knowledge variables of all array elements of \(a\). Only, if all are true, then the knowledge variable for \(a[i]\) is true. If the intersection is empty, we default to the safe value, i.e., false for knowledge variables, empty for trace variables and unknown for relation variables. When writing an array element using variable indexing we compute the intersection for all propositional variables of all array elements. This implies that for knowledge variables writing using variable indexing has no impact, since knowledge variables are only set to true. Only in trace and relation variables we may reduce Alice’s knowledge. For example, when writing the input source \(B\) to a trace variable for \(a[i]\) we set the trace variable of all array elements of \(a[j]\) to the intersection of their previous value and \(B\).

For reading we greedily compute the intersection when updating the value of any array element. We store the intersection in its own propositional value, e.g. \(a\). We apply this technique in our analysis of AES (see Section 5.3).

4.2 Inference

<table>
<thead>
<tr>
<th>Algorithm 1 Inference Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> abstract syntax tree of intermediate language (ast), list of input variables (input), list of output variables (output)</td>
</tr>
<tr>
<td><strong>Output:</strong> List of known program variables (in all traces)</td>
</tr>
<tr>
<td><strong>function</strong> Infer((ast, input, output))</td>
</tr>
<tr>
<td>(symbol \leftarrow \text{BuildSymbolTable}(ast))</td>
</tr>
<tr>
<td>(know \leftarrow \text{CreateKnowledge}(symbol))</td>
</tr>
<tr>
<td>(trace \leftarrow \text{GenerateTraces}(ast, symbol, input))</td>
</tr>
<tr>
<td>5: (relation \leftarrow \text{CreateRelation}(symbol))</td>
</tr>
<tr>
<td>(\text{InitKnowledge}(input, output, trace.all))</td>
</tr>
<tr>
<td>(\text{rules} \leftarrow \text{GenerateRules}(ast))</td>
</tr>
<tr>
<td>repeat</td>
</tr>
<tr>
<td>(flag \leftarrow true)</td>
</tr>
<tr>
<td>10: for all rule (\in) rules do</td>
</tr>
<tr>
<td>(validset \leftarrow trace.all)</td>
</tr>
<tr>
<td>for all cond (\in) rule.lhs do</td>
</tr>
<tr>
<td>(condset \leftarrow \text{GetTraces}(\text{cond}, know, trace, relation))</td>
</tr>
<tr>
<td>if (\text{cond.hasModalOperator}) then</td>
</tr>
<tr>
<td>(\text{if condset} \neq) trace.all then</td>
</tr>
<tr>
<td>(condset \leftarrow \emptyset)</td>
</tr>
<tr>
<td>end if</td>
</tr>
<tr>
<td>end if</td>
</tr>
<tr>
<td>validset (\leftarrow) validset (\cap) condset</td>
</tr>
<tr>
<td>20: end for</td>
</tr>
<tr>
<td>if (\text{SetVar}(know, relation, rule.rhs, validset)) then</td>
</tr>
<tr>
<td>(\text{flag} \leftarrow false)</td>
</tr>
<tr>
<td>end if</td>
</tr>
<tr>
<td>end for</td>
</tr>
<tr>
<td>25: (\text{ComputeTransitivity}(relation))</td>
</tr>
<tr>
<td>until flag</td>
</tr>
<tr>
<td>return (\text{ComputeKnown}(know, trace.all))</td>
</tr>
<tr>
<td>end function</td>
</tr>
</tbody>
</table>

The pseudocode of the main loop of our algorithm is depicted in Algorithm 1. It shows our main function Infer. Infer takes as input the root of the abstract syntax tree of the intermediate language described in Section 3.2, a list of the input variables and a list of the output variables. The function BuildSymbolTable returns a symbol table stored in hash table symbol computed by traversing the abstract syntax tree. Then we initialize our data structures of Section 4.1. The function CreateKnowledge creates a one-dimensional array of lists for the knowledge variables stored in know. The array has an entry for each entry in
the symbol table (with arrays expanded as in Section 4.1.1). The lists are initially empty indicating that no knowledge variable is set. The function GenerateTraces generates a list of traces each with an array of the trace variables stored in trace. It traverses the abstract syntax tree recording the source of each assignment of an integer variable in the corresponding array entry starting with the list of input variables input. When encountering an assignment of a Boolean variable, the number of traces is doubled and one trace for each possible assignment (TRUE, FALSE) is generated. The function returns a completed list of arrays. Sometimes it is necessary to access a set of all traces, e.g. for checking whether a variable is known. We accomplish this using the object variable trace.all. The function CreateRelation creates a two-dimensional array of lists for the relation variables stored in relation. Similarly to know, the lists are initially empty indicating that all relations are unknown. We then set the initial knowledge variables of input and output. The function InitKnowledge sets the lists to all traces for each input or output variable. The definition of semi-honest security (see Section 3.1) justifies this initial knowledge. For this, it needs the set of all traces trace.all. This completes the initialization of the data structures.

In line 7 the function GenerateRules then traverses the abstract syntax tree to generate the inference rules. The rules are stored in the list rules. Recall that there at most 4 rules per statement. These fixed rules are appended to the list for each statement. For a description of the rules the reader is referred to [23]. Rules are logical implications and have a fixed format. They have a left-hand side (object variable lhs) consisting of a list of conditions. Each condition can be any type of propositional variable – knowledge, trace or relation. Conditions are combined in a conjunction, i.e., all conditions must be true for the implication. Conditions may optionally carry the modal operator indicated by the Boolean object variable hasModalOperator. Recall that there are at most 4 conditions per rule. The right-hand side of a condition is a single propositional variable of either knowledge or relation type.

After initialization the main functions of inference are performed in a loop. Lines 8 to 26 encompass this main loop. The main loop is controlled by a flag variable flag. At each iteration this variable is initially set (line 9). It is eventually cleared (line 22) in case any knowledge or relation variable has been changed, i.e., the loop continues as long as new knowledge is generated. The function SetVar (called in line 21) returns true if any new propositional variable is set. Once no new propositional variable is set the loop exits. Note that propositional variables are only set and never cleared.

In each iteration of the main loop we process each inference rule. This loop runs from line 10 to 24. In this loop we process each rule iteratively. First, we analyze each condition (lines 12 to 20). For each condition we get all traces where it is true using the function GetTraces (line 13). We iteratively compute the intersection of the sets of all conditions (line 19). If a condition carries the modal operator K, it is either valid in all traces or none. Consequently, we set the set of traces to empty, if it does not encompass all traces (lines 14 to 18).

The intersection of traces where all conditions are true is stored in the variable validset. We then add these traces to the propositional variables for the right-hand side of the rule.

This is accomplished in function SetVar (line 21). Recall that SetVar returns true, if and only if a new propositional variable has been set.

After processing all rules we need to update transitive relations. Assume Alice knows a < b and just learned b < c in this iteration of the main loop. Clearly this implies a < c, but this is not yet stored in the relation variables. We compute the transitive closure of the relation variables in function ComputeTransitivity using the Floyd-Warshall algorithm [13] for each trace. Note that this computation is performed in the main loop, but only once for all rules. Also note that inference of new transitive relations is only possible, if new relation variables have been set by the rules. Therefore, we do not need to update exit flag for the main loop.

Once the main loop has exited, all possible (but safe) knowledge variables are set. We can then compute the set of variable known in all traces, i.e., those that can be computed locally by Alice. The function ComputeKnown compares the list of set traces for each program variable to all traces. If they are the same, then the program variable is included in the return set. This completes our inference algorithm.

4.3 Complexity Analysis

4.3.1 Time Complexity

We first analyze the number of iterations of the loops. The inference algorithm (see Algorithm 1) consists of three nested loops. The main loop (line 8 - 26) iterates while updates are performed to the propositional variables. There are \(O(m^2n^2)\) knowledge variables and \(O(m^2n^2)\) relation variables. Initially none are set and during each iteration at least one is set (none are cleared). Therefore the main loop has at most \(O(m^2n^2n^2)\) iterations.

The loop over the rules (line 10 to 24) has at most \(O(m)\) iterations, since there are at most 4 rules per statement, i.e., at most \(O(m)\) rules. It is nested into the main loop and therefore invoked at most \(O(m^2n^2)\) times. The loop over the conditions of the rules (line 12 to 20) has at most \(O(1)\) iterations, since there are at most 4 conditions per rule. It is nested into the rule loop and therefore invoked at most \(O(m^2n^2)\) times.

Next, we analyze the complexity of the functions called. We begin with the initialization functions. The function BuildSymbolTable needs one pass over the abstract syntax tree. The syntax tree has size \(O(m)\). Consequently the function has complexity \(O(m)\). The function CreateKnowledge needs to allocate an array of size \(O(m)\) and initialize it with empty lists. This has time complexity \(O(m)\). The function GenerateRules needs one pass over the abstract syntax tree and has a recursion for each branch statement in the program. In each recursion an entry for the trace variables is created. This entry can be computed in constant \(O(1)\) time – one symbol lookup, one array lookup and one array element write. The function has therefore time complexity \(O(m^2n^2)\). The function CreateRelation needs to allocate an array of size \(O(m^2)\) and initialize it with empty lists. This has time complexity \(O(m^2)\). The function InitKnowledge sets the knowledge variables for the inputs. There are at most \(O(m)\) input variables for which \(O(2^n)\) variables must be set. This has time complexity of at most \(O(m2^n)\). The function GenerateRules needs one pass over the abstract syntax tree. For each of the \(O(m)\) statements a
constant $O(1)$ number of rules of constant $O(1)$ size is generated. Generation is done using a case statement. Therefore the time complexity is $O(m)$.

In the loops we have to call additional functions. The function GetTraces needs to look up the truth value of propositional variables (in all traces). A lookup of a program variable is an array access and therefore constant ($O(1)$), but we need to traverse and copy the list of worlds (size $O(2^n)$). Therefore time complexity is $O(2^n)$. Set intersection $\cap$ operates on sets of sizes at most $O(2^n)$. Its time complexity is $O(2^n)$. The function SetVar needs to lookup the truth value of a propositional variable (in all traces). A lookup of a program variable is an array access and therefore constant ($O(1)$).

In Table 1 the sequential composition. It is the number of functions, the overall complexity is the sum of resulting contribution – the product of the previous two – to the overall time complexity. Since there are a constant number of functions, the overall complexity is the sum of the sequential composition. It is $O(m^2 2^n)$, i.e., polynomial in the number of statements.

### 5. Examples

In this section we present a number of examples we ran with an implementation of our inference algorithm. We demonstrate that we not only achieve an asymptotically better bound, but make larger programs amenable to our analysis. Particularly, in Section 5.3 we show that we can now automatically perform the formerly manual optimization of the AES key schedule.

#### 5.1 Set Equality

In this toy example we compare two sorted sets. Let Alice have $l$ elements $a_i$ ($1 \leq i \leq l$) and Bob have $l$ elements $b_i$. We securely compute $\bigwedge_{i=1}^{l}(a_i = b_i)$. This program has $m = 2l - 1$ statements and $n = 0$ branches. It is therefore not susceptible to a state explosion in the number of branches, but only in the number of statements. Using this example we can easily show the exponential growth in the number of statements of the running time of algorithm in [23]. Whereas our algorithm is polynomial in the number of statements.

We ran the example for $l \in \{1, 2, 3\}$. Our implementation is based on Java 7 64-Bit on Windows 7 Professional 64-Bit. We used a laptop with Intel Core Duo e6300 (2 MB Cache, 1.86 GHz) and 2 GB RAM for our experiments. We stopped the experiment for the algorithm of [23] and $l = 3$ after roughly 37 minutes (220000 ms). Our algorithm runs in this case in 125 milliseconds. Our results are summarized in Table 2.

<table>
<thead>
<tr>
<th>$l$</th>
<th>Algorithm 1 [23]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>93</td>
</tr>
<tr>
<td>2</td>
<td>109</td>
</tr>
<tr>
<td>3</td>
<td>125 &gt; 2200000</td>
</tr>
</tbody>
</table>

#### 5.2 Median

In [23] Aggarwal et al.’s protocol for computing the median was used as the running example. Again, let Alice and Bob have $l$ elements $a_i$ and $b_i$, respectively. We securely compute the median of the set $\{a_1, \ldots, a_l, b_1, \ldots, b_l\}$. This program has $m = 2l \log_2 l + 3$ statements and $b = \log_2 l + 1$ branches.

Aggarwal et al.’s protocol proceeds as follows: Each party locally computes its median. These values are then compared securely. Depending on the result, either the lower or upper half of one party’s input is discarded. The protocol then iterates over the remaining sets. In [23] it was shown that Aggarwal et al.’s protocol can be deduced from its FairPlay specification. It was further shown that this can be practically performed for $l = 2$.

Based on our experiments and calculations we estimate that the run-time of [23] for $l = 4$ exceeds our life-time. Instead we used our implementation of Algorithm 1 and were able to complete the inference in 4477 ms.

Furthermore, it is noteworthy that Aggarwal et al.’s protocol does a manual local computation optimization itself. All lists are sorted before used as input to the FairPlay. Our algorithm is now also able to automatically infer this sorting as a local computation when programmed in FairPlay due to its support of arrays.
5.3 AES

Another interesting example of secure computation optimization is the advanced encryption standard (AES). Let Alice have a message $x$ and Bob a secret key $k$. Then they securely compute the cipher $E_k(x)$ (where only Alice receives the output). This computation has many practical applications and was first introduced in [33]. They showed feasible runtimes even in the malicious model. Their performance results were superseded by [19] which claims that a significant performance gain is due to the local computation of the key schedule. AES is computed in rounds and in each round the expanded key is XOR-ed with the cipher. The key expansion is computed by the key schedule and can be computed from the key only. Therefore Bob can perform this as a local pre-computation.

Where [19] did this inference manually, we show that our implementation is now able to perform this automatically. In order to save space (see Section 6) we implemented only the first, the second and the last round of AES. This implementation has $m = 720$ statements and $n = 0$ branches. Clearly, this is not amenable to the algorithm of [23]. Instead our algorithm runs in $295122\ms \approx 295$ seconds and successfully infers the local key schedule.

This underpins that our inference algorithm presented in this paper is capable of analyzing significantly larger programs than [23] and makes several new, useful example amenable to automatic analysis.

6. CONCLUSIONS AND FUTURE WORK

In this paper we have presented an improved inference algorithm for secure computation optimization. It works based on an insight of the structure of the inference rules and completely avoids the use of epistemic modal logic and a corresponding theorem prover. It is therefore possible to reduce time complexity from exponential in the number of statements to polynomial. Since also space complexity is polynomial in the number of statements, this does not represent a mere space-time tradeoff, but a significant improvement in the complexity of secure computation optimization.

We have shown that this complexity reduction makes additional, larger programs amenable to automatic analysis, such as the key schedule of the AES. This formerly manual optimization can now be performed automatically.

Nevertheless, several problems are still open as future work. While we significantly improve runtime the space requirements of our implementation are quite high in practice albeit also polynomial. As such, we could not complete the median example for $l = 8$ due to memory exhaustion. It is therefore necessary to reduce memory consumption particularly for multiple traces.

Also, our algorithm is still exponential in the number of branches. This may lead to a state explosion for programs with many branches. Unfortunately the backward rule for the condition of a conditional assignment requires knowledge about all traces. As such it is not feasible to compute this rule without knowledge of all traces. Furthermore, the entire median example depends on this rule.

We expect that the following strategy might be capable of dealing with the memory problems. First, our algorithm does not consider all traces at once, but one trace at a time. Then, the knowledge about all traces – as required in this one rule – is only computed on demand, if all other conditions are also fulfilled. This does not reduce time complexity in the worst case, since all traces may need to be considered (as in the median example), but may reduce space complexity. We leave this on-demand computation of the traces as future work.

7. REFERENCES
