Faster Secure Computation through Automatic Parallelization

Niklas Buescher and Stefan Katzenbeisser, ***Technische Universität Darmstadt***

https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/buescher

This paper is included in the Proceedings of the 24th USENIX Security Symposium

August 12–14, 2015 • Washington, D.C.

ISBN 978-1-939133-11-3

Open access to the Proceedings of the 24th USENIX Security Symposium is sponsored by USENIX

Faster Secure Computation through Automatic Parallelization

Niklas Buescher
Technische Universität Darmstadt

Stefan Katzenbeisser
Technische Universität Darmstadt

Abstract

Secure two-party computation (TPC) based on Yao's garbled circuits has seen a lot of progress over the past decade. Yet, compared with generic computation, TPC is still multiple orders of magnitude slower. To improve the efficiency of secure computation based on Yao's protocol, we propose a practical parallelization scheme. Its advances over existing parallelization approaches are twofold. First, we present a compiler that detects parallelism at the source code level and automatically transforms C code into parallel circuits. Second, by switching the roles of circuit generator and evaluator between both computing parties in the semi-honest model, our scheme makes better use of computation and network resources. This *inter-party parallelization* approach leads to significant efficiency increases already on single-core hardware without compromising security. Multiple implementations illustrate the practicality of our approach. For instance, we report speed-ups of up to 2.18 on 2 cores and 4.36 on 4 cores for the example application of parallel modular exponentiation.

1 Introduction

In the thirty years since Yao's seminal paper [34], Secure Multiparty Computation (MPC) and Secure Two-Party Computation (TPC) have transitioned from purely theoretic constructions to practical tools. In TPC, two parties jointly evaluate a function f over two inputs x and y provided by the parties in such a way that each party keeps its input unknown to the other. TPC enables the construction of privacy-enhancing technologies which protect sensitive data during processing steps in untrusted environments.

Many privacy enhancing implementations use the approach of "garbled circuits" introduced by Yao in the 1980s, where f is transformed into a Boolean circuit C_f and encrypted in a special way. Beginning with the realization of the first practical implementation of Yao's protocol by Fairplay in 2004 [27], theoretical and practical advances, including Garbled-row-reduction [31], free-XOR [21], garbling from fixed-key blockciphers [5] and others have led to a significant speed-up of Yao's original protocol. Furthermore, high-level language compilers [14, 22, 23] that demonstrate the usability of Yao's protocol have been developed. Nowadays, millions of gates can be garbled on off-the-shelf hardware within seconds. Nonetheless, compared with generic computation, Yao's garbled circuits protocol is still multiple orders of magnitudes slower. Even worse, recently Zahur et al. [36] indicated that the information theoretic lower bound on the number of ciphertexts for gate-by-gate garbling techniques has been reached. Hence, further simplification of computations is unlikely.

Observing the ongoing trend towards parallel hardware, e.g., many-core architectures on a single chip, we investigate whether parallelism within Yao's protocol targeting security against semi-honest adversaries can be exploited to further enhance its performance. To the best of our knowledge, all previous parallelization efforts have focused on Yao's protocol secure against malicious adversaries, which is easily parallelizable by "design", or explored the parallelization possibilities of semi-honest Yao to only limited extent (see § 2) by using manually annotated parallism in handcrafted circuits.

Therefore, in this paper we systematically look at three different levels of automatic parallelization that have the potential to significantly speed up applications based on secure computation:

Fine-grained parallelization (FGP). As the first step, we observe that independent gates, i.e., gates that do not provide input to each other, can be garbled and evaluated in parallel. Therefore, a straight forward parallelization approach is to garble gates in parallel that are located at the same circuit depth, because these are guaranteed to be independent. We refer to this approach as *fine-grained parallelization (FGP)* and show that this approach can

be efficient for circuits of suitable shape. For example, when securely computing a matrix-vector multiplication with 3 million gates, we report a speed-up of 2.4 on 4 cores and 7.5 on 16 cores. Nevertheless, the achievable speed-up heavily depends on circuit properties such as the average circuit width, which can be comparably low even for larger problems when compiling from a high-level language, as we show.

Coarse-grained parallelization (CGP). To overcome the limitations of FGP for inadequately shaped circuits, we make use of high-level circuit descriptions, such as program blocks, to automatically detect larger coherent clusters of gates that can be garbled independently. We refer to this parallelization as *coarse-grained parallelization (CGP)*. As one of our main contributions, we extend the CBMC-GC compiler of Holzer et al. [14], which translates functionalities described in ANSI-C into circuits, with the capability to detect concurrency at the source code level. This enables the compilation of parallel circuits. Hence, one large circuit is automatically divided into multiple smaller, independently executable circuits. We show that these circuits lead to more scalable and faster execution on parallel hardware. For example, the matrix-vector multiplication circuit, mentioned above, scales with a speed-up of 3.9 on 4 cores and a speed-up of 12 on 16 cores, thus, significantly outperforming FGP. Furthermore, integrating automatic detection of parallel regions into a circuit compiler gives potential users the opportunity to exploit parallelism without knowledge of the internals of Yao's garbled circuits and relieves them of writing parallel circuits.

Inter-party parallelization (IPP). Finally, we present an extension to Yao's garbled circuit protocol secure against semi-honest adversaries to balance the computation costs of both parties. In the original protocol (using the defacto standard point-and-permute optimization [4, 27]), the garbling party has to perform four times the cryptographic work than the evaluating party. Hence, assuming similar computational capabilities the overall execution time is dominated by the garbling costs. Given the identified coarse-grained parallelism, the idea of our protocol is to divide the work in a symmetric manner between both parties by switching the roles of the garbling and evaluating party to achieve better computational resource utilization without compromising security in the semi-honest model. This approach can greatly reduce the overall runtime. By combining CGP and IPP, we report a speed-up over a serial implementation of 2.14 when using 2 cores and a speed-up of 4.34 when using 4 cores for the example application of a modular exponentiation.

Summarizing our results, the performance of Yao's protocols secure against semi-honest adversaries can significantly be improved by using automatic parallelization.

Outline. Next, we discuss related work. An introduction of the used primitives and tools is given in § 3. In § 4 we discuss FGP, CGP, and present our parallel circuit compiler. Moreover, we introduce the IPP protocol in § 5. In § 6 we evaluate our approaches on practical example applications.

2 Related Work

We give a short overview on parallelization approaches for Yao's garbled circuits in the semi-honest model, before discussing solutions in the malicious model. Furthermore, we discuss parallel compilation approaches for multi-party computation.

Semi-honest model. Husted et al. [17] showed a CPU and GPU parallelization with significant speed-ups on both architectures. Their approach is based on the idea of integrating an additional encryption layer between every circuit level to enable efficient fine-grained parallelization. However, their approach significantly increases the communication costs by sending one additional ciphertext per XOR gate. Moreover, bandwidth saving optimizations, such as garbled row reduction, are incompatible. This is undesirable, as network bandwidth is already a significant bottleneck.

Barni et al. [3] proposed a parallelization scheme similar to ours, which distinguishes between fine- and coarse-grained parallelism. Their approach showed speed-ups for two example applications. However, their coarse-grained approach requires manual user interaction to annotate parallelism in handcrafted circuits. Unfortunately, their timing results are hardly comparable with other work, due to the missing implementation of concurrent circuit generation and evaluation, which is required to garble larger circuits.

Most recently, Nayak et al. [30] presented an orthogonal and complementary work to ours. Their framework *GraphSC* supports the parallel computation of graph oriented applications using RAM based secure computation. GraphSC shows very good scalability for data intensive computations. Parallelism has to be annotated manually and has to follow the Pregel [26] pattern. To exploit further parallelism within different computing nodes of GraphSC, the ideas presented in this work could be exploited.

Malicious model. The "Billion gates" framework by Kreuter et al. [23] was designed to execute large circuits on cluster architectures. The framework supports parallelization in the malicious model using message passing technologies. Frederiksen et al. [11] also addressed the malicious model, yet they targeted the GPU as execution environment. In both cases, the protocol is based upon the idea of *cut-and-choose*, which consists of mul-

tiple independent executions of Yao's protocol secure against semi-honest adversaries. This independence enables naive parallelization up to the constant number of circuits required for cut-and-choose. Unfortunately, this degree of parallelism cannot be transferred to the semi-honest setting considered in this paper.

Parallel compiler. Zhang et al. [37] presented a compiler for distributed secure computation with applications for parallelization. Their compiler converts manually annotated parallelism in an extension of C into secure implementations. Even so the compiler is targeting MPC and not TPC, it could be used as an additional front-end to the ideas presented in this work.

In summary, up to now there is no work that addresses parallelization of Yao's protocol in the semi-honest model without making compromises towards the communication costs or relying on manually annotated parallism in handcrafted circuits.

3 Preliminaries

In this section, we give a short introduction into existing tools and techniques required for our parallelization approach. First, we give a brief overview of Yao's protocol (§ 3.1). Next, we introduce the compiler CBMC-GC that transfers ANSI-C to garbled circuits (§ 3.2), followed by an introduction of the Par4all framework that detects parallelism on source code level (§ 3.3).

3.1 TPC and Yao's protocol

In the following paragraphs, we give a short introduction into Yao's TPC protocol. For a complete description, we refer the reader to the detailed work of Lindell and Pinkas [25].

Semi-honest adversary model. In this work, we use the *semi-honest* (passive) adversary model. TPC protocols secure against semi-honest adversaries ensure correctness and guarantee that the participating parties do not learn more about the other party's input than they could already derive from the observed output of the joint computation. The semi-honest model is opposed the *malicious model*, where the adversary is allowed to actively violate the protocol. TPC protocols in the semi-honest model are used for many privacy-preserving applications and are therefore interesting on their own. A discussion on example applications is given in § 5.3.

Oblivious transfers. An Oblivious transfer protocol (OT) is a protocol in which a sender transfers one of multiple messages to a receiver, but it remains oblivious which piece has been transferred. In this paper, we use 1-out-of-2 OTs, where the sender inputs two l-bit strings m_0, m_1 and the receiver inputs a bit $c \in \{0,1\}$. At the end of the protocol, the receiver obliviously receives m_c such that neither the sender learns the choice c nor the receiver learns anything about the other message m_{1-c}. In 2003 Ishai et al. [18] presented the idea of OT Extension, which significantly reduces the computational costs of OTs for most interesting applications of TPC. We use OT_l^n to denote a number n of 1-out-of-2 oblivious transfers with message bit length l.

Yao's protocol. Yao's garbled circuits protocol, proposed in the 1980s [35], is a TPC protocol secure in the semi-honest model. The protocol is executed by two parties P_A, P_B and operates on functionality descriptions in form of Boolean circuits denoted with C_f. A Boolean circuit consists of n Boolean gates, two sets of inputs wires (one for each party), and a set of output wires. A gate is described by two input wires w_l, w_r, one output wire w_o, and a Boolean function $\gamma = g(\alpha, \beta)$ mapping two input bits to one output bit. The output of each gate can be used as input to multiple subsequent gates.

During protocol execution, one party becomes the circuit generator (the garbling party), the other the circuit evaluator. The generator initializes the protocol by assigning each wire w_i in the circuit two random labels w_i^0 and w_i^1 of length κ (the security parameter) representing the respective values 0 and 1. For each gate the generator computes a *garbled truth table*. Each table consists of four encrypted entries of the output wire labels w_o^γ. These are encrypted according to the gate's Boolean functionality using the input wire labels w_l^α and w_r^β as keys. Thus, an entry in the table is encrypted as

$$E_{w_l^\alpha}(E_{w_r^\beta}(w_o^{g(\alpha,\beta)})).$$

After their creation, the garbled tables are randomly permuted and sent to the evaluator, who, so far, is unable to decrypt a single row of any garbled table due to the random choice of wire labels.

To initiate the circuit evaluation, the generator sends its input bits x in form of input wire labels to the evaluator. Moreover, the evaluator's input y is transferred via an OT_κ^m protocol with the generator being the OT sender and m being the number of input bits. After the OT step, the evaluator is in possession of the garbled circuit and one input label per input wire. With this information the evaluator is able to iteratively decrypt the circuit from input wires to output wires. Once all gates are evaluated, all output wire labels are known to the evaluator. In the last step of the protocol, the generator sends an output description table (ODT) to the evaluator, containing a mapping between output label and actual bit value. The decrypted output is then shared with the generator.

Optimizations. Yao's original protocol has seen multiple optimizations in the recent past. Most important are *pipe-lining* [15], which is nescessary for the evaluation

of larger circuits and a faster online execution of Yao's protocol, *garbled-row-reduction (GRR)* [31], which reduces the number of ciphertexts that are needed to be transferred per gate, and *free-XOR* [21], which allows to evaluate linear gates (XOR/XNOR) essentially for "free" without any encryption or communication costs. Most recently, Zahur et al. [36] presented an communication optimal garbling scheme, which only requires two ciphertexts per non-linear gate while being compatible with free-XOR.

3.2 CBMC-GC

In 2012, Holzer et al. [14] presented the first compiler for a large subset of C to garbled circuits, named CBMC-GC. The compiler unrolls all loops and recursive statements present in the input program up to a given or statically determined bound. Afterwards, each statement is transformed to a Boolean formula preserving the bit-precise semantics of C. The Boolean formula is then translated into a circuit, which is optimized for Yao's garbled circuits [10].

The only difference between C code and code for TPC is a special naming convention introduced by CBMC-GC. Listing 1 shows example source code for the millionaires' problem. The shown procedure is a standard C procedure, where only the input and output variables are specifically marked as designated input of party P_A or P_B (Lines 2 and 3) or as output (Line 4). Aside from this naming convention, arbitrary C computations are allowed to produce the desired result, in this case a simple comparison (Line 5).

```
1 void millionaires() {
2  int INPUT_A_income;
3  int INPUT_B_income;
4  int OUTPUT_result = 0;
5  if (INPUT_A_income > INPUT_B_income)
6   OUTPUT_result = 1;
7 }
```

Listing 1: CBMC-GC Code for Yao's Millionaires' Problem.

3.3 Automatic source code parallelization

In 2012, Amini et al. [1] presented *Par4all*, an automatic parallelizing and optimizing compiler for C. It was developed to integrate several compilation tools into one single powerful compiler. Par4all is based on the Pips [6] source-to-source compiler infrastructure that detects parallelism and uses the POCC [32] polyhedral loop optimizer to perform memory access optimizations. Par4all is capable of producing parallel OpenMP [7], Cuda and OpenCL code. Par4all operates on any ANSI-C code as input, automatically detects parallel control flow and either annotates or exports parallel regions. Annotations are realized with the OpenMP language, parallel executable kernels for Cuda/OpenCL are exported using static code analysis techniques. In this work, we mainly build upon the OpenMP output.

4 Parallelizing of Yao's Garbled Circuits

To exploit parallelism in Yao's protocol, groups of gates that can be garbled independently need to be identified. Independent gates can be garbled in parallel by the generator, as well as evaluated in parallel by the evaluator. However, detecting independent, similar sized groups of gates is known as the NP-hard graph partitioning problem [28]. The common approach to circumvent the expensive search for an optimal solution is to use heuristics. In this section, we first discuss sequential and parallel composition of functionalities (§ 4.1) and show how circuits can be garbled in parallel (§ 4.2), before introducing the fine-grained parallelization heuristic (§ 4.3) and the coarse-grained parallelization heuristic (§ 4.4).

4.1 Parallel and sequential decomposition

Throughout this paper, we consider functionalities $f(x,y)$ with two input bit strings x, y and an output bit string o. Furthermore, we use C_f to denote the circuit that represents functionality f. We refer to a functionality f as *sequentially* decomposable into *sub functionalities* f_1 and f_2 iff $f(x,y) = f_2(f_1(x,y),x,y)$.

Moreover, we consider a functionality $f(x,y)$ as parallel decomposable into sub functionalities $f_1(x,y)$ and $f_2(x,y)$ with non-zero output bit length, if a bit string permutation σ_f exists such that $f(x,y) = \sigma_f(f_1(x,y)||f_2(x,y))$, where $||$ donates a bitwise concatenation operator. Thus, functionality f can directly be evaluated by independent evaluation of f_1 and f_2. We note that f_1 and f_2 do not necessarily have to be defined over all bits of x and y. Depending on f they could share none, some, or all input bits.

We use the operator $\diamond$ to express a parallel composition of two functionalities through the existence of a permutation σ. Thus, we write $f(x,y) = f_1(x,y,) \diamond f_2(x,y)$ if there exists a permutation σ_f such that $f(x,y) = \sigma_f(f_1(x,y)||f_2(x,y))$.

We call a parallelization of f to be *efficient* if the circuit size (i.e., number of gates) of the parallelized functionality is roughly equal to the circuit size of the sequential functionality: $size(C_f) \approx size(C_{f_1}) + size(C_{f_2})$. Due to the different garbling methods for linear and non-linear gates in Yao's protocol using the free-XOR technique, $size(C_f)$ is better measured by the number of non-linear gates. Furthermore, we refer to a parallelization as

symmetric if sub functionalities have almost equal circuit sizes: $size(C_{f_1}) \approx size(C_{f_2})$.

Finally, we refer to functionalities that can be decomposed into a sequential and a parallel part as *mixed functionalities*. For example the functionality $f(x,y) = f_3(f_1(x,y) \diamond f_2(x,y), x, y)$ can first be decomposed sequentially in f_3 and $f_1 \diamond f_2$, where the latter part can then be further decomposed in f_1 and f_2.

Without an explicit definition, we note that all definitions can be extended from the dual case f_1 and f_2 to the general case $f_1, f_2, \ldots, f_n$.

4.2 Parallel circuit creation and evaluation

A circuit that consists of annotated sequential and parallel parts can be garbled in parallel as follows. Sequential regions of a circuit can be garbled using standard techniques by iterating topologically over all gates. Once a parallel decomposable region of a circuit is reached, parallelization is applied. All independent sub circuits in every parallel region can be garbled in any order by any available thread (see Figure 1). We note that the garbling order has no impact on the security [25]. After every parallel region a synchronization between the different threads is needed to guarantee that all wire labels for the next region of the circuit are computed. Multiple subsequent parallel regions with different degrees of parallelism can be garbled, when ensuring synchronization in-between.

The circuit evaluation can be parallelized in the same manner. Sequential regions are computed sequentially, parallel regions are computed in parallel by different threads. After every parallel region a thread synchronization is required to ensure data consistency.

When using pipe-lining the garbled tables have to be transmitted in an identifiable order to ensure data consistency between generator and evaluator. We propose three different variants. First, all garbled tables can be enriched with a numbering, e.g., an index, which allows a unordered transfer to the evaluator. The evaluator is then able to reconstruct the original order based on the introduced numbering. This approach has the disadvantage of an increased communication cost. The second approach is that garbled tables are sent in a synchronized and predefined order. This approach functions without additional communication, yet can lead to an undesirable 'pulsed' communication pattern. The third approach functions by strictly separating the communication channels for every sub circuit. This can either be realized by multiplexing within the TPC framework or by exploiting the capabilities of the underlying operating system. Due to the aforementioned reasons, our implementation of a parallel framework (presented in § 6.1) builds upon the latter approach.

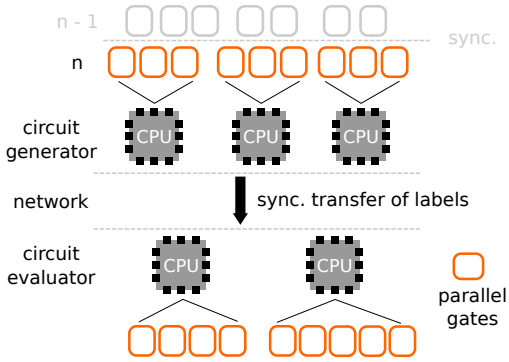


Figure 1: Interaction between a parallel circuit generator and evaluator. The layer n of the presented circuit is garbled and evaluated in parallel. The independent partitions of the circuit can be garbled and evaluated by different threads in any order.

4.3 Fine-grained parallelism

A first heuristic to decompose a circuit into parallel parts is the *fine-grained* gate level approach, described in the following. Similar to the evaluation of a standard Boolean circuit, gates in garbled circuits are processed in topological execution order. Gates provide input to other gates and hence, can be ordered by the circuit level (depth) when all their inputs are ready or the level when their output is required for succeeding gates. Consequently and as proposed by others [3, 17], gates on the same level can be garbled in parallel. Thus, a circuit is sequentially decomposable into different levels and each level is further decomposable in parallel with a granularity up to the number of gates. Figure 2 illustrates fine-grained decomposition of a circuit into three levels L1, L2 and L3.

To achieve an efficient distribution of gates onto threads during protocol execution, it is useful to identify the circuit levels during the circuit compilation process. Furthermore, a reasonable heuristic to symmetrically distribute the workload onto all threads when using the free-XOR optimization is to divide linear and non-linear gates independently. Hence, each thread gets assigned the same number of linear and non-linear gates to garble. Therefore, we extended the circuit compiler CBMC-GC with the capability to mark levels and to strictly separate linear from non-linear gates within each level. This information is stored in the circuit description.

Overhead. In practice, multi-threading introduces an computational overhead to enable thread management and thread synchronization. Therefore, it is useful to experimentally determine a system dependent threshold τ that describes the minimal number of gates that are required per level to profit from parallel execution. In practical settings (see § 6) we observe that at least ~ 8 non-

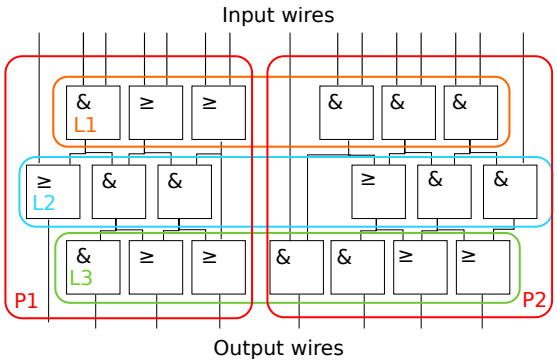


Figure 2: Circuit decomposition. Each level *L1, L2* and *L3* consists of multiple gates that can be garbled using FGP with synchronization in-between. The circuit can also be decomposed in two coarse-grained partitions *P1* and *P2*.

linear gates per core are required to observe first speed-ups. Achieving a parallelization efficiency of 90%, i.e, a speed up of 1.8 on 2 cores, requires at least 512 non-linear gates per core. In the next section we present an approach that overcomes the limitations of FGP.

4.4 Coarse-grained parallelism

Another useful heuristic to partition a circuit is the usage of high-level functionality descriptions. Given a circuit description in a high-level language, parallelizable regions of the code can be identified using programming language techniques. These detected code regions can then be tracked during the circuit compilation process to produce parallel decomposable circuits. The different *sub circuits* are guaranteed to be independent of each other and therefore can be garbled in parallel. We refer to this parallelization scheme as *coarse-grained parallelization (CGP)*. Figure 2 illustrates an example decomposition in two coarse-grained partitions P1 and P2. Furthermore, we note that FGP and CGP can be combined by utilizing FGP within all coarse partations. In the following paragraphs, we introduce our CBMC-GC compiler extension that automatically produces coarse-grained parallel circuits.

Compiler for parallel circuits. Our parallel circuit complier *ParCC* extends the CBMC-GC compiler and builds on top of the Par4all compiler introduced in § 3. ParCC takes C code as input, which carries annotations according the CBMC-GC notation. Hence, TPC input and output variables of both parties are marked as such. ParCC detects parallelism within this code with the help of Par4all and produces one *global circuit* that is interrupted by one or multiple *sub circuits* for every parallel region. If a parallel region follows the single-instruction-multiple-data paradigm (SIMD), only one sub circuit per parallel region is compiled, which reduces the circuit storage costs. During protocol runtime, the sub circuit is unrolled and garbled in full extent. The global and sub circuits are interconnected by explicitly defining *inner* input and output wires. These are not exposed as TPC inputs or outputs, but have to be used by TPC frameworks to recompose the complete and parallel executable circuit. The compilation process itself consists of four different steps:

(1) In the first step, parallelism in C code is detected by Par4all and annotated using the OpenMP notation and source-to-source transformations.

(2) The annotated C code is parsed by ParCC in the second step. The source code is decomposed using source-to-source techniques into a *global* sequentially executable part, which is interrupted by one or multiple parallel executable *sub* parts. Additionally, functional OpenMP annotations, such as reduction statements, are replaced with C code that is compiled into the circuits. Furthermore, information about the degree of detected parallelism as well as the interconnection between the global and sub parts is extracted for later compilation steps.

(3) Given the decomposed source code, the different parts are compiled independently with CBMC-GC.

(4) In the final step information about the mapping of wires between gates in the global and the sub circuits is exported for use in TPC frameworks. For performance reasons, we distinguish static wires that are shared between parallel sub circuits and wires that are dedicated for each individual sub circuit.

Example. To illustrate the functionality of ParCC, we discuss the source-to-source compilation steps on a small *fork and join* task, namely the dot product of two vectors **a** and **b**:

$$r = \mathbf{a} \cdot \mathbf{b} = a_0 \cdot b_0 + \cdots + a_n \cdot b_n.$$

The source code of the function `dot_product()` is presented in Listing 2.

```
1  int mult(int a, int b) {
2    return a * b;
3  }
4  void dot_product() {
5    int INPUT_A_a[100], INPUT_B_b[100];
6    int res = 0;
7    for(i = 0; i < 100; i++)
8      res += mult(INPUT_A_a[i], \
9        INPUT_B_b[i]);
10   int OUTPUT_res = res;
11 }
```

Listing 2: Dot vector product written in C with CBMC-GC input/output notation.

In this example code, two parties provide input for two vectors in form of constant length integer arrays (Line 5). A loop iterates pairwise over all array elements (Line 7), multiplies the elements and aggregates the result. In the first compilation step, Par4all detects the parallelism in the loop body and annotates this parallel region accordingly. Therefore, Par4all adds the statement `#pragma omp parallel for reduction(+:res)` before the `for` loop in Line 7.

ParCC parses the annotations in the second compilation step to export the loop body, in this case the sub function `mult()`, printed in Listing 3.

```
1 void mult(int INPUT_A_a, int INPUT_A_b,
2     int OUTPUT_return)
3 {
4   int a = INPUT_A_a;
5   int b = INPUT_A_b;
6   OUTPUT_return = a*b;
7 }
```

Listing 3: Exported sub function with CBMC-GC input-output notation.

The functions arguments are rewritten according the notation of CBMC-GC. Thus, the two arguments `a` and `b` of `mult()` become inner inputs of the sub circuit, and the return statement becomes an inner output variable. Note, that during the protocol execution all inner wires are not assigned to any party, instead they connect global and sub circuits. Yet, to keep compatibility with CBMC-GC a concrete assignment for the party P_A is specified. The later exported mapping information is used to distinguish between inner wires and actual input wires of both parties. In the same step, the global function `dot_product()`, printed in Listing 4, is transformed by ParCC to replace and unroll the loop.

```
1 void dot_product() {
2   int INPUT_A_a[100], INPUT_B_b[100];
3   int res = 0;
4   int OUTPUT_SUB_a[100];
5   int OUTPUT_SUB_b[100];
6   int i;
7   for(i = 0; i <= 99; i++) {
8     OUTPUT_SUB_a[i] = INPUT_A_a[i];
9     OUTPUT_SUB_b[i] = INPUT_B_b[i];
10  }
11  int INPUT_A_SUB_res[100];
12  for(i = 0; i <= 99; i++)
13    res += INPUT_A_SUB_res[i];
14  int OUTPUT_res = res;
15 }
```

Listing 4: Rewritten `dot_product()`. The loop has been replaced by inner input/output variables (marked with `SUB`).

Therefore, the two input arrays `INPUT_A_a` and `INPUT_B_b` are exposed as inner output variables beginning in Line 4. Therefore, two new output arrays using CBMC-GC notation are added based on the statically determined information about parallel variables. Furthermore, an inner input array for the intermediate results is introduced in Line 11. Finally, the reduction statement is substituted by synthesized additions over all intermediate results in Line 13.

In the third and fourth compilation step, the two circuits are compiled and the mapping of wires between global and sub circuits is exported.

5 Inter-Party Parallelization (IPP)

In this section, we describe a novel protocol extension to Yao's protocol to balance computation between parties, assuming *symmetric efficiently parallelizable* functionalities. We refer to this protocol extension as *inter-party parallelization* (IPP). Without compromising security, we show in § 6 that the protocol runtime can be reduced in practical applications when using IPP. This is also the case when using only one CPU core per party.

We recap the initial motivation: The computational costs that each party has to invest in semi-honest Yao is driven by the encryption and decryption costs of the garbled tables as well as the communication costs. Considering the garbling technique with the least number of cryptographic operations, namely GRR combined with free-XOR, the generator has to compute four ciphertexts per non-linear gate, whereas the evaluator has to compute only one ciphertext per non-linear gate. When considering the communication optimal half-gate approach [36], the generator has to compute four and the evaluator two ciphertexts per non-linear gate. Assuming two parties that are equipped with similar computational power, a better overall resource utilization would be achieved, if both parties could be equally involved in the computation process. This can be realized by sharing the roles generator and evaluator. Consequently, the overall protocol runtime could be decreased. Figure 3 illustrates this efficiency gain.

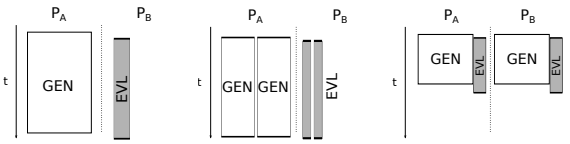
In the following sections we first discuss how to extend Yao's protocol to use IPP for purely parallel functionalities. In a second step we generalize this approach by showing how mixed functionalities profit from IPP.

5.1 Parallel functionalities

We assume that two parties P_A and P_B agree to compute a functionality $f(x,y)$ with x being P_A's input and y being P_B's input. Moreover, we assume $f(x,y)$ to be parallelizable into two (or more) sub functionalities $f_0,\ldots,f_n$:

$$f(x,y) = f_0(x,y) \diamond f_1(x,y) \diamond \ldots \diamond f_n(x,y).$$

Given such a decomposition, all sub functionalities can be computed independently with any TPC protocols (secure against semi-honest adversaries) without



(a) Original (b) Independence analysis (c) Inter-party Parallelization

Figure 3: The idea and performance gain of IPP visualized. The OT phase and output sharing are omitted. In Figure 3(a) the sequential execution of Yao's protocol is visualized. Given a parallel decomposition by two circuits representing parallal program regions as displayed in Figure 3(b), the protocol runtime can be reduced when sharing the roles generator and evaluator, as displayed in Figure 3(c).

any sacrifices towards the security [12]. This observation allows us to run two independent executions of Yao's protocol, each for one half of f's sub functionalities, instead of computing f with a single execution of Yao's protocol. Hence, P_A could garble one half of f's sub functionalities, for example $f_{even} = f_0, f_2, \ldots$, and P_A could evaluate the other half $f_{odd} = f_1, f_3, \ldots$. Vice versa, P_B could evaluate f_{even} and garble f_{odd}. Following this approach, P_A and P_B have to switch their roles for the OT phase of Yao's protocol. In the output phase, both parties share their output labels and description tables (ODT) with each other.

Analytical performance gain. As discussed, the computational costs for Yao's protocol are dominated by encrypting and decrypting garbled truth tables. Thus, idealizing and highly abstracted, the time spent to perform a computation t_{total} is dominated by the time to garble a circuit t_{garble}. Using GRR with free-XOR and assuming that t_{garble} is approximately four times the time to evaluate a circuit t_{eval}, by symmetrically sharing this task the total time could be reduced to:

$$t'_{total} \approx \frac{(t_{garble} + t_{eval})}{2} \approx \frac{(4 \cdot t_{eval} + t_{eval})}{2} \approx 2.5 \cdot t_{eval}.$$

This result translates to a theoretical speed-up of $t_{total}/t'_{total} = 4/2.5 = 1.6$. When using the half-gate approach the approximate computational speed-up is 1.33.

5.2 Mixed functionalities

To exploit IPP in mixed functionalities, a protocol extension is required, allowing to switch from sequential (dedicated roles) to IPP (shared roles) without violating the privacy property of TPC. Therefore, we introduce the notion of *transfering roles* to securely interchange between IPP and sequential execution.

Transferring roles We introduce the idea of transferring roles in two steps. First we sketch an insecure protocol, which is then made secure in a second step. To switch the roles of evaluator and generator during execution, we consider two parties P_A, P_B and the sequentially composed functionality $f(x,y) = f_2(f_1(x,y),x,y)$. In the following description, f_1 is computed using Yao's protocol with P_A being generator and P_B being evaluator, f_2 is computed with reversed roles.

The transfer protocol begins by computing $f_1(x,y)$ with Yao's original protocol. Once f_1 is computed, the roles have to be switched. Naïvely, the parties 'open' the circuit by interchanging output wires and the ODT. This reveals the intermediate result $o_1 = f_1(x,y)$ to both parties. In the second phase of the protocol, f_2 is computed using Yao's protocol. This time, P_A and P_B switch roles, such that P_B garbles f_2 and P_A evaluates f_2. The decrypted output bits $o_1 = f_1(x,y)$ are used by P_A as input in the OT protocol. After garbling f_2, the output labels and the ODT are shared between both parties. This protocol resembles a pause/continue pattern and preserves correctness. However, this protocol leaks o_1 to both parties, which violates the privacy requirement of TPC. Therefore, we propose to use an XOR-blinding during the role switch. The full protocol is printed below.

Protocol: Transferring Roles

P_A and P_B agree to securely compute the sequentially decomposable functionality $f(x,y) = f_2(f_1(x,y),x,y)$ without revealing the intermediate result $f_1(x,y)$ to either party, where x is P_A's input bit string, y is P_B's input bit string. The protocol consists of two phases, one per sub functionality.

Phase 1: Secure computation of $f_1(x,y)$

1. f_1 is extended with a XOR blinding for every output bit. Thus, the new output $o'_1 = f'_1(x,y||y_r) = f_1(x,y) \oplus y_r$ is calculated by xoring the output of f_1 with additional, randomly drawn input bits by the evaluator of f_1.

2. P_A and P_B securely compute f'_1 using Yao's protocol. We assume P_A to be the generator. Additional randomly drawn input bits are then input of P_B.

3. The blinded output o'_1 of the secure computation is only made visible to the generator P_A. This is realized by transmitting the output wire labels to P_A, but not sharing the ODT with P_B.

Phase 2: Secure computation of $f_2(o_1,x,y)$

1. The circuit representing f_2 is extended with a XOR unblinding for every input bit of o'_1. Hence, $f'_2(o'_1,x,y,y_r) = f_2(o'_1 \oplus y_r,x,y)$.

2. P_A and P_B securely compute f_2' using Yao's protocol. We assume P_B to be the generator. P_A provides the input o_1' and P_B provides the input bits for the blinding with y_r.

3. The output of the computation is shared with both parties.

We observe that, informally speaking the protocol preserves privacy, since the intermediate state o_1 is shared securely between both parties. A detailed formal proof on sequential decomposed functionalities is given by Hazay and Lindell [12, page 42ff]. Correctness is preserved due to blinding and unblinding with the equal bit string y_r:

$$\begin{aligned} f_2'(f_1'(x,y||y_r),x,y,y_r) &= f_2'(f_1(x,y)\oplus y_r,x,y,y_r) \\ &= f_2(f_1(x,y)\oplus y_r\oplus y_r,x,y) \\ &= f_2(f_1(x,y),x,y). \end{aligned}$$

Finally, we note that transferring roles protocol can further be improved. Demmler et al. [8] presented an approach to securely share a state of Yao's protocol that uses the point-and-permute bits [27] as a blinding. This approach has equivalent costs in the number of cryptographic operations, yet removes the need of an ODT transfer. Our implementation uses this optimization.

Transferring roles for mixed functionalities. With the idea of transferring roles, IPP can be realized for mixed functionalities. In the following paragraphs, we show how to switch from IPP to sequential computation. Switching into the other direction, namely from sequential to IPP can be realized analogously. With protocols to switch in both directions, it is possible to garble and evaluate any functionality that consists of an arbitrary number of sequential and parallel regions.

To show the switch from IPP to sequential computation, we assume a functionality that is sequentially decomposable into a parallel and a sequential functionality:

$$f(x,y) = f_3(f_1(x,y)\diamond f_2(x,y),x,y).$$

Note that f_1, f_2 and f_3 could further be composed of any sequential and parallel functionalities. We observe that f_3 can be merged with f_1 (or f_2) into one combined functionality f_c. Thus, $f(x,y)$ can also be decomposed as $f(x,y) = f_c(f_2(x,y),x,y)$ with f_c being the sequential composition of f_3 and f_1. Given such a decomposition, f_c and f_2 can be computed with alternating roles in Yao's protocol by following the transferring roles protocol. Hence, f_c could be garbled by P_A while f_2 could be garbled by P_B to securely compute f.

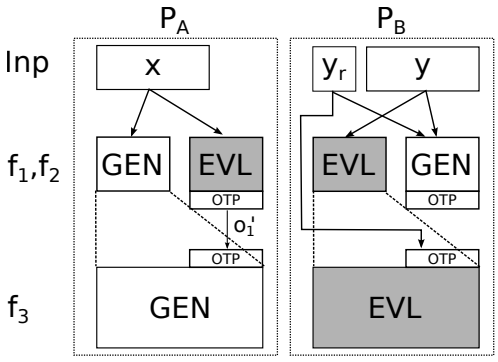


Figure 4: The IPP protocol for a mixed functionality with a switch from parallel to sequential computation. Functionality $f_1 \diamond f_2$ is garbled in parallel using IPP, f_3 is garbled sequentially in combination with f_1. No interaction between parties is shown. The blinded output o_1' of f_1 is only made visible to P_A and used as additional input for the computation of f_2 using the transferring roles protocol.

As a second observation we note that the output of f_2 is not required to start the computation of f_c. Therefore, the computation of f_c can start in parallel to the computation of f_2. This inter-party parallelism can be exploited to achieve further speed-ups. Figure 4 illustrates this approach. Party P_A garbles f_c and P_B garbles f_2. The first part of f_c, namely f_1 can be garbled in parallel to f_2. Once the blinded output o_1' of f_2 is computed, the parties can start computing the second part of f_c, namely f_3. Switching from sequential to IPP computation can be realized in the same manner.

We remark that FGP, CGP and IPP can be combined to achieve even further speed-ups. Therefore, every parallel region has first be decomposed in two parts for IPP. If the parts can further be decomposed in parallel functionalities, these could be garbled following the ideas of CGP and FGP.

Overhead. Investigating the overhead of IPP, we observe that during the cost intensive garbling and evaluation phase, no computational complexity is added. Particularly, the number of cryptographic operations and messages is left unchanged. However, when using OT Extension, a constant one-time overhead for base OTs in the size of the security parameter k is introduced to establish bi-directional OT Extension. To switch from and to IPP in mixed functionalities, additional OTs in the size of the intermediate state are required. Thus, the performance gain through IPP for mixed functionalities not only depends on the ratio between parallel and sequential regions, but also on the ratio of circuit size and shared state. These ratios are application dependent. A practical evaluation of the trade-off between overhead and performance gain is presented in § 6.5.

5.3 Security implications and applications

As discussed in the previous section, Yao's protocol and IPP are secure against semi-honest adversaries. Nevertheless, semi-honest Yao's garbled circuits are often used to bootstrap TPC protocols secure against active adversaries. Therefore, in this section, we sketch the security implications of IPP and its compatibility with the most common techniques to strengthen Yao's protocol. Furthermore, we depict applications and protocols the could profit from IPP.

Yao's original protocol is already secure against malicious evaluators (when using an OT protocol secure against malicious adversaries), yet not secure against malicious generators. We note that this one-sided security does not longer hold when using IPP because both parties incorporate the role of a circuit generator. Consequently, cut-and-choose protocols [24], which garble multiple copies of the same circuit to achieve active security, are incompatible with IPP because they are built on the assumption that only one party can actively manipulate the protocol. A similar observation can be made for Dual-execution protocols [29, 16] that prevent an active adversary from learning more than a small number of bits during the protocol execution. Even so the concept of Dual-execution is close the the idea of IPP, i.e., symmetrically sharing the roles of generator and evaluator, it also requires one-sided security against malicious evaluators and is therefore incompatible with IPP.

Applications of IPP. IPP can be applied in all application scenarios where semi-honest model is sufficient. These are scenarios where either the behaviour is otherwise restricted, e.g. limited physical access, or where the parties have sufficient trust into each other. Moreover, IPP can be used in all the scenarios where the parties inputs and seeds could be revealed at a later point to identify cheating parties. Examples might be negotiations (auctions) or games, such as online poker. Another field of application is the joint challenge creation, e.g. RSA factorization. Using secure computation, two parties could jointly create a problem instance without already knowing the solution. This allows them to create a problem and to participate in the challenge at the same time without a computational advantage. Once a solution is computed, all parts of the secure computation can be verified in hindsight.

For further work, we note that the core idea of IPP could be applied in other TPC protocols. To profit from IPP, a state sharing mechanism in the protocols security model is required as well as an asymmetric workload between the parties. One example might be the highly asymmetric STC protocol by Jarecki and Shmatikov [19] that uses zero-knowledge proofs over every gate.

6 Evaluation

In this section, we evaluate the three proposed parallelization schemes. We begin by introducing our parallel STC framework named UltraSFE and benchmark its performance on a single core in § 6.1. The applications and their circuit descriptions used for benchmarking are described in § 6.2. We evaluate the offline garbling performance of the proposed parallelization techniques in § 6.3, before integrating and evaluating the promising coarse-grained parallelization (CGP) in a online setting in § 6.4. Finally, in § 6.5 we benchmark the inter-party parallelization (IPP) approach.

6.1 UltraSFE

UltraSFE[1] is a parallel framework for Yao's garbled circuits built up on the *JustGarble (JG)* [5] garbling scheme. To realize efficient parallelization, data structures and memory layout are optimized with the purpose of parallelization in mind. This is, to the best of our knowledge, not the case with existing open source frameworks. Therefore, we adapted the JustGarble garbling scheme to support parallelization.

UltraSFE is written in C++ using SSE4, OpenMP and Pthreads to realize multi-core parallelization. Conceptually UltraSFE is using ideas from the Java based memory efficient *ME_SFE* framework [13], which itself is based on the popular *FastGC* framework [15]. The fast hardware AES extension in current CPU generations is exploited by the JustGarble garbling scheme. Oblivious transfers are realized with the help of the highly efficient and parallelized OTExtension library written by Asharov et al. [2]. Moreover, UltraSFE adopts techniques from many recent theoretical and practical advances of Yao's protocol. This includes pipe-lining, point-and-permute, garbled row reduction, free-XOR and the half-gate approach [15, 21, 27, 31, 36].

Framework comparison. To illustrate that UltraSFE is suited to evaluate the scalability of different parallelization approaches, we present a comparison of its garbling performance with other state of the art frameworks using a CPU architecture in Table 1. Namely, we compare the single core garbling speed of UltraSFE, which is practically identical to JG, with the parallel frameworks by Barni et al. (*BCPU*) [3], Husted et al. (*HCPU*) [17] and Kreuter et al. (*KSS*) [23]. Note, these results are compared in the *offline* setting, i.e., truth tables are written to memory. This is because circuit garbling is the most cost intensive part of Yao's protocol and therefore the most interesting when comparing the performance of different frameworks. The previous parallelization efforts HCPU

[1] UltraSFE will be made available as open source on http://www.seceng.informatik.tu-darmstadt.de/research/software/

	Ours / JG [5]	BCPU [3]	HCPU[17]	KSS [23]
gps	8.3M	0.11M	<0.25M	0.1M
cpg	108	>3500	-	>6500 [5]
arch	E5-2680	E5-2609	E5-2620	i7-970

Table 1: Single core garbling speed comparison of different frameworks on circuits with more than 5 million gates. Metrics are non-linear gates per second (*g*ps) in millions (M) and clocks per gate (*cpg*). All results have been observed on the Intel processor specified in row *arch*. Note, for HCPU [17] only circuit evaluation times have been reported on the CPU, the garbling speed can be assumed to be lower.

and BCPU actually abstained from implementing an *online* version of Yao's protocol that supports pipe-lining. As metrics we use garbled gates per second (*gps*) and the average number of CPU clock cycles per gate (*cpg*), as proposed in [5]. The numbers are taken from the cited publications and if not given, the cpq results are calculated based on the CPU specifications (*arch*). Even when considering theses numbers only as rough estimates, due to the different CPU types, we observe that UltraSFE performs approximately 1-2 orders of magnitude faster than existing parallelizations of Yao's protocol. This is mostly due to the efficient fixed-key garbling scheme using the AES-NI hardware extension and a carefully optimized implementation using SSE4. Summarizing, UltraSFE shows competitive garbling performance on a single core and hence, is a very promising candidate for parallelization.

6.2 Evaluation methodology

To evaluate the different parallelization approaches we use three example applications that have been used to benchmark and compare the performance of Yao's garbled circuits in the past.

Biometric matching (BioMatch). The first application that we use is privacy-preserving biometric matching. In this application a party matches one biometric sample against the other's party database of biometric templates. Example scenarios are face-recognition or fingerprint-matching [9]. One of the encompassing concepts is the computation of the Euclidean distance between the sample and all database entries. Once all distances have been computed, the minimal distance determines the best match. Thus, the task is to securely compute the minimal distance $min\left(\sum_{i=1}^{d}(s_{i,1}-e_i)^2,\ldots,\sum_{i=1}^{d}(s_{i,n}-e_i)^2\right)$ with s_i being the sample of degree d provided by the first party and $e_1,\ldots,e_n$ being the database elements with the same degree provided by the other party. Following the examples of [8, 20], the chosen parameters for this circuit are the number of elements in the database $n = 512$, the degree of each element $d = 4$ and the integer size $b = 64$bit.

	BioMatch	**MExp**	**MVMul**
Code size	22 LOC	28 LOC	10 LOC
Circuit size	66M	21.5M	3.3M
Non-linear gates	25%	41%	37%
# Input bits P_A/P_B	131K/256	1K/1K	17K/1K
Offline garbling time	2.07s	1.136s	0.154s

Table 2: *Circuit properties.* Presented are the code size, the overall circuit size in the number of gates, the fraction of non-linear gates that determine the majority of computing costs, the number of input bits as well as the sequential offline garbling time with UltraSFE.

Modular exponentiation (MExp). The second application that we benchmark is parallel modular exponentiation. Modular exponentiation has been used before to benchmark the performance of Yao's garbled circuits [5, 8, 23]. It has many applications in privacy-preserving computation. For example, blind signatures where the message to be signed should not be revealed to the signing party. For this application, we differentiate the circuit by the number of iterated executions $k = 32$, as well as the integer width $b = 32$.

Matrix-vector multiplication (MVMul). Algebraic operations such a matrix multiplication or the dot vector product are building blocks for many privacy-preserving applications and have been used before to benchmark Yao's garbled circuits [14, 22]. We use a Matrix-vector multiplication as required in the learning with errors (LWE) cryptosystem [33]. We parametrize this task according the size of the matrix $m \times k = 16 \times 16$ and vector $k = 16$, as well the integer size of each element $b = 64$ bit.

Circuit creation. All circuits are compiled twice, once with CBMC-GC and once with ParCC using textbook C implementations. The time limit for the circuit minimization through CBMC-GC is set to 10 minutes. The resulting circuits and their properties are shown in Table 2. The BioMatch circuit is the largest circuit and shows the most input bits. The MVMul circuit garbles in a fraction of a second and thus, fits to evaluated the performance of parallelelization on smaller circuits. The MExp circuit shows a large circuit complexity in comparison to the number of input bits. Even so not shown here, we note that the sequential (CBMC-GC) and parallel (ParCC) circuits slightly differ in the overall number of non-linear gates due to the circuit minimization techniques of CBMC-GC, which profit from decomposition.

Environment. As testing environment we used Amazon EC2 cloud instances. These provide a scalable number of CPUs and can be deployed at different sites around the globe. If not state otherwise, for all experiments instances of type `c3.8xlarge` have been used. These instances report 16 physical cores on 2 sockets with CPUs

of type Intel Xeon E5-2680v2, and are equipped with a 10Gbps ethernet connection. A fresh installation of Ubuntu 14.04 was used to ensure as little background noise as possible. UltraSFE was compiled with `gcc 4.8 -O2` and `numactl` was utilized when benchmarking with only a fraction of the available CPUs. Numactl allows memory, core and socket binding of processes. Results have been averaged 10 executions.

Methodology. Circuit garbling is the most expensive task in Yao's protocol. Therefore, we begin by evaluating FGP and CGP for circuit garbling independent of other parts of Yao's protocol. This allows an isolated evaluation of the computational performance gains through parallelization. Following the offline circuit garbling phase is an evaluation of Yao's full protocol in an online LAN setting. This evaluation also considers the bandwidth requirements of Yao's protocol. Finally, we present an evaluation of the IPP approach in the same LAN setting. Therefore, we first evaluate the performance of IPP on a single core, before evaluating its performance in combination with CGP. The main metric in all experiments is the overall runtime and the number of non-linear gates that can be garbled per second.

6.3 Circuit garbling (offline)

We begin our evaluation of FGP and CGP with the offline task of circuit garbling. In practice the efficiency of any parallelization is driven by the ratio between computational workload per thread and synchronization between threads. When garbling a circuit with FGP, the workload is bound by the width of each level, when garbling with CGP the workload is bound by the size of parallel partitions. Both parameters are circuit and hence, application dependent.

Artificial circuits and thread utilization. To get a better insight, we first empirically evaluate the possible efficiency gain for different sized workloads, independent of any application. This also allows to observe a system dependent threshold τ, introduced in § 4.3, which describes the minimal number of gates required per thread to profit from parallelization. Therefore, we run the following experiment: For every level width $w = 2^4, 2^5, \ldots, 2^{10}$ we created artificial circuits of depth $d = 1000$. The width is kept homogeneous in all levels. Furthermore, the wiring between gates is randomized and only non-linear gates are used. Each circuit is garbled using FGP and we measured the parallelization efficiency (speed-up divided by the number of cores) when computing with a different numbers of threads. The results are illustrated in Figure 5.

The experiment shows that on the tested system $\tau \approx 8$ non-linear gates per thread are sufficient to observe first performance gains through parallelization. To achieve

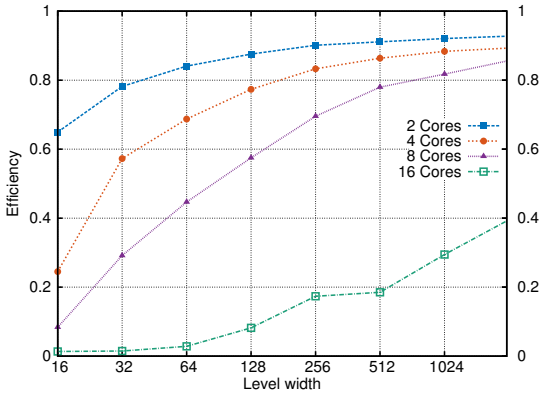


Figure 5: *Level-width experiment.* Displayed is the efficiency of FGP for different circuit level widths. A larger width increases the efficiency of parallelization. The gap between 8 (one socket) and 16 cores (two sockets) is due the communication latency between two sockets.

an efficiency of 90% approximately 512 non-linear gates per thread are required. Investigating the results for 16 parallel threads, we observe that a significantly larger workload per thread (at least one order of magnitude) is required to overcome the communication latency between the sockets on the testing hardware.

Example applications. We evaluated the speed-up of circuit garbling when using FGP and CGP for the three applications BioMatch, MExp and MVMul compiled with CBMC-GC (FGP) and ParCC (CGP). The speed-up is calculated in relation to the single core garbling performance given in Table 2. The results are presented in Figure 6. The results have been observed for a security level of $k = 128$ bit. We note, that in this experiment no significant differences where observable when using a smaller security level, e.g., $\kappa = 80$ bit, due to the fixed block size of AES-NI. Discussing the results for FGP, we observe that all applications profit from parallelization. BioMatch and MExp show very limited scalability, whereas the MVMul circuit is executable with a speed-up of 7.5 on 16 cores. Analyzing the performance of CGP, we observe that all applications achieve practically ideal parallelization when using up to 4 threads. In contrast to the FGP approach, scalability with high efficiency is observable with up to 8 threads. Further speed-ups when using the CPU located on the second socket are noticeable in the MExp and MVMul experiments, achieving a throughput of more than 100M non-linear gates per second.

In summary, for all presented applications the CGP approach significantly outperforms the FGP approach regarding scalability and efficiency due to its coarser granularity, which implies a better thread utilization.

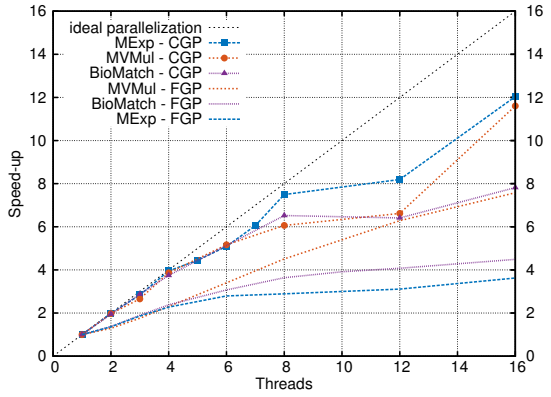


Figure 6: *Circuit garbling.* The speed-up of circuit garbling for all three applications when using the FGP, CGP and different numbers of computing threads. CGP significantly outperforms FGP for all applications.

Circuit width analysis. The limited scalability of FGP is explainable when investigating the different circuit properties. In Figure 7 the distribution of level widths for all circuits is illustrated.

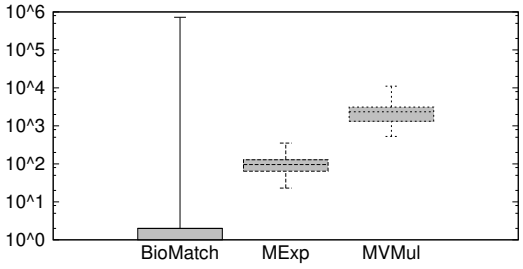


Figure 7: Number of non-linear gates per level and circuit.

For the MVMul application, the CBMC-GC compiler produces a circuit with a median level width of 2352 non-linear gates per level, whereas the BioMatch and MExp circuits only show a median width below 100 non-linear gates per level. The major reasons for small circuit widths in comparison to the overall circuit size is that high-level TPC compilers such as CBMC-GC or the compiler by Kreuter et al. [23] have been developed with a focus on minimizing the number of non-linear gates. Minimizing the circuit depth or maximizing the median circuit width barely influence the sequential runtime of Yao's protocol and is therefore not addressed in the first place. Looking at the building blocks that are used in CBMC-GC, we observe that arithmetic blocks (e.g. adder, multiplier) show a linear increase in the average circuit width when increasing the input size. However, multiplexers, as used for dynamic array accesses and for 'if' statements, show a circuit width that is independent (constant) of the number of choices. Thus, a 2-1 multiplexer and a n-1 multiplexer are compiled to circuits with similar sized levels, yet with different circuit depths. Moreover, comparisons have a constant circuit width for any input bit size. Based on these insights we deduce, that the MVMul circuit shows a significantly larger median circuit width, because of the absence of any dynamic array access, conditionals or comparisons. This is not the case with the BioMatch and MExp applications. Considering that every insufficient saturation of threads leads to an efficiency loss of parallelization, we conclude that scalability of FGP is not guaranteed when increasing input sizes.

6.4 Full protocol (online)

To motivate that the parallelization of circuit garbling can be used in Yao's full protocol, we evaluated the protocol for all applications running on two separated cloud instances in the same Amazon region (LAN setting). We observed an average round trip time of 0.6±0.3ms and a transfer bandwidth of 5.0±0.4 Gbps using `iperf`. Following the results of the offline experiments, we benchmark the more promising CGP approach in the online setting.

To measure the benefits of parallelization, we first benchmark the single core performance of Yao's protocol in the described network environment. Table 3 shows the sequential runtime for all applications using two security levels $\kappa = 80$ bit (short term) and $\kappa = 128$ bit (long term). This runtime includes the time spent on the input as well as the output phase. Furthermore, the observed throughput, measured in non-linear gates per second, as well as the required bandwidth are presented. We observe that for security levels of $\kappa = 80$ and $\kappa = 128$ a similar gate throughput is achieved. Consequently, we deduce that in this setup the available bandwidth is not stalling the computation. We also observe that that the time spent on OTs in all applications is practically negligible ($< 5\%$) in comparison with the time spent on circuit garbling.

In Figure 8 the performance gain of CGP is presented. The speed-up is measured in relation to the sequential total runtime. The timing results show that CGP scales almost linearly with up to 4 threads when using $\kappa = 80$ bit labels. Using $\kappa = 128$ bit labels, no further speed-up beyond 3 threads is noticeable. Thus, the impact of the network limits is immediately visible. Five ($\kappa = 80$ bit), respectively three ($\kappa = 128$ bit) threads are sufficient to saturate the available bandwidth in this experiment. Achieving further speed-ups is impossible without increasing the available bandwidth or developing new TPC techniques. However, to the best of our knowledge with 6M non-linear gates per second on a single core, as well as with approximately 32M non-linear gates per second on a single socket, we report the fastest garbling speed in

Circuits		BioMatch	MExp	MVMul
t_{total} [s]	$\kappa = 128$	2.71±0.02	1.43±0.01	0.20±0.00
	$\kappa = 80$	2.56±0.03	1.42±0.01	0.19±0.00
gps [M]	$\kappa = 128$	6.23±0.04	6.17±0.05	6.22±0.00
	$\kappa = 80$	6.56±0.07	6.21±0.04	6.43±0.00
bw [Gbps]	$\kappa = 128$	1.48±0.01	1.47±0.01	1.48±0.00
	$\kappa = 80$	0.97±0.01	0.92±0.01	0.95±0.00
t_{input} [s]	$\kappa = 128$	<0.02s	<0.01s	< 0.01s
	$\kappa = 80$	<0.02s	<0.01s	< 0.01s

Table 3: *Yao's protocol, single-core performance.* The run-time (t_{total}), non-linear gate throughput in million gates per second (*gps*), required bandwidth (*bw*) and time spent in the input phase (t_{input}), including the OTs when executing Yao's protocol for all applications in a LAN setting.

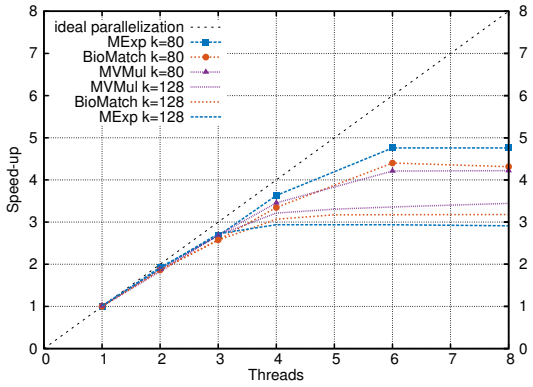


Figure 8: *Yao's protocol - CGP* The speed-up of all three applications in the LAN setting with $\kappa = 128$ bit and $\kappa = 80$ bit security.

an online setting of Yao's protocol. We abstain from an evaluation in a WAN setting due to the high bandwidth that is required to show the scalability of parallelization. The best bandwidth that we could observe during our experiments between two cloud regions was 350 Mbps, which is insufficient to benchmark parallel scalability.

6.5 Inter-party parallelization

A new application of parallelization in Yao's protocol is presented in § 5. We performed two experiments to show the applicability of IPP in practical settings. The first experiment measures the computational efficiency gain in the same setting as described in § 6.4. In the second experiment the benefits of IPP in a WAN setting with limited bandwidth are presented.

Computational efficiency gain. In this experiment the raw IPP performance for all example applications, as well as the combination of CGP and IPP techniques is explored. To realize IPP, our implementation uses multiple threads per core to utilize the load balancing capabilities of the underlying OS without implementing a sophisticated load balancer. Due to the heterogeneous hardware environment, e.g. unpredictable caching and networking behaviour, we evaluated three different workload distribution strategies. The first strategy uses one thread per core and thus only functions with at least two cores. Then, each party has exactly one garbling and one evaluating thread. The second and third strategy use two or four independent threads per core to garble and evaluate at the same time. Moreover, to illustrate that IPP is a modular concept, all circuits are evaluated using a sequential code block that exposes all inner input and output wires before and after every parallel region. This guarantees the evaluation of mixed functionalities. Consequently, all results include the time spent on transferring all required input bits to and from parallel regions. Otherwise applications such as the MVMul application, which is a pure parallel functionality, would profit more easily from IPP. Even though this weakens the results for the example applications, we are convinced that this procedure provides a better insight into the practical performance of IPP.

The results of this experiment are reported in Table 4. We first observe that only the MExp application significantly profits from IPP. This is due to the small sharing state in comparison to the circuit complexity. For both security levels IPP outperforms the raw CGP approach with an additional speed-up of 10-30% on all cores. The performance of the MVMul applications actually decreases when using IPP. This is because of the large state that needs to be transferred. The performance gain through IPP cannot overcome the newly introduced overhead of 31ms, which is more than 15% of the sequential run-time.

In summary, parallelizable applications that show a small switching surface (measured in number of bits compared to the overall circuit size) profit from IPP. Thus, IPP is a promising extension to Yao's protocol that utilizes circuit decomposition beyond naive parallelization, independently of other optimization techniques.

Bi-directional bandwidth exploitation. The second experiment aims towards increasing the available bandwidth by exploiting bidirectional data transfers. Commonly, Ethernet connections have support for full duplex (bi-directional) communication. When using standard Yao's garbled circuits, only one communication direction is fully utilized. However, with IPP the available bandwidth can be doubled by symmetrically exploiting both communication channels. This practical insight is evaluated in a WAN setting between two cloud instances of type `m3.xlarge` with $100 \pm 10ms$ latency and a measured bandwidth of 92 ± 27Mbps. Each hosts runs two threads (a garbling and a evaluating thread) using only

Environment		$\kappa = 80$						$\kappa = 128$					
		BioMatch		**MExp**		**MVMul**		**BioMatch**		**MExp**		**MVMul**	
Cores	**IPP**	time[s]	S	time[s]	S	time[s]	S	time[s]	S	time[s]	S	time[s]	S
1	none	2.559	1.000	1.423	1.000	0.192	1.000	2.712	1.000	1.485	1.000	0.199	1.000
	2	2.624	0.975	**1.287**	**1.106**	0.206	0.932	2.781	0.975	**1.370**	**1.084**	0.215	0.926
	4	2.556	1.001	**1.285**	**1.107**	0.208	0.923	2.707	1.002	**1.386**	**1.071**	0.218	0.913
2	none	1.384	1.849	0.734	1.939	0.103	1.864	1.497	1.812	0.780	1.904	0.108	1.844
	1	1.524	1.679	**0.686**	**2.074**	0.126	1.524	1.535	1.767	**0.699**	**2.124**	0.134	1.485
	2	1.472	1.738	**0.699**	**2.036**	0.132	1.455	1.516	1.789	**0.726**	**2.045**	0.137	1.453
	4	1.396	1.833	**0.654**	**2.176**	0.124	1.548	1.465	1.851	**0.693**	**2.143**	0.131	1.519
4	none	0.795	3.219	0.395	3.603	0.057	3.368	0.937	2.894	0.450	3.300	0.064	3.088
	1	0.996	2.569	0.426	3.340	0.084	2.286	1.041	2.605	0.452	3.285	0.087	2.287
	2	0.830	3.083	**0.336**	**4.235**	0.085	2.259	**0.874**	**3.103**	**0.356**	**4.171**	0.088	2.261
	4	0.818	3.128	**0.329**	**4.325**	0.081	2.370	**0.856**	**3.168**	**0.341**	**4.355**	0.084	2.369
8	none	0.652	3.925	0.298	4.775	0.045	4.267	0.872	3.110	0.364	4.080	0.048	4.189
	1	0.676	3.786	**0.239**	**5.954**	0.072	2.667	0.947	2.864	**0.303**	**4.901**	0.080	2.488
	2	**0.629**	**4.068**	**0.204**	**6.975**	0.077	2.494	**0.861**	**3.150**	0.342	**4.342**	0.075	2.653
	4	**0.636**	**4.024**	**0.233**	**6.107**	0.070	2.743	0.871	3.114	**0.337**	**4.407**	0.073	2.726
Transferring roles		0.231s		0.076s		0.031s		0.257s		0.082s		0.031s	

Table 4: Evaluation of IPP in a LAN setting. Column *IPP* specifies the number of threads used per core for load balancing. The total protocol run-time is measured in seconds and the speed-up in comparison with CGP is presented in column *S*. Marked in bold are settings, where IPP leads to performance gains. The time spent on the transferring roles protocol is presented in the last row.

a single core. The results of this experiment are illustrated in Table 5. IPP leads to significant speed-ups of BioMatch and MExp, showing the successful exploitation of bi-directional data transfers. MVMul shows limited performance gains because the time spent on the newly introduced communication rounds for the transferring roles protocol becomes significant. Summarizing, IPP can be very useful for TPC in bandwidth limited environments.

		BioMatch	**MExp**	**MVMul**
$\kappa = 128$	raw	45.02±0.49s	24.13±0.21s	4.83±0.05s
	IPP	29.94±0.31s	16.05±0.12s	4.66 ±0.35s
	S	1.50	1.50	1.03
$\kappa = 80$	raw	30.34±0.62s	14.56±0.21s	4.31±0.23s
	IPP	19.13±0.47s	11.16±0.32s	3.84±0.12s
	S	1.58	1.30	1.12

Table 5: Evaluation of IPP on a single core with limitted networking capabilities. Measured is the total protocol runtime, when sequentially (*raw*) computing and with IPP (*IPP*). Furthermore, the speed-up (*S*) between the two measurements is calculated.

7 Conclusion and Future Work

TPC based on Yao's garbled circuits protocol can greatly benefit from automatic parallelization. The FGP approach can be efficient for some circuits, yet its scalability highly depends on the circuit's width. The CGP approach shows a more efficient parallelization, given parallel decomposable applications. In contrast to previous work, a complete compile chain, which takes C code as input and automatically compiles parallel circuits, supports the practicability of our parallelization scheme. Moreover, we proposed the idea of IPP to achieve a symmetric workload distribution between two computing parties. With this technique, IPP achieves speed-ups through parallelization, even when using a single physical core. Concluding, in this work we presented an efficient, versatile and practical parallelization scheme for Yao's garbled circuits.

Further work includes the investigation of different parallel compilation targets for ParCC, such as the GMW protocol or RAM based secure computation frameworks. Also worthwhile for future investigations is the compilation of circuits optimized for FGP. Likewise, the application of IPP to other protocols is of interest.

8 Acknowledgements

We thank David Evans and all anonymous reviewers for their very helpful and constructive comments. This work has been co-funded by the German Federal Ministry of Education and Research (BMBF) within EC SPRIDE, by the DFG as part of project S5 within the CRC 1119 CROSSING, and by the Hessian LOEWE excellence initiative within CASED.

References

[1] AMINI, M., CREUSILLET, B., EVEN, S., KERYELL, R., GOUBIER, O., GUELTON, S., MCMAHON, J. O., PASQUIER, F.-X., PÉAN, G., AND VILLALON, P. Par4All: From Convex Array Regions to Heterogeneous Computing. In *Workshop on Polyhedral Compilation Techniques* (2012).

[2] ASHAROV, G., LINDELL, Y., SCHNEIDER, T., AND ZOHNER, M. More efficient oblivious transfer and extensions for faster secure computation. In *ACM Conference on Computer and Communications Security CCS* (2013).

[3] BARNI, M., BERNASCHI, M., LAZZERETTI, R., PIGNATA, T., AND SABELLICO, A. Parallel Implementation of GC-Based MPC Protocols in the Semi-Honest Setting. In *Data Privacy Management and Autonomous Spontaneous Security*. 2014.

[4] BEAVER, D., MICALI, S., AND ROGAWAY, P. The Round Complexity of Secure Protocols. In *ACM Symposium on Theory of Computing STOC* (1990).

[5] BELLARE, M., HOANG, V. T., KEELVEEDHI, S., AND ROGAWAY, P. Efficient garbling from a fixed-key blockcipher. In *IEEE Symposium on Security and Privacy S&P* (2013).

[6] BONDHUGULA, U., HARTONO, A., RAMANUJAM, J., AND SADAYAPPAN, P. A practical automatic polyhedral parallelizer and locality optimizer. *ACM SIGPLAN Notices 43*, 6 (2008).

[7] DAGUM, L., AND MENON, R. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science and Engineering 5*, 1 (1998).

[8] DEMMLER, D., SCHNEIDER, T., AND ZOHNER, M. ABY A Framework for Efficient Mixed-Protocol Secure Two-Party Computation. *Network and Distributed System Security NDSS* (2015).

[9] ERKIN, Z., FRANZ, M., GUAJARDO, J., KATZENBEISSER, S., LAGENDIJK, I., AND TOFT, T. Privacy-preserving face recognition. In *Privacy Enhancing Technologies PETS* (2009).

[10] FRANZ, M., HOLZER, A., KATZENBEISSER, S., SCHALLHART, C., AND VEITH, H. CBMC-GC: An ANSI C compiler for secure two-party computations. In *Compiler Construction CC* (2014).

[11] FREDERIKSEN, T. K., JAKOBSEN, T. P., AND NIELSEN, J. B. Faster maliciously secure two-party computation using the GPU. In *Security and Cryptography for Networks SCN*. 2014.

[12] HAZAY, C., AND LINDELL, Y. Efficient secure two-party protocols. *Information Security and Cryptography. Springer, Heidelberg* (2010).

[13] HENECKA, W., AND SCHNEIDER, T. Faster secure two-party computation with less memory. In *ACM Conference on Computer and Communications Security ASIACCS* (2013).

[14] HOLZER, A., FRANZ, M., KATZENBEISSER, S., AND VEITH, H. Secure Two-Party Computations in ANSI C. In *ACM Conference on Computer and Communications Security CCS* (2012).

[15] HUANG, Y., EVANS, D., KATZ, J., AND MALKA, L. Faster Secure Two-Party Computation Using Garbled Circuits. In *USENIX Security Symposium* (2011).

[16] HUANG, Y., KATZ, J., AND EVANS, D. Quid-pro-quo-tocols: Strengthening semi-honest protocols with dual execution. In *IEEE Symposium on Security and Privacy S&P* (2012).

[17] HUSTED, N., MYERS, S., SHELAT, A., AND GRUBBS, P. GPU and CPU parallelization of honest-but-curious secure two-party computation. In *Annual Computer Security Applications Conference ACSAC* (2013).

[18] ISHAI, Y., KILIAN, J., NISSIM, K., AND PETRANK, E. Extending Oblivious Transfer Efficiently. In *Advances in Cryptology CRYPTO*. Springer, 2003.

[19] JARECKI, S., AND SHMATIKOV, V. Efficient Two-Party Secure Computation on Committed Inputs. In *Advances in Cryptology EUROCRYPT*, vol. 4515. Springer, 2007.

[20] KERSCHBAUM, F., SCHNEIDER, T., AND SCHRÖPFER, A. Automatic protocol selection in secure two-party computations. In *Applied Cryptography and Network Security ACNS* (2014).

[21] KOLESNIKOV, V., AND SCHNEIDER, T. Improved garbled circuit: Free XOR gates and applications. In *International Conference on Automata, Languages and Programming ICALP*. 2008.

[22] KREUTER, B., MOOD, B., SHELAT, A., AND BUTLER, K. PCF: A Portable Circuit Format for Scalable Two-Party Secure Computation. In *USENIX Security Symposium* (2013).

[23] KREUTER, B., SHELAT, A., AND SHEN, C. Billion-Gate Secure Computation with Malicious Adversaries. *USENIX Security Symposium* (2012).

[24] LINDELL, Y., AND PINKAS, B. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In *Advances in Cryptology EUROCRYPT*. 2007.

[25] LINDELL, Y., AND PINKAS, B. A proof of security of yao's protocol for two-party computation. *Journal of Cryptology 22*, 2 (2009).

[26] MALEWICZ, G., AUSTERN, M. H., BIK, A. J. C., DEHNERT, J. C., HORN, I., LEISER, N., AND CZAJKOWSKI, G. Pregel: a system for large-scale graph processing. In *ACM Conference on Management of Data SIGMOD* (2010).

[27] MALKHI, D., NISAN, N., PINKAS, B., AND SELLA, Y. Fairplay-Secure Two-Party Computation System. In *USENIX Security Symposium* (2004).

[28] MCCREARY, C., AND GILL, H. Efficient Exploitation of Concurrency Using Graph Decomposition. In *International Conference on Parallel Processing ICPP* (1990).

[29] MOHASSEL, P., AND FRANKLIN, M. Efficiency tradeoffs for malicious two-party computation. In *Public Key Cryptography PKC*. 2006.

[30] NAYAK, K., WANG, X. S., IOANNIDIS, S., WEINSBERG, U., TAFT, N., AND SHI, E. GraphSC: Parallel Secure Computation Made Easy. In *IEEE Symposium on Security and Privacy S&P* (2015).

[31] PINKAS, B., SCHNEIDER, T., SMART, N. P., AND WILLIAMS, S. C. Secure two-party computation is practical. *Advances in Cryptology ASIACRYPT* (2009).

[32] POUCHET, L.-N. Polyhedral Compiler Collection (PoCC), 2012.

[33] REGEV, O. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM 56*, 6 (2009).

[34] YAO, A. C. Protocols for secure computations. In *Symposium on Foundations of Computer Science SFCS* (1982).

[35] YAO, A. C. How to generate and exchange secrets. In *Symposium on Foundations of Computer Science SFCS* (1986).

[36] ZAHUR, S., ROSULEK, M., AND EVANS, D. Two halves make a whole. *Advances in Cryptology - EUROCRYPT* (2015).

[37] ZHANG, Y., STEELE, A., AND BLANTON, M. PICCO: A general-purpose compiler for private distributed computation. In *ACM Conference on Computer and Communications Security CCS* (2013).