Combining Secret Sharing and Garbled Circuits for Efficient Private IEEE 754 Floating-Point Computations

Pille Pullonen[1,2] and Sander Siim[1,2]

[1] Cybernetica AS, Tartu, Estonia
[2] University of Tartu, Tartu, Estonia
{pille.pullonen, sander.siim}@cyber.ee

Abstract. Two of the major branches in secure multi-party computation research are secret sharing and garbled circuits. This work succeeds in combining these to enable seamlessly switching to the technique more efficient for the required functionality. As an example, we add garbled circuits based IEEE 754 floating-point numbers to a secret sharing environment achieving very high efficiency and the first, to our knowledge, fully IEEE 754 compliant secure floating-point implementation.

1 Introduction

Secure multi-party computation (MPC) enables parties to securely compute some function on their secret inputs and receive the secret outputs, without leaking anything to other parties. The fastest MPC protocols for integer arithmetic, like Sharemind [8][5] and SPDZ [10], rely on additive secret sharing. Additive sharing supports efficient addition and multiplication due to the algebraic properties of the scheme. However, floating-point arithmetic is much more sophisticated and contains a composition of different operations, both integer arithmetic as well as bitwise operations. Existing implementations based on secret sharing provide near approximations to the IEEE 754 standard [1][17][22]. Although [11] proposes IEEE 754 protocols, no implementation is provided.

Another MPC approach is based on the garbled circuits method (GC) attributed to Yao [29] and detailed in [21]. A good overview of the recent advances can be found in [3][2], especially in the full versions. The baseline method is applicable to the two-party setting, however it can be extended to the case with more parties [4]. State-of-the-art garbling methods are already very efficient [2] and, in addition, means to derive optimized circuits from existing programs have been developed [14][19]. This allows secure protocols for arbitrary computations

This research was, in part, funded by the U.S. Government. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government. This work has also received funding from the Estonian Research Council through grant IUT27-1, and ERDF through EXCS.

to be built with small effort using a general GC approach. However, in many cases the obtained protocols are less efficient than their secret-sharing-based alternatives. In practice, it would be useful to choose the more efficient technique, either secret sharing or GC, for each particular subprotocol, but this requires interleaving secret sharing and GC based protocols in one computation.

In this paper, we present a *hybrid protocol*, which enables arbitrary secure computations through a combination of GC and secret sharing protocols. In our protocol, GC gives the power to do bit-level operations in a compact manner, whereas secret sharing complements the construction with a fast oblivious transfer as well as composability with other secret-sharing-based protocols. Thereby, large and complex algorithms can be implemented by composition of the most efficient basic primitives. We illustrate the benefits by extending the Sharemind MPC framework [8][5] with, to our knowledge, the first secure floating-point protocol suite fully conforming to the IEEE 754 standard.

2 Preliminaries

In MPC, parties $\mathcal{P}_1, \ldots, \mathcal{P}_m$ want to securely compute a function f on secret inputs $x_1, \ldots, x_m$ to learn $f(x_1, \ldots, x_m) = (y_1, \ldots, y_m)$, without leaking anything about inputs x_i to other parties. Secret sharing is a mechanism of distributing data between participants without giving any of them direct access to the data, but enabling computations [27]. We denote a secret-shared vector $\mathbf{x}$ of n elements shared between parties $\mathcal{P}_1,\ldots,\mathcal{P}_m$ by $[\![\mathbf{x}]\!] = [\![x_1, x_2, \ldots, x_n]\!] = [\![x_1]\!], \ldots, [\![x_n]\!]$ where party $\mathcal{P}_i$ holds $[\![\mathbf{x}]\!]_i = [\![x_1, x_2, \ldots, x_n]\!]_i$. We focus on the additive secret sharing scheme, where sharing is defined with $\sum_{i=1}^{m} [\![x]\!]_i = x$. However, other schemes may also be used to implement our proposed protocol.

In the garbled circuits protocol [29][21], two parties called garbler and evaluator securely compute a known function $f(x, y)$ on their joint inputs. The garbler encrypts a Boolean circuit of $g = f(a, \cdot)$ and sends the *garbled* truth tables of gates to the evaluator. The evaluator then uses oblivious transfer to obtain the keys corresponding to its input to decrypt the garbled circuit and evaluate $g(b)$.

We use notation from [2] to describe circuits as a tuple $f = (n, m, q, A, B, G)$, where n, m and q respectively denote the number of external input wires, external output wires and gates in f. All wires are labelled by indexes. Namely, 1 to n are input wires, $n+1$ to $n+q$ mark gate output wires and $n+q-m+1$ to $n+q$ are circuit outputs. Functions A and B, respectively, identify the first and second input wire of any gate. For each gate g in f, the function $G(g) : \{0,1\}^2 \to \{0,1\}$ denotes the functionality of g. We use $X_j^b \in \{0,1\}^k$ to denote the token of the j-th wire corresponding to bit $b \in \{0,1\}$, where k is the length of the generated tokens. We say X_j^b has the *semantics* of b and type $\mathtt{lsb}(X_j^b)$.

Here we only emphasize the important aspects of the used security proof framework, for details we refer to [6]. A protocol is said to be *input private* if, for any collection of allowed corrupted parties, there exists a simulator that can simulate the view of the adversary based on the inputs of corrupted parties. The *ordered composition* of an input private and a secure protocol, where all outputs

Algorithm 1: Hybrid protocol for processing bitwise secret-shared data with a garbled circuit

Input: Shared bit vector $[\![\mathbf{x}]\!] = [\![x_1, \ldots, x_n]\!]$
Boolean circuit f that calculates $f : \{0,1\}^n \to \{0,1\}^m$
Output: Shared bit vector $[\![\mathbf{y}]\!] = [\![y_1, \ldots, y_m]\!]$ such that $[\![\mathbf{y}]\!] = f([\![\mathbf{x}]\!])$
1 **foreach** input wire $i \in \{1, \ldots, n\}$ **do**
2 $\quad \mathcal{CP}_1$ generates a token pair $(X_i^0, X_i^1) \in \{0,1\}^k \times \{0,1\}^k$
3 The computing parties initiate an OT protocol which results in $\mathcal{CP}_2$ receiving $\mathbf{X} = \{X_1^{x_1}, \ldots, X_n^{x_n}\}$ (the input tokens corresponding to the actual input bits)
4 $\mathcal{CP}_1$ garbles circuit f and sends the garbled truth tables to $\mathcal{CP}_2$
5 $\mathcal{CP}_2$ evaluates garbled f using $\mathbf{X}$ to get output tokens $\mathbf{Y} = \{X_{o_1}^{y_1}, \ldots, X_{o_m}^{y_m}\}$
6 The garbled output is converted to secret-shared form to receive $[\![\mathbf{y}]\!]$
7 **return** $[\![\mathbf{y}]\!]$

are provided by the secure protocol, is secure if it is *output predictable*. The latter means that the composed protocols are correct and the final protocol does not leak information about its input shares to ensure the privacy of the first part.

Garbled circuits have two important security definitions: privacy and obliviousness [3]. Respectively, we consider prv.ind, prv.sim and obv.ind, obv.sim for either indistinguishability or simulation based versions of these definitions. Both properties are formalised via the *side-information function* Φ that captures the information that is revealed by the garbled circuit. We consider Φ_{topo} and Φ_{xor} that leak the topology and XOR operations. These functions are both efficiently invertible [2]. Therefore, by equivalence relations from [3], indistinguishability and simulation-based definitions coincide for both privacy and obliviousness.

3 Combining Garbled Circuits with Secret Sharing

Our goal is to construct an efficient protocol for securely evaluating Boolean circuits on bitwise secret-shared input, thereby allowing secret sharing protocols to be composed with computations more suitable for GC. Thus, each subprotocol can use the method more suitable for the given functionality and inputs. A similar approach is also used in the TASTY framework that combines GC and additively homomorphic encryption in a two-party setting [13].

The idea of our protocol is to set up GC to accept secret-shared inputs and to produce shared outputs. Thus, our protocol in Alg. 1 consists of three important steps. First, we require an efficient oblivious garbling scheme to securely evaluate circuits. Suppose we have $\mathcal{CP}_1, \ldots, \mathcal{CP}_m$ who hold some secret-shared data. We will let computing parties $\mathcal{CP}_1$ and $\mathcal{CP}_2$ respectively perform the computations of garbler and evaluator from the GC protocol. Note that additionally to the properties of the secret sharing scheme we require that $\mathcal{CP}_1$ and $\mathcal{CP}_2$ are not colluding. Second, the GC protocol requires a special oblivious transfer (OT) to provide the input tokens to the evaluator from the secret-shared inputs. Third, we must convert the garbled outputs to the appropriate secret-shared form.

Algorithm 2: Oblivious transfer of input tokens (`OT`)

Input: $\mathcal{CP}_1$ holds the input tokens $\left\{X_1^0, \ldots, X_n^0, X_1^1, \ldots, X_n^1\right\}$
The input bit vector $[\![\mathbf{x}]\!] = [\![x_1, \ldots, x_n]\!]$ is shared between all parties
Output: $\mathcal{CP}_2$ receives input tokens $\{X_1^{x_1}, \ldots, X_n^{x_n}\}$

1 $[\![\mathbf{X^0}]\!] = [\![X_1^0, \ldots, X_n^0]\!]$ and $[\![\mathbf{X^1}]\!] = [\![X_1^1, \ldots, X_n^1]\!]$ are instantiated as shared values, with shares of $\mathcal{CP}_2$ and $\mathcal{CP}_3$ initialized to 0
2 $[\![\mathbf{X}]\!] \leftarrow [\![\mathbf{X^0}]\!] \cdot ([\![\mathbf{1}]\!] - [\![\mathbf{x}]\!]) + [\![\mathbf{X^1}]\!] \cdot [\![\mathbf{x}]\!]$
3 $\mathcal{CP}_1$ and $\mathcal{CP}_3$ send their shares of $[\![\mathbf{X}]\!]$ to $\mathcal{CP}_2$
4 $\mathcal{CP}_2$ combines the shares of $[\![\mathbf{X}]\!]$ to get $\{X_1^{x_1}, \ldots, X_n^{x_n}\}$
5 **return** $\{X_1^{x_1}, \ldots, X_n^{x_n}\}$

3.1 An Implementation of the Hybrid Protocol

Generally, the hybrid protocol can be implemented using various secret sharing and garbling schemes, provided they retain the security properties from Sec. 3.2 and conversion protocols in Alg. 1 exist. However, we will focus on our instantiation built into the Sharemind MPC platform [5]. We chose Sharemind because it already provides an optimized multi-party computation environment based on secret sharing, which could easily be extended with our GC based protocol.

Our protocol extends Sharemind's `additive3pp` protection domain, which implements various secure computation protocols using 3-out-of-3 additive secret sharing [7]. This allows us to easily compose the hybrid protocol with fast existing primitives for integer arithmetic. Note that different data types provided by Sharemind can be efficiently converted to shared bit vectors required in our construction [8]. Consequently, we fix a setting with three computing parties $\mathcal{CP}_1$, $\mathcal{CP}_2$ and $\mathcal{CP}_3$ and additive secret sharing. As with `additive3pp` protocols, our construction provides security against a single passively corrupted party.

Oblivious Transfer The garbler $\mathcal{CP}_1$ generates a pair (X_i^0, X_i^1) of tokens for every bit x_i in the beginning of the garbling process. We need to transfer the tokens that correspond to the protocol inputs to the evaluator $\mathcal{CP}_2$. Clearly, if we have a subprotocol that calculates the necessary secret-shared tokens $[\![X_1^{x_1}, \ldots, X_n^{x_n}]\!]$, then we can complete the transfer by sending all result shares to $\mathcal{CP}_2$. This subprotocol can be easily implemented using secret-sharing-based multiplication and addition protocols. The resulting `OT` protocol is given in Alg. 2. In addition to basic OT security properties, we also require that the inputs x_i are not leaked.

The computations on line 2 are performed using the secure and input private multiplication and addition protocols from [8]. As a result, $[\![\mathbf{X}]\!] = [\![X_1^{x_1}, \ldots, X_n^{x_n}]\!]$ and the inputs $[\![\mathbf{x}]\!]$ remain private. Note that each x_i can easily be extended to the length of the tokens, therefore the operations are performed in $\mathbb{Z}_{2^k}$. On line 3, the shares $[\![\mathbf{X}]\!]$ are sent to $\mathcal{CP}_2$ who combines them to receive the input tokens.

Garbling The emphasis of efficient GC is on reducing network communication, as this is the bottleneck for GC protocols. Recent garbling schemes bring the cost

of local computations to a minimum as demonstrated in [2]. Due to these considerations, we chose the GAXR scheme with the A4 DKC instantiation from [2] for our protocol, which is one of the fastest to date. Other garbling schemes could be used just as well, provided they retain the obliviousness property [3]. The GAXR scheme incorporates *free-XOR* [18] and *garbled row reduction* [24] optimizations, both of which significantly reduce network communication of GC.

The authors of [2][3] formalize the underlying encryption primitive of the garbling process as a *dual-key cipher*. The dual-key cipher used in the GAXR scheme is a function $\mathbb{E} : \{0,1\}^k \times \{0,1\}^k \times \{0,1\}^\tau \times \{0,1\}^k \to \{0,1\}^k$. It takes secret wire tokens A and B and a tweak T to encrypt a wire token X, resulting in a ciphertext $\mathbb{E}(A,B,T,X) = \pi(K \,\|\, T)_{[1:k]} \oplus K \oplus X$, where $K = 2A \oplus 4B$ and $X, A, B \in \{0,1\}^k$ and $T \in \{0,1\}^\tau$. Here $\pi(K \,\|\, T)_{[1:k]}$ denotes the first k bits of the result. The decryption is completely symmetric.

The function $\pi : \{0,1\}^{k+\tau} \to \{0,1\}^{k+\tau}$ denotes a random permutation, as the security of GAXR is shown in the *random permutation model*. We use a fixed-key AES-128 with $k = 80$ and $\tau = 48$ to instantiate π, which provides reasonable security guarantees for this garbling scheme [2]. Tweak T is the encrypted gate index encoded as a τ-bit integer. For the doubling function denoted by $2A$ we use multiplication with element x over finite field $GF(2^k)$, as it provides the best security guarantees over other possible alternatives [2]. Here k corresponds to bit-length of the wire tokens. Our implementation uses the irreducible polynomial $x^{80} + x^9 + x^4 + x^2 + 1$ from [26] for defining the finite field.

Fig. 1 summarizes the hybrid protocol. The garbler $\mathcal{CP}_1$ first generates a token pair (X_i^0, X_i^1) for each input wire, with X_i^0 and X_i^1 having the semantics of 0 and 1 respectively. Then all three computing parties synchronously execute the `OT` protocol in Alg. 2. As a result $\mathcal{CP}_2$ receives the correct input tokens needed for evaluation. Next, $\mathcal{CP}_1$ garbles the circuit according to the GAXR scheme and sends the garbled truth tables P to $\mathcal{CP}_2$. The evaluator $\mathcal{CP}_2$ can then evaluate the garbled circuit using the transferred input tokens to receive the garbled output.

As an implementation detail, we have parallelized our protocol on two levels. First, the garbled tables are streamed by fixed-size batches from garbler to evaluator, similarly to [15]. The evaluator can then start evaluating the circuit while the garbler encrypts the next batch. This is especially relevant performance-wise for large circuits. The batch size can be fixed for different circuits separately and fine-tuned to match the Sharemind instance's network and hardware capabilities.

In addition, our implementation allows both garbler and evaluator to run several threads to evaluate the same circuit with different inputs simultaneously. This can be thought of as using a number of garbler-evaluator pairs, similarly to the *cut-and-choose* implementation of [20] for actively secure GC. Besides parallel garbling, this allows a joint `OT` to be done for all the scheduled evaluations. This parallelization greatly reduces the cost of a single circuit evaluation.

Resharing The final step in the protocol is resharing the output between all three computing parties using perfectly secure `Reshare` protocol Alg. 1 from [5].

This protocol rerandomizes the output shares held by $\mathcal{CP}_1$ and $\mathcal{CP}_2$ as $\mathbf{y} = [\![\mathbf{y}]\!]'_1 + [\![\mathbf{y}]\!]'_2$ to a uniformly secret-shared output $[\![\mathbf{y}]\!]$ and ensures that we can securely compose our protocol with all `additive3pp` protocols, which is vital for efficient computations that would benefit from both GC and secret sharing.

Algorithm 3: Hybrid protocol algorithm for $\mathcal{CP}_1$

Input: $[\![\mathbf{x}]\!]_1 = [\![x_1, \ldots, x_n]\!]_1$ and circuit $f = (n, m, q, A, B, G)$
Output: $[\![\mathbf{y}]\!]_1 = [\![y_1, \ldots, y_m]\!]_1$ such that $[\![\mathbf{y}]\!] = f([\![\mathbf{x}]\!])$

$R \xleftarrow{\$} \{0,1\}^{k-1} \,\|\, 1$
for $i \leftarrow 1$ **to** n **do**
  $t \xleftarrow{\$} \{0,1\}$
  $X_i^0 \xleftarrow{\$} \{0,1\}^{k-1} \,\|\, t, \quad X_i^1 \leftarrow X_i^0 \oplus R$
`OT` $([X_1^0, \ldots, X_n^0], [X_1^1, \ldots, X_n^1], [\![\mathbf{x}]\!]_1)$
for $g \leftarrow n+1$ **to** $n+q$ **do**
  $a \leftarrow A(g), \quad b \leftarrow B(g)$
  if $G(g) = XOR$ **then**
    $X_g^0 \leftarrow X_a^0 \oplus X_b^0, \quad X_g^1 \leftarrow X_g^0 \oplus R$
  else
    for $i \leftarrow 0$ **to** $1, j \leftarrow 0$ **to** 1 **do**
      $u \leftarrow i \oplus \texttt{lsb}(X_a^0)$
      $v \leftarrow j \oplus \texttt{lsb}(X_b^0)$
      $r \leftarrow G(g, u, v)$
      if $i = 0$ **and** $j = 0$ **then**
        $X_g^r \leftarrow \mathbb{E}(X_a^u, X_b^v, g, 0^k)$
        $X_g^{r-1} \leftarrow X_g^r \oplus R$
      else
        $P[g, i, j] \leftarrow \mathbb{E}(X_a^u, X_b^v, g, X_g^r)$
Send P to $\mathcal{CP}_2$
for $i \leftarrow 1$ **to** m **do**
  $y'_{i1} \leftarrow \texttt{lsb}(X^0_{n+q-m+i})$
$[\![\mathbf{y}]\!]'_1 \leftarrow [y'_{11}, \ldots, y'_{m1}]$
$[\![y_1, \ldots, y_m]\!]_1 \leftarrow$ `Reshare`$([\![\mathbf{y}]\!]'_1)$
return $[\![\mathbf{y}]\!]_1 = [\![y_1, \ldots, y_m]\!]_1$

Algorithm 4: Hybrid protocol algorithm for $\mathcal{CP}_2$

Input: $[\![\mathbf{x}]\!]_2 = [\![x_1, \ldots, x_n]\!]_2$ and circuit $f = (n, m, q, A, B, G)$
Output: $[\![\mathbf{y}]\!]_2 = [\![y_1, \ldots, y_m]\!]_2$ such that $[\![\mathbf{y}]\!] = f([\![\mathbf{x}]\!])$

$[X_1, \ldots, X_n] \leftarrow$ `OT` $(0^{k \cdot n}, 0^{k \cdot n}, [\![\mathbf{x}]\!]_2)$
Receive P from $\mathcal{CP}_1$
for $g \leftarrow n+1$ **to** $n+q$ **do**
  $a \leftarrow A(g), \quad b \leftarrow B(g)$
  $i \leftarrow \texttt{lsb}(X_a), \quad j \leftarrow \texttt{lsb}(X_b)$
  if $G(g) = XOR$ **then**
    $X_g \leftarrow X_a \oplus X_b$
  else if $i = 0$ **and** $j = 0$ **then**
    $X_g \leftarrow \mathbb{E}(X_a, X_b, g, 0^k)$
  else
    $X_g \leftarrow \mathbb{D}(X_a, X_b, g, P[g, i, j])$
for $i \leftarrow 1$ **to** m **do**
  $y'_{i2} \leftarrow \texttt{lsb}(X_{n+q-m+i})$
$[\![\mathbf{y}]\!]'_2 \leftarrow [y'_{12}, \ldots, y'_{m2}]$
$[\![y_1, \ldots, y_m]\!]_2 \leftarrow$ `Reshare`$([\![\mathbf{y}]\!]'_2)$
return $[\![\mathbf{y}]\!]_2 = [\![y_1, \ldots, y_m]\!]_2$

Algorithm 5: Hybrid protocol algorithm for $\mathcal{CP}_3$

Input: $[\![\mathbf{x}]\!]_3 = [\![x_1, \ldots, x_n]\!]_3$
Output: $[\![\mathbf{y}]\!]_3 = [\![y_1, \ldots, y_m]\!]_3$ such that $[\![\mathbf{y}]\!] = f([\![\mathbf{x}]\!])$

`OT` $(0^{k \cdot n}, 0^{k \cdot n}, [\![\mathbf{x}]\!]_3)$
$[\![\mathbf{y}]\!]'_3 \leftarrow 0^m$
$[\![y_1, \ldots, y_m]\!]_3 \leftarrow$ `Reshare`$([\![\mathbf{y}]\!]'_3)$
return $[\![\mathbf{y}]\!]_3 = [\![y_1, \ldots, y_m]\!]_3$

Fig. 1. Detailed algorithms of the hybrid protocol for all computing parties.

3.2 Security of the Hybrid Protocol

Based on [6], we need to prove that the protocol up until the `Reshare` function is passively input private and then apply the composition result from [6]. For this, we also need to establish the output predictability of the composition. We

denote the part of the hybrid protocol on Fig. 1 before final `Reshare` protocol as `Hybrid'`. This section gives an overview of the important aspects of the proof, a full proof can be found in the full version of this paper [25]. Note, that the obliviousness of the garbling scheme [3] is quite like the input privacy [6] and is necessary for the input privacy of the `Hybrid'` protocol.

Theorem 1. *Ordered composition of `Hybrid'` and `Reshare` is output predictable.*

Proof (Proof sketch). Clearly, `Hybrid'` and `Reshare` are in ordered composition because all outputs of `Hybrid'` are inputs to `Reshare`. There is no data flow from `Reshare` to `Hybrid'`. The correctness of `Hybrid'` follows from the correctness of the sub-protocols used in the `OT` part and the correctness of the GaXR garbling scheme. Therefore, output predictability follows from Lemma 2 in [6].

Theorem 2. GaXR *scheme is computationally obv.ind and obv.sim secure.*

Proof (Proof sketch). The types of the input wires are independent of the semantics as they are generated independently on line 3 by $\mathcal{CP}_1$. In short, the obv.ind security follows from the fact that the keys of the outputs are generated the same way as the intermediate keys. Therefore, if there exists an adversary that breaks the obv.ind security for two functions f_1 and f_2 then this adversary can be extended to break the prv.ind security for two functions $c \circ f_1$ and $c \circ f_2$ for a constant function c. Finally, obv.ind security and obv.sim security coincide.

Theorem 3. *Protocol `Hybrid'` is perfectly input private for statically corrupted $\mathcal{CP}_1$ or $\mathcal{CP}_3$ and computationally input private against corrupted $\mathcal{CP}_2$.*

Proof (Proof sketch). We have to show the existence of the privacy simulator that can simulate the view of the corrupted party based on its inputs.

Corrupted $\mathcal{CP}_1$ or $\mathcal{CP}_3$. The only incoming communication for these protocols occurs during the `OT` computation phase. Therefore, the perfect input privacy of these parties is ensured by the perfect input privacy of the addition and multiplication protocol and the composability of input privacy (Thm. 3 in [6]).

Corrupted $\mathcal{CP}_2$. From obv.sim security in Thm. 2 we know that there exists a simulator $\mathcal{S}$ such that, for inputs $\mathcal{S}(1^k, \Phi(f))$, it outputs (F, X) indistinguishable from those output by the garbling scheme. This simulator is defined by the game ObvSim in [3]. The privacy simulator $\mathcal{P}$ for $\mathcal{CP}_2$ can be built from the simulator $\mathcal{S}$. This $\mathcal{P}$ knows the circuit f and also has the security parameter k, therefore, it can run $\mathcal{S}(1^k, \Phi(f))$ to obtain (F, X). Next, it has to simulate the `OT` that can be done perfectly by using the privacy simulator for the computation part and simulating the declassifying procedure with output X. All of the simulation, except for the choice of F and X, is perfect. Therefore, if the adversary gains any power to distinguish between the real life and simulation $\mathcal{P}$, it must result from the values F and X. However, this would invalidate the obv.sim security.

Corollary 4. *Hybrid protocol (Fig. 1) is perfectly secure against passively corrupted $\mathcal{CP}_1$ and $\mathcal{CP}_3$ and computationally secure against passively corrupted $\mathcal{CP}_2$.*

Proof. The composition is jointly output predictable (Thm. 1) and `Hybrid'`, first part of the ordered composition, is input private (Thm. 3). Using the composition result (Thm. 2 in [6]) we conclude that the full hybrid protocol is secure.

4 Using the Hybrid Protocol for Efficient Computations

Sharemind's `additive3pp` protocols enable fast integer operations. On the other hand, bit-level operations are more costly. However, in practical applications we are also interested in more complex primitives that rely heavily on bit-level operations. A very relevant example of this is floating-point computations.

The *de facto* standard today for binary floating-point arithmetic is IEEE 754 [16]. Although existing secure implementations of floating-point operations resemble IEEE 754 [1][17][22], they do not always produce identical results compared to regular hardware implementations. The main shortcomings are in not rounding inexact results to nearest representable floating-point numbers, lack of support for gradual underflow and missing error handling [12].

Using the CBMC-GC circuit compiler [14] (v.0.9.3 [9]), we were able to implement an efficient and fully IEEE 754 compliant floating-point protocol suite based on our hybrid protocol. The CBMC-GC compiler transforms C programs directly to highly optimized circuits usable in a GC protocol. This allowed us to use exact IEEE 754 software implementations as a basis for our protocols.

We implemented both single and double precision secret-shared floating-point data types. The `float` and `double` types are represented as 32-bit and 64-bit bitwise secret-shared integers that correspond exactly to the IEEE 754 standard. Our construction guarantees bit-by-bit identical results to those of regular hardware floating-point procedures, excluding non-standardized details such as the significand bits of a NaN. We empirically verified this claim for the four arithmetic operations and square root on a machine with Intel Core i7-870 2.93 GHz processor against equivalent C programs compiled with GCC 4.8.1-2.

4.1 Circuits for IEEE 754 Primitives

The circuits used in our protocol suite are listed and described in Table 1. We list circuit sizes as well as the number of garbled tables batches sent during one evaluation of the circuit. The circuits were compiled on a workstation with 16 GB RAM and an Intel Core i7-870 2.93 GHz processor. Although it would have much reduced the circuit sizes, we were unable to use the SAT-minimization functionality of CBMC-GC for larger circuits due to high compilation times.

We used the efficient SoftFloat [28] IEEE 754 software implementation for compiling addition, multiplication, division and square root circuits. We additionally used musl libc [23] for double precision e^x and error function (erf) as an example of more complex operations and the flexibility of our approach to implement arbitrary primitives. Only minor syntactic modifications of the source code were required to compile it with CBMC-GC. We hardcoded rounding to the default "Round to nearest even" mode defined in IEEE 754, since this is most used in practice and provides the best bounds on rounding errors [12]. Alternatively, we could give the rounding mode as input to the circuit.

We chose to ignore all floating-point exceptions that may be raised during computations, since in an MPC environment, raising an exception (e.g. division by zero) in the middle of a computation can possibly leak information about

Table 1. IEEE 754 floating-point operation circuits compiled with CBMC-GC

Circuit	Non-XOR Gates	Total Gates	No of Batches	Used SAT-minimization	Compilation Time
float_add	5671	7052	1	+	8 min 32 s
float_sub	5671	7052	1	+	5 min 28 s
float_mul	5138	7701	1	+	5 min 1 s
float_div	12851	21384	1	-	58 s
float_sqrt	35987	66003	2	-	2 min 40 s
double_add	13129	15882	1	+	1 h 8 min
double_sub	13129	15882	1	+	1 h 11 min
double_mul	13104	25276	1	+	3 h 46 min
double_div	36133	73684	2	-	5 min 35 s
double_sqrt	85975	169932	4	-	10 min 11 s
double_exp	393807	579281	8	-	1 h 13 min
double_erf	2585188	3979603	52	-	47 h 4 min

inputs. As our protocols correctly handle all special cases defined in the standard (NaNs, infinities, denormalized numbers), any exceptions will be reflected in the final result. Previous implementations [1][17][22] did not explicitly handle such cases and produced valid but meaningless results in error situations.

4.2 Performance Analysis

We benchmarked the performance of our implemented IEEE 754 primitives as well as the existing floating-point operations [17] in Sharemind for comparison. The benchmarks were performed on a cluster of three nodes hosting Sharemind. All nodes had 48 GB of RAM and a 12-core 3GHz Intel CPU supporting AES-NI and HyperThreading. The nodes were connected to a LAN with 1 Gbps full duplex links. All tests were executed with a maximum of 24 concurrent garbler-evaluator pairs, as the hardware supports up to 24 parallel threads.

The performance results are shown in Table 2 for single precision and Table 3 for double precision. All measurements are presented in operations per second (ops) as the mean of 5 to 1000 iterations depending on the circuit size. The measurements depict the whole running time of the protocol including oblivious transfer, garbling and evaluation. Circuits are parsed and cached in an offline phase, however. The input size refers to the number of respective operations computed in one test using the parallelization techniques described in Sec. 3.1.

Our measurements show that hybrid protocol IEEE 754 operations, excluding error function, are faster than approximation-based operations for smaller input sizes. The error function clearly illustrates the substantial overhead for evaluating very large circuits, thereby motivating the composition of small but efficient primitives as opposed to full circuit programs. The IEEE 754 division and square root perform very well compared to approximation-based versions, whereas the error function and multiplication are slower on larger input sizes. Our protocols well outperform the results from [22] and our double precision addition and

Table 2. Performance of single precision floating-point operations (ops)

		Input size in elements				
		1	**10**	**100**	**1000**	**10000**
Add	Approx.	2.43	24.1	228.1	1496	3790
	IEEE 754	24.99	134.5	477.2	583.6	597
Multiply	Approx.	7.76	77.94	751.6	5413	16 830
	IEEE 754	26.17	135.5	506	632.9	632.9
Divide	Approx.	0.53	5.25	46.48	237	432.6
	IEEE 754	14.53	88.2	233.6	279.1	284.5
Square root	Approx.	0.34	3.26	28.07	126.1	206.1
	IEEE 754	7.83	44	92.9	105.1	106.6

division are faster than the implementation of [1], however, multiplication is slightly slower. The latter is expected, since it is efficient to implement floating-point multiplication using secret sharing and less is gained from a GC approach.

The results also show that IEEE 754 operations do not benefit much from parallelization already after inputs of size ~100, while the approximation-based operations parallelize well to 10000 elements. This is due to the large size of the garbled tables that are transmitted over the network. In all larger tests with the IEEE 754 operations, the network link was constantly saturated, which introduced an inevitable upper bound on performance. This demonstrates the tradeoff between GC and secret sharing, as GC generally requires more network communication, but has better round-complexity. For example, in our instantiation, the garbled circuit for `float` addition has size ~175 KB, whereas the approximation-based protocol uses at most 12 KB of one-way network communication over a series of communication rounds. Consequently, the sharing-based protocols have better amortized performance for larger inputs. In practice, the input size can be used to dynamically choose between GC or sharing-based protocols.

The IEEE 754 protocols used significantly more memory and processing power, as all processor cores of the garbler node were nearly constantly working at maximum capacity. The effect of the high-speed parallel garbling on overall performance was nevertheless ultimately dominated by the network bandwidth, suggesting that less powerful hardware could be used for similar results. The approximation-based counterparts used only ~10% of the hardware capability.

5 Conclusion

This work provided a protocol for combining GC with secret sharing. For this we consider a setting where the oblivious transfer for the garbled evaluation inputs can work for secret-shared inputs rather than the inputs known to the evaluator. In addition, it is required that the outputs of the garbled evaluation remain private. This allows us to combine the strengths of both approaches. Especially, efficient secret-sharing-based computation protocols can be augmented with easily generated GC based protocols for the functionalities where no known efficient

Table 3. Performance of double precision floating-point operations (ops)

		Input size in elements			
		1	**10**	**100**	**1000**
Add	Approx.	2.29	22.22	188.2	857.7
	IEEE 754	16	103	228	260
Multiply	Approx.	7.17	71.98	647.9	3560
	IEEE 754	13.74	90.8	221	259
Divide	Approx.	0.5	4.78	35.22	115.7
	IEEE 754	7.31	46	89.2	101
Square root	Approx.	0.26	2.4	14.23	31
	IEEE 754	3.57	23.3	39.5	43.4
e^x	Approx.	0.28	2.57	14.7	31.1
	IEEE 754	1.1	6.38	9	9.5
Error function	Approx.	0.3	2.92	19.8	55.4
	IEEE 754	0.18	0.95	1.35	1.47

sharing-based protocol exists. As an example, we added a very efficient first fully IEEE 754 compliant secure floating-point implementation to Sharemind.

Acknowledgments We would like to thank the authors of the CBMC-GC circuit compiler for supporting us in our efforts to generate the described circuits.

References

[1] Aliasgari, M., Blanton, M., Zhang, Y., Steele, A.: Secure computation on floating point numbers. In: Proc. of NDSS'13. The Internet Society (2013)
[2] Bellare, M., Hoang, V.T., Keelveedhi, S., Rogaway, P.: Efficient garbling from a fixed-key blockcipher. In: Proc. of SP'13. pp. 478–492. IEEE Computer Society, Washington, DC, USA (2013)
[3] Bellare, M., Hoang, V.T., Rogaway, P.: Foundations of garbled circuits. In: Proc. of CCS'12. pp. 784–796. ACM, New York, USA (2012)
[4] Ben-David, A., Nisan, N., Pinkas, B.: FairplayMP: a system for secure multi-party computation. In: Proc. of CCS'08. pp. 257–266. ACM (2008)
[5] Bogdanov, D.: Sharemind: programmable secure computations with practical applications. Ph.D. thesis, University of Tartu (2013)
[6] Bogdanov, D., Laud, P., Laur, S., Pullonen, P.: From input private to universally composable secure multi-party computation. In: Proc. of CSF'14. IEEE Computer Society (2014)
[7] Bogdanov, D., Laud, P., Randmets, J.: Domain-polymorphic programming of privacy-preserving applications. In: Proc. of PETShop'13. pp. 23–26. ACM (2013)
[8] Bogdanov, D., Niitsoo, M., Toft, T., Willemson, J.: High-performance secure multi-party computation for data mining applications. IJIS 11(6), 403–418 (2012)

[9] CBMC-GC. http://forsyte.at/software/cbmc-gc/
[10] Damgård, I., Pastro, V., Smart, N.P., Zakarias, S.: Multiparty computation from somewhat homomorphic encryption. In: Proc. of CRYPTO 2012. LNCS, vol. 7417, pp. 643–662. Springer (2012)
[11] Franz, M., Katzenbeisser, S.: Processing encrypted floating point signals. In: Proc. of MM&Sec'11. pp. 103–108. ACM, New York, NY, USA (2011)
[12] Goldberg, D.: What every computer scientist should know about floating-point arithmetic. ACM Computing Surveys 23(1), 5–48 (1991)
[13] Henecka, W., Kögl, S., Sadeghi, A.R., Schneider, T., Wehrenberg, I.: TASTY: Tool for automating secure two-party computations. In: Proc. of CCS'10. pp. 451–462. ACM, New York, NY, USA (2010)
[14] Holzer, A., Franz, M., Katzenbeisser, S., Veith, H.: Secure two-party computations in ANSI C. In: Proc. of CCS'12. pp. 772–783. ACM (2012)
[15] Huang, Y., Evans, D., Katz, J., Malka, L.: Faster secure two-party computation using garbled circuits. In: Proc. of SEC'11. USENIX Association (2011)
[16] 754-2008 - IEEE standard for floating-point arithmetic. http://ieeexplore.ieee.org/servlet/opac?punumber=4610933 (2008)
[17] Kamm, L., Willemson, J.: Secure floating-point arithmetic and private satellite collision analysis. IJIS (2014)
[18] Kolesnikov, V., Schneider, T.: Improved garbled circuit: Free XOR gates and applications. In: Proc. of ICALP (2). LNCS, vol. 5126, pp. 486–498. Springer (2008)
[19] Kreuter, B., Mood, B., Shelat, A., Butler, K.: PCF: A portable circuit format for scalable two-party secure computation. In: Proc. of SEC'13. pp. 321–336. USENIX Association, Berkeley, CA, USA (2013)
[20] Kreuter, B., Shelat, A., Shen, C.: Billion-gate secure computation with malicious adversaries. In: Proc. of Security'12. USENIX Association (2012)
[21] Lindell, Y., Pinkas, B.: A proof of security of Yao's protocol for two-party computation. J. Cryptology 22(2), 161–188 (2009)
[22] Liu, Y.C., Chiang, Y.T., Hsu, T.S., Liau, C.J., Wang, D.W.: Floating point arithmetic protocols for constructing secure data analysis application. Procedia Computer Science 22(0), 152 – 161 (2013)
[23] musl libc. http://www.musl-libc.org/
[24] Pinkas, B., Schneider, T., Smart, N.P., Williams, S.C.: Secure two-party computation is practical. In: Proc. of ASIACRYPT 2009. LNCS, vol. 5912, pp. 250–267. Springer (2009)
[25] Pullonen, P., Siim, S.: Combining secret sharing and garbled circuits for efficient private IEEE 754 floating-point computations. Cryptology ePrint Archive, Report 2014/990 (2014)
[26] Seroussi, G.: Table of low-weight binary irreducible polynomials. http://www.hpl.hp.com/techreports/98/HPL-98-135.html (1998)
[27] Shamir, A.: How to share a secret. Commun. ACM 22(11), 612–613 (1979)
[28] SoftFloat. http://www.jhauser.us/arithmetic/SoftFloat.html
[29] Yao, A.C.: Protocols for secure computations. In: Proc. of SFCS'82. pp. 160–164. IEEE Computer Society, Washington, DC, USA (1982)