Bit Decomposition Protocols in Secure Multiparty Computation

Peeter Laud
Cybernetica AS
Tartu, Estonia
peeter@cyber.ee

Alisa Pankova
Cybernetica AS, STACC
Tartu, Estonia
alisa@cyber.ee

ABSTRACT

We present improved protocols for the conversion of secret-shared bit-vectors into secret-shared integers and vice versa, for the use as subroutines in secure multiparty computation (SMC) protocols and for protocols verifying the adherence of parties to prescribed SMC protocols. The protocols are primarily designed for three-party computation with honest majority. We evaluate our protocols as part of the Sharemind three-party protocol set and see a general reduction of verification overheads, thereby increasing the practicality of covertly or actively secure Sharemind protocols.

CCS CONCEPTS

• **Security and privacy** → *Formal methods and theory of security*;

KEYWORDS

Secure multiparty computation; verifiable computation; covert adversary

ACM Reference Format:
Peeter Laud and Alisa Pankova. 2018. Bit Decomposition Protocols in Secure Multiparty Computation. In *WAHC '18: 6th Workshop on Encrypted Computing & Applied Homomorphic Cryptography, Oct. 19, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/XXXXXX.XXXXXX

1 INTRODUCTION

Secure Multiparty Computation (SMC) allows a number of mutually distrustful participants to perform computations over the union of their data, such that no participant or coalition (under a certain size) learns anything about other parties' data [22]. Presently, there exist a large number of protocol sets and implementations differing from each other in used cryptographic techniques, security guarantees offered to different parties, symmetry of participant roles, communication and round complexity, relative complexity of different operations with respect to each other, etc. See e.g. [31] for an overview.

In this paper we focus on one kind of SMC protocols — those based on secret sharing among a few (two or three) *computing parties*, who may receive the inputs of the computation in secret-shared manner from a large number of *input parties* and deliver the shares of results to any number of *result parties*. A number of platforms are using this kind of protocols, including ABY [19], Sharemind [7], SEPIA [9], MEVAL [26], VIFF [13], MASCOT [24], etc. The sharing can be either Shamir's [43] or additive (the sum of the shares in some ring equals the secret value), most recent work has been on additive sharing. A platform may offer several rings over which the secrets may be shared; different rings have different associated performance profiles for the operations with secrets. E.g. sharing over a ring with large characteristic offers efficient arithmetic (addition and multiplication), while sharing each bit separately (we call such sharing over $\mathbb{Z}_2$ *XOR-sharing* in this paper) offers efficient comparisons. Hence there is a need to convert secret-shared values between different rings. A complex protocol (e.g. floating-point addition) may internally perform many such conversions.

To achieve security against stronger than passive adversaries, some verification mechanisms need to be built into the SMC protocols. There are different kinds of mechanisms available, discussed in Sec. 2 below. In this paper, we consider mechanisms that are again based on secret sharing. Recently, Laud et al. [30] have proposed an efficient method for implementing the GMW compiler [22] that turns passively secure protocols to actively secure ones by making the parties prove in zero-knowledge that they have followed the protocol. Here the proof consists of the party secret-sharing its private values among other parties who will then repeat its computation as a SMC protocol, with the proving party's assistance. These verification steps have their own performance profile, with e.g. the depth of the computation being irrelevant. Here again the verifying parties may need to convert between an additively shared and XOR-shared version of prover's private value.

In this paper we propose and discuss conversion protocols between additively and XOR-shared integers (or: between values shared as integers and values shared as bit-vectors), both for private computation and for the verification of the observance of the protocol. We discuss the performance of these protocols in the context of Sharemind's protocol set [7, 20, 27, 33], showing how they improve certain aspects of them, particularly their communication costs. We also discuss these protocols in the context of verifiable SMC, improving the verification overheads reported in [30] for the protocols of Sharemind. We note that the ideas discussed here have been mentioned before in some form [19, 32], but have not been followed through in the settings of this paper (discussed further in Sec. 2).

WAHC '18, October 19, 2018, Toronto, ON, Canada

ACM ISBN 978-1-4503-5987-0/18/10.
https://doi.org/10.1145/XXXXXX.XXXXXX

After reviewing related work in Sec. 2 we describe the protocol set of Sharemind and its verification in Sec. 3 and explain the parameters we strive to optimize. In Sec. 4 we describe our protocols and in Sec. 5 provide an evaluation of their benefits. We conclude in Sec. 6.

2 RELATED WORK

Conversion between integers and bit-vectors. For secret-sharing based SMC protocol sets, where a private value is normally stored as a secret-shared integer, one often wants to obtain sharings of single bits of that value in order to perform some relational operations — equality or inequality checks, or oblivious choice. Such *bit-decomposition*, also wanted for SMC protocol sets based on homomorphic encryption, has received a fair amount of attention, starting from Algesheimer et al. [1]. Work on more efficient protocols has continued over the years [12, 40, 42, 44], sometimes in parallel to eliminating the bit-decomposition step from certain protocols [37]. Bit decomposition protocols have been proposed as subprotocols for more complex private computations [10]. SMC protocols and frameworks that use both homomorphic encryption and garbled circuits typically need bit decomposition [8, 23]. Interestingly, ABY [19] does not have a bit-decomposition protocol from additively to XOR-shared values, instead going through garbled circuits.

The other direction — going from a vector of secret-shared bits to an additively shared integer — has received much less attention. This is likely because the bit-decomposition operation does not change the ring over which the sharing is done: each bit is just a shared integer with the value 0 or 1. In this case the backwards conversion is merely a trivial linear combination. Converting a bit-vector to an integer is meaningful only if sharings over different rings are involved. This is the case for Sharemind [7] and for ABY [19], both of which convert each bit separately to a shared integer and then compute their linear combination. Demmler et al. [19] passingly mention the idea we explore in Sec. 4, only to immediately dismiss it and focus on converting single bits.

When verifying that a party has followed the protocol, the direction of the computation does not matter [30]. Instead, the proving party needs to show that a shared bit-vector is equal to a shared integer. They propose to convert each bit of the bit-vector to an integer and check that their linear combination is equal to the shared integer.

Covert and actively secure multiparty computation based on secret sharing. Actively secure SMC is a broad research area and we cannot hope to give an overview of all of it in this section. We therefore constrain ourselves to secret-shared based SMC protocols and leave out garbled circuits. In all of these approaches, the protocol is in some sense executed many times, and the values coming from different executions are compared to each other. The approaches differ on whether the comparison is done (1) at the runtime or (2) after it, and whether different executions are (a) similar to each other or (b) not. The approaches also differ on whether they achieve active or covert [2] security, and, in the former case, whether a malicious party can be pinpointed.

The approach (1a) is present in the basic Shamir's secret sharing [43] based actively secure SMC [11]. It is also present in the schemes of Furukawa, Lindell et al. [21, 35], where redundancy is built into the sharing construction, and in the scheme of Damgård et al. [16] where messages are repeated.

The approach (1b) is perhaps most well-known, where the messages are accompanied with some sort of MACs which are also subjected to the computation and verified at each round. SPDZ [15, 17, 18] is the best-known protocol following this approach, but there are others, based e.g. on actively secure oblivious transfer [36].

The approach (2a) can often be obtained from a protocol following (1a), if the checks are moved to the end of the protocol. Depending on the checked protocol, this may reduce the security from active to covert. This approach may also very cheaply turn a passively secure protocol into a covertly secure one [14].

The post-execution verification that the party (or parties) followed the protocol, possibly with assistance from these parties, is the heart of approach (2b). This approach is followed by Laud et al. [29, 30], where the local computations of a party are repeated by the verifiers who have a secret-shared copy of that party's local state. It is also followed by Baum et al. [3], where anyone with access to the transcript of the protocol can check that the output is indeed correct.

3 PROTOCOLS OF SHAREMIND

In this section, we discuss in more details Sharemind SMC platform [6]. We explain how its passively secure protocol set has been extended to covertly and actively secure.

3.1 Passively secure protocols

The main protocol set of Sharemind is based on sharing data among three computing parties, tolerating one passively corrupted party (i.e an honest majority assumption). In other words, no party in the three-party set is able to infer any private information, as far as it does not form coalition with one of the other two parties. Although two parties would already be sufficient to hold a shared secret, assistance of the third party allows to get much more efficient protocols.

Sharemind protocol set deploys both additive and XOR-sharing, over finite rings $\mathbb{Z}_{2^m}$ and $\mathbb{Z}_2^m$ respectively. Additive and XOR-sharing can be mixed in the same application, and the protocols implementing transitions between $\mathbb{Z}_{2^m}$ and $\mathbb{Z}_2^m$ play a very important role. Sharemind derives its efficiency from the great variety of protocols [7, 25, 27, 34] for integer, fix- and floating point operations, as well as for shuffling the arrays.

Sharemind has a tool for automatic compilation of lower-level protocols from a specialized domain-specific language [32]. The protocols are presented as a clear description of how messages are computed and exchanged between parties. There are over 100 such composable lower-level protocols, used as building blocks for larger applications. During compilation, the protocols undergo an intermediate format that represents the *local* computation of each party as an arithmetic circuit. These circuits in general consist of addition and multiplication operations, defined over rings $\mathbb{Z}_{2^n}$. One circuit may be defined over several rings, so there are also transitions between rings $\mathbb{Z}_{2^m}$ and $\mathbb{Z}_{2^n}$. The circuits also contain bitwise AND, OR, XOR operations, which are reduced to addition and multiplication over $\mathbb{Z}_2$, introducing *implicit* transitions between $\mathbb{Z}_{2^n}$ and $\mathbb{Z}_2^n$.

A privacy-preserving application is described in a high-level programming language (called SecreC) that is compiled into bytecode [5]. It instructs the Sharemind virtual machine (VM) to call the compiled lower-level protocols in certain order with certain arguments. Combining the application bytecode with the lower-level protocols that it uses, we get a full description of local computation performed by each party in that application.

3.2 Verifiable protocols

We now briefly repeat the results of [30], which describe how *passively* secure Sharemind protocols can be extended to *covertly* and *actively* secure. An important property of Sharemind protocols is that, as far as no intermediate declassifications take place, the main protocol set of Sharemind is *actively private* [38], i.e. no party can infer any information about the secrets of the other parties, even if it tampers with the computation. Therefore, it suffices to verify the computation only before some value gets declassified, which often happens in the end, when the shares of final outputs are already available.

The main idea is to verify the computation of *each* party separately. Hence, the *local* computation of a party should be verified, and its description can be extracted from a Sharemind application as described in Sec. 3.1. This approach allows to identify the cheater. Moreover, it is more flexible with respect to operations being verified, since the prover party (P) may give helpful hints to the other two parties serving as verifiers (V_1 and V_2), and these hints are easier to verify than to compute.

The entire computation can be split into preprocessing, execution, and verification phases. We now describe this process step by step.

Preprocessing. The parties compute sufficient number of *precomputed tuples*, i.e. random numbers that are correlated in a certain way. They will be used only in the verification phase, to verify a particular party's computation. As a prover, each party gets his own personal set of tuples. In particular, the prover generates his own tuples himself, and then additively shares them among the the two other parties, who will later serve as verifiers. Correctness of these tuples is immediately verified, and it is achieved by generating more tuples than necessary and sacrificing some of them to verify the others. Two main types of tuples are:

- The *trusted multiplication triples* are triples (a, b, c) from some ring $\mathbb{Z}_{2^m}$ (including $m = 1$), such that $a \cdot b = c$.
- The *trusted bits* are values b from some ring $\mathbb{Z}_{2^m}$, $m > 1$, such that $b \in \{0, 1\}$.

Execution. The parties execute a passively secure protocol without any changes. To make further verification possible, each party needs to get "committed" to its inputs and the messages that it has sent or received. No cryptographic commitment schemes are actually used, and all relevant values are merely additively shared among the two other parties.

- *Inputs:* At the beginning of execution phase, each party P shares its input x as $x = x_1 + x_2$ among the two other parties V_1 and V_2.
- *Randomness:* Sharemind protocols are constructed in such a way that each piece of randomness r of P is also known either to V_1 or V_2, and it can be viewed as being additively shared as $r = r + 0$ or $r = 0 + r$. The randomness of P and V_i is only needed to hide data from the third party V_j, and choosing it in a bad way does not give any benefits to P.
- *Messages:* In three-party computation, each message m that has been sent or received by P is known at least to one other party V_1 or V_2 that has been on the other side of the communication. Any message m can be viewed as being additively shared among the verifiers as $m = m+0$ or $m = 0 + m$.

Verification. Treating each party as a prover P, the two other parties V_1 and V_2 repeat the computation of P from committed inputs, and see if they get committed outputs in the end. They execute two-party computation, which is assisted by P. In verification setting, this task is much easier, since P already knows all the data that both verifiers have, and may give them hints. It is sufficient for verifiers to compute addition, multiplication, and conversion between $\mathbb{Z}_{2^n}$ and $\mathbb{Z}_2^n$, although there exist special, more efficient verification methods for higher-level operations. These basic operations can be computed by the verifiers as follows:

- All additions are computed locally, since all values are additively shared.
- A multiplication $z = x \cdot y$ is computed using a precomputed triple $c = a \cdot b$ as $z = x \cdot (y - b) + b \cdot (x - a) + c$, where $y' = y - b$ and $x' = x - a$ are computed and published by the prover. The verifiers postpone the checks $x - a - x' = 0$ and $y - b - y' = 0$.
- A ring conversion between $\mathbb{Z}_{2^n}$ and $\mathbb{Z}_2^n$ is computed using n trusted bits $b_0, \ldots, b_{n-1} \in \mathbb{Z}_{2^n}$. Suppose that the verifiers hold a value x, either additively shared over $x \in \mathbb{Z}_{2^n}$, or XOR-shared over $\mathbb{Z}_2^n$. The prover knows the exact value of x and is able to publish the bits $c_0, \ldots, c_{n-1}$ such that, for all i, $c_i \oplus b_i \bmod 2 = x_i$, where x_i is the i-th bit of x, and $\oplus$ denotes XOR operation. For all i, the verifiers may now compute $y_i = b_i$ if $c_i = 0$ and $y_i = 1 - b_i$ otherwise. They take $y = \sum_{i=0}^{n-1} 2^i y_i \in \mathbb{Z}_{2^n}$ and postpone the checks $x_i \oplus c_i \oplus b_i \bmod 2 = 0$ if they need additive sharing of x. They take $[y_0 \bmod 2, \ldots, y_{n-1} \bmod 2]$ and postpone the check $x - \sum_{i=0}^{n-1} 2^i y_i = 0$ if they need XOR-sharing of x.

- For each outgoing message of P, postpone the check $y - y' = 0$, where y' is the committed output and y is the value reconstructed by the verifiers themselves using previous steps.

Finally, the verifiers check $[z_1, \ldots, z_s] = [0, \ldots, 0]$ succinctly for all checks postponed during the verification. At this point, the verifiers are holding shares $[z_1^1, \ldots, z_s^1]$ and $[z_1^2, \ldots, z_s^2]$, such that $[z_1^1 + z_1^2, \ldots, z_s^1 + z_s^2] = [z_1, \ldots, z_s]$. They take collision-resistant hash function h (e.g. SHA-256), compute $h([z_1^1, \ldots, z_s^1])$ and $h([-z_1^2, \ldots, -z_s^2])$, and exchange these hashes. The verification has passed iff these hashes are equal.

To achieve accountability or covert security, we may want to identify who exactly was cheating. In that case, the same verification mechanism can be used, only that all the shares that are issued to the verifiers (including shares of precomputed tuples) need to be signed. If the hashes in the end do not match, the prover has right to accuse one of the verifiers V_j with whom he does not agree (at most one hash is wrong since there is at most one corrupted party), and all signed shares of V_j are revealed to the other verifier V_k, who may now repeat computation of V_j and find out who is guilty.

4 THE CONVERSION PROTOCOLS

In this section, we present our conversion protocols between $\mathbb{Z}_{2^n}$ and $\mathbb{Z}_2^n$. As mentioned in Sec. 3.2, to convert a passively secure protocol to an *actively secure*, the initial protocol needs to be *actively* private. If it is not actively private, then post-verification can make it only *covertly* secure, unless verification is applied immediately *before* the steps that break active privacy.

In Sec. 4.1, we discuss an alternative for the *passively secure* ring conversion protocol itself. A disadvantage of the new protocol is that it is not actively private, so post-verification would give us only covert security. In Sec. 4.2, we see if similar ideas could help us in reducing the *verification cost* of ring conversion. We use an analogous construction to establish a new *verification* procedure, which will not suffer from missing active privacy. The verification procedure is independent, and can be used with both the old and the new version of the passively secure ring conversion protocols.

As described in Sec. 3.2, each conversion from $\mathbb{Z}_2$ to $\mathbb{Z}_{2^n}$ would require a trusted bit $b_i \in \mathbb{Z}_{2^n}$, which is already an n-bit value. To convert all n bits, the cost of generating all required trusted bits would become $O(n^2)$. We see if we can reduce it to $O(n)$.

In this section, we denote a protocol converting $\mathbb{Z}_2^n \to \mathbb{Z}_{2^n}$ by XorToAdd, and a protocol converting $\mathbb{Z}_{2^n} \to \mathbb{Z}_2^n$ by AddToXor. We use $[\![x]\!] = (x^1, x^2, x^3)$ to denote a secret-shared value x, where x^i is the share of party P_i.

4.1 An Alternative Conversion Protocol

The current implementation of XorToAdd protocol of Sharemind is given in Algorithm 1. For each input bit, it uses the subprotocol ShareConv [7] (i.e. *share conversion*), for clarity repeated in Algorithm 2, that converts a bit shared over $\mathbb{Z}_2$ to a bit shared over $\mathbb{Z}_{2^n}$. The linear combination $\sum_{i=0}^{n-1} 2^i b_i$ of bits b_i shared over $\mathbb{Z}_{2^n}$ comprises the final result. In ShareConv, one party sends an n-bit random value to one of the other parties, followed by all three parties exchanging some single bits that do not depend on n. This subprotocol is quite asymmetric, making the whole XorToAdd conversion asymmetric. The random bit b does not depend on the input data, so it can be preshared in the preprocessing phase, and the online phase of the protocol in Algorithm 2 can be actually made single-round. Hence, the protocol in Algorithm 1 is also single-round, but its communication cost is $O(n^2)$ since each bit needs to be converted.

Algorithm 1: Old XorToAdd protocol

Data: $n \in \mathbb{N}$, and shared bits $[\![b_0]\!], \ldots, [\![b_{n-1}]\!]$
Result: $[\![m]\!]$, shared over $\mathbb{Z}_{2^n}$, such that $m = \sum_{i=0}^{n-1} 2^i b_i$

Parties use the protocol ShareConv to convert each $[\![b_i]\!]$ from $\mathbb{Z}_2$ to $\mathbb{Z}_{2^n}$, getting n additively shared values c_i.
Return the additively shared value $[\![m]\!] = \sum_{i=0}^{n-1} 2^i [\![c_i]\!]$.

Algorithm 2: [7] Protocol $[\![v]\!] \leftarrow \mathsf{ShareConv}([\![u]\!])$ for converting a share $[\![u]\!] \in \mathbb{Z}_2$ to $[\![v]\!] \in \mathbb{Z}_{2^n}$

Data: Value $[\![u]\!] = (u^1, u^2, u^3)$ shared over $\mathbb{Z}_2$.
Result: Value $[\![v]\!]$ such that $[\![u]\!] = [\![v]\!]$, shared over $\mathbb{Z}_{2^n}$.

P_1 generates a random bit $b \xleftarrow{\$} \mathbb{Z}_2$ and sets $m = b \oplus u^1$.
P_1 *locally* converts m to $\mathbb{Z}_{2^n}$, generates a random $m_{12} \xleftarrow{\$} \mathbb{Z}_{2^n}$ and computes $m_{13} = m - m_{12}$.
P_1 *locally* generates a random $b_{12} \xleftarrow{\$} \mathbb{Z}_2$ and computes $b_{13} = b - b_{12} = b \oplus b_{12}$.
P_1 sends (b_{12}, m_{12}) to P_2, and (b_{13}, m_{13}) to P_3.
P_2 sets $s_{23} = b_{12} \oplus u^2$ and sends the result to P_3.
P_3 sets $s_{32} = b_{13} \oplus u^3$ and sends the result to P_2.
P_2 and P_3 both set $s = s_{23} \oplus s_{32}$.
if $s = 1$ **then**
  P_2 sets $v^2 = (1 - m_{12})$
  P_3 sets $v^3 = (-m_{13})$
else
  P_2 sets $v^2 = m_{12}$
  P_3 sets $v^3 = m_{13}$
end
Return $[\![v]\!] = (0, v^2, v^3)$.

A new variant of XorToAdd protocol is given in Algorithm 3. The protocol idea was hinted in [32], but dismissed for a protocol similar to Algorithm 1. This protocol is less efficient for small n, including $n = 32$ and $n = 64$, which are currently the main data types of Sharemind system. Nevertheless, this protocol has less communication for larger number of bits since its complexity is $O(n)$, although its number of rounds is $O(\log(n))$. As a subprotocol, it uses secure addition of two XOR-shared numbers, which returns a XOR-shared output [41, Sec. 3.1]. An adaptation of this protocol to *subtraction*, denoted SubXor, is given in Algorithm 4. It consists

Algorithm 3: New XorToAdd protocol

Data: $n \in \mathbb{N}$ and shared bits $[\![b_0]\!], \dots, [\![b_{n-1}]\!]$
Result: $[\![m]\!]$, shared over $\mathbb{Z}_{2^n}$, such that $m = \sum_{i=0}^{n-1} 2^i b_i$

Parties P_1 and P_3 generate n random bits $r_i^1 \overset{\$}{\leftarrow} \mathbb{Z}_2$ and $r_i^3 \overset{\$}{\leftarrow} \mathbb{Z}_2$ respectively for $i \in \{0, \dots, n-1\}$.
Treating $(r_i^1, 0, r_i^3)$ as the sharing of a new shared bit r_i, parties invoke the addition protocol SubXor$([[\![b_0]\!], \dots, [\![b_{n-1}]\!]], [[\![r_0]\!], \dots, [\![r_{n-1}]\!]])$, obtaining $[\![t_0]\!], \dots, [\![t_{n-1}]\!]$.
Parties open $[\![r_0]\!], \dots, [\![r_{n-1}]\!]$ to party P_2.
Parties open $[\![t_0]\!], \dots, [\![t_{n-1}]\!]$ to party P_3.
Return $[\![m]\!] = (0, \sum_{i=0}^{n-1} 2^i r_i, \sum_{i=0}^{n-1} 2^i t_i)$.

Algorithm 4: Secure subtraction of XOR-shared numbers (SubXor), adapted from [41]

Data: $n \in \mathbb{N}$, shared bits $[\![b_0]\!], \dots, [\![b_{n-1}]\!], [\![r_0]\!], \dots, [\![r_{n-1}]\!] \in \mathbb{Z}_2$
Result: $[\![s_0]\!], \dots, [\![s_{n-1}]\!] \in \mathbb{Z}_2$ satisfying $\sum_{i=0}^{n-1} 2^i b_i = (\sum_{i=0}^{n-1} 2^i r_i + \sum_{i=0}^{n-1} 2^i s_i) \bmod 2^n$

```
[[c_0]] = 0                                   /* c_0, ..., c_n ∈ Z_2 */
for k = 0 to n − 1 do
  | [[s_k]] = [[b_k]] ⊕ [[r_k]] ⊕ [[c_k]]
  | [[c_{k+1}]] = (([[c_k]] ⊕ [[r_k]]) ∧ ([[c_k]] ⊕ [[s_k]])) ⊕ [[c_k]]
end
Return [[s_0]], ..., [[s_{n-1}]].
```

of bit additions and n sequential bit multiplications, which can be computed in $\log(n)$ rounds if parallelized. Since the value c_{n-1} is actually never used, the number of multiplications can be reduced to $n-1$.

Proposition 4.1. *The protocol of Algorithm 4 is correct.*

Proof. First, the computation of c_{k+1}, the next carry bit, could be expressed as

$$c_{k+1} = \mathit{if}\ c_k = 0\ \mathit{then}\ r_k \wedge s_k\ \mathit{else}\ r_k \vee s_k\ .$$

Second, at the beginning of the loop, the following invariant is satisfied:

$$\sum_{i=0}^{k-1} 2^i r_i + \sum_{i=0}^{k-1} 2^i s_i = \sum_{i=0}^{k-1} 2^i b_i + 2^k c_k\ .$$

During the loop, we compute s_k and c_{k+1} so, that the invariant stays valid. There are two possibilities:

- $c_k = 0$, meaning that there is no carry from bit $k-1$ to bit k. In this case, we select s_k so, that $r_k \oplus s_k = b_k$. The outgoing carry is $r_k \wedge s_k$.
- $c_k = 1$, i.e. there is incoming carry from bit $k-1$ to bit k. In this case we select s_k so, that $r_k \oplus s_k \oplus 1 = b_k$. The outgoing carry is 1 if at least one of r_k or s_k is 1.

When $k = n$ is reached, the invariant gives us $\sum_{i=0}^{n-1} 2^i b_i = (\sum_{i=0}^{n-1} 2^i r_i + \sum_{i=0}^{n-1} 2^i s_i) \bmod 2^n$ as we wanted. □

We note that Algorithm 4 can easily be parallelized, similarly to parallel adders. Indeed, the only state going from one iteration of the loop to the next is the bit c_k. The parallelized version would work in (parallel) time $O(\log n)$.

Protocols of Sharemind need to be *universally composable* [31], so that they could be combined with each other in arbitrary way in larger applications. However, the output of XorToAdd has been shared only among P_2 and P_3, so there might be a question whether some final reshare step is missing. It has been proven in [4] that final resharing is not necessary if composed protocols are *input-private*. Resharing is needed only when followed by a protocol that is *not* input private, and Sharemind system avoids such protocols. Input privacy is defined as indistinguishability of the party's view and the simulation of this view, whereas the simulation is based only on the input of that party.

Proposition 4.2. *The SubXor protocol of Algorithm 4 is universally composable in presence of a passive adversary corrupting one party.*

Proof. The protocol Algorithm 4 only uses universally composable subprotocols (AND and XOR of bits) as black boxes, so it is itself also universally composable by composition theorem. □

Proposition 4.3. *The XorToAdd protocol of Algorithm 3 is input-private in presence of a passive adversary corrupting one party.*

Proof. We show how to construct the simulator S_j for the view of a party P_j, using only inputs of P_j and without interacting with the real protocol.

The protocol SubXor is universally composable and hence can be substituted by a blackbox ideal functionality that computes XOR-sharing of $t = b - r$ from bits $[[\![b_0]\!], \dots, [\![b_{n-1}]\!]]$ and $[[\![r_0]\!], \dots, [\![r_{n-1}]\!]]$, where $b = \sum_{i=0}^{n-1} 2^i b_i$ and $r = \sum_{i=0}^{n-1} 2^i r_i$. For its simulation, S_j needs the inputs b_i^j and r_i^j for $i \in \{0, \dots, n-1\}$. For all i, S_1 takes $r_i^1 \overset{\$}{\leftarrow} \mathbb{Z}_2$, S_3 takes $r_i^3 \overset{\$}{\leftarrow} \mathbb{Z}_2$, and S_2 takes $r_i^2 = 0$. Each b_i^j is an input of P_j and so is available to S_j.

After executing SubXor, there are more values in the views of parties that need to be simulated. First of all, we note that, if n bits $a_0, \dots, a_{n-1}$ are mutually independent, and each a_i is uniformly distributed in $\mathbb{Z}_2$, then $a = \sum_{i=0}^{n-1} 2^i a_i$ is uniformly distributed in $\mathbb{Z}_{2^n}$, and vice versa.

(1) The party P_1 does not receive any messages from the other parties and always returns 0, so S_1 does not need to simulate anything else.
(2) The party P_2 gets the bits $r_0, \dots, r_n$. More precisely, since there is no additional resharing, for all i, it gets the bit r_i^1 from P_1, and r_i^3 from P_3. Both r_i^1 and r_i^3 are uniformly distributed and have not been used by S_2 so far. Hence, S_2 generates $r_i^1 \overset{\$}{\leftarrow} \mathbb{Z}_2$ and $r_i^3 \overset{\$}{\leftarrow} \mathbb{Z}_2$.
(3) The party P_3 gets the value $t = b - r$, where $b = \sum_{i=0}^{n-1} 2^i b_i$ and $r = \sum_{i=0}^{n-1} 2^i r_i$. Since SubXor is universally composable, the bit shares $t_0^3, \dots, t_{n-1}^3$ that S_3 produced during

simulation of SubXor are random values that are not related to $b - r$ in any way, so there are no constraints on t yet.

Each r_i is computed as $r_i^1 \oplus r_i^3$, where S_3 has already simulated r_i^3, but not r_i^1. Hence, r_i^1 masks r_i, so each r_i is uniformly distributed in $\mathbb{Z}_2$, and $r = \sum_{i=0}^{n-1} 2^i r_i$ uniformly distributed in $\mathbb{Z}_{2^n}$, so r can serve as a mask for b in $t = b - r$. Again, since t is uniformly distributed in $\mathbb{Z}_{2^n}$, each t_i is uniformly distributed in $\mathbb{Z}_2$, so S_3 generates $t_i \xleftarrow{\$} \mathbb{Z}_2$ for all $i \in \{0, \dots, n-1\}$. It now needs to simulate shares t_i^1 and t_i^2 sent to P_3 by P_1 and P_2 respectively. S_3 generates $t_i^1 \xleftarrow{\$} \mathbb{Z}_2$ and takes $t_i^2 = t_i \oplus t_i^1 \oplus t_i^3$. □

So far, we have proven that the protocol is passively secure. However, it is not actively private. If we want to achieve active security, the verification should be applied immediately before the opening of final shares to P_2 and P_3. In the next section, we focus on using the construction of Algorithm 3 to establish a new *verification* procedure, which will not have this problem.

4.2 An Alternative Verification of Conversion

The outline of verification of ring conversion of [30], that we repeated in 3.2, is very similar to the outline of Algorithm 1. For each conversion, n trusted bits $b \in \{0, 1\}$ have been generated and shared over $\mathbb{Z}_{2^n}$. Both additive and XOR-sharings can be reconstructed from such bits $[b_0, \dots, b_{n-1}] \in \mathbb{Z}_{2^n}^n$, which are $[b_0 \bmod 2, \dots, b_{n-1} \bmod 2] \in \mathbb{Z}_2^n$ and $\sum_{i=0}^{n-1} 2^i b_i \in \mathbb{Z}_{2^n}$. To transform random bits b_i to actual bits x_i, it remains for the prover to publish the hint $c_i = b_i \oplus x_i$.

The good property of this method is that the prover needs to publish n bits for an n-bit conversion. On the other hand, each conversion requires n trusted bits b_i, and since each of them should be shared over $\mathbb{Z}_{2^n}$, the complexity of the preprocessing phase becomes $O(n^2)$. As the result, for complex protocols that use a lot of share conversions (such as division or bit shift protocols), the communication complexity of the preprocessing phase can be thousands of times larger than the execution phase [30]. This can be a problem, since the main advantage of the verification mechanism of [30] is that it can turn arbitrary customized passively secure protocols to actively secure ones, and the verification of such complex protocols is actually one of the most interesting things that it is able to do. We would like to get reasonable preprocessing cost for these protocols, making them practical.

The problem of the $O(n^2)$ complexity of trusted bit generation is very similar to the protocol of Algorithm 1. We can improve the complexity of preprocessing by using the structure of Algorithm 3. Instead of computing additively shared value as a linear combination of n bits shared over $\mathbb{Z}_{2^n}$, the verifiers will execute a two-party analogue of Algorithm 3. Let $\langle\langle x \rangle\rangle = (x^1, x^2)$ denote a value x additively shared among the verifiers V_1 and V_2. The verification procedure is described in Figure 1.

In the preprocessing phase, the parties generate $n-1$ trusted multiplication triples in $\mathbb{Z}_2$ for each ring conversion $\mathbb{Z}_{2^n} \leftrightarrow \mathbb{Z}_2^n$ that will be performed by the verifiers. The triples will be needed for the subprotocol SubXor. Triple generation method is the same as in Sec. 3.2.

Verification of the prover's entire computation is verified gate-by-gate, as described in Sec. 3.2. The difference comes at the point when the verifiers need to apply share conversion. Assume that the verifiers V_1 and V_2 hold additive shares x^1 and x^2 respectively, such that $x = x^1 + x^2 \in \mathbb{Z}_{2^n}$. They want to convert $\langle\langle x \rangle\rangle$ to shared bits $[\langle\langle y_0 \rangle\rangle, \dots, \langle\langle y_{n-1} \rangle\rangle]$, each bit shared as $y_i = y_i^1 \oplus y_i^2$ in $\mathbb{Z}_2$.

First, V_1 locally computes bit decomposition $[x_0^1, \dots, x_{n-1}^1] \leftarrow x^1$, and V_2 computes $[x_0^2, \dots, x_{n-1}^2] \leftarrow (-x^2)$. Let $v_i, u_i \in \mathbb{Z}_2$ be shared as $v_i = x_i^1 \oplus 0$ and $u_i = 0 \oplus x_i^2$. This sharing does not require any communication since V_1 already knows x_i^1, and V_2 knows x_i^2. The verifiers now hold shared bit vectors $[\langle\langle v_0 \rangle\rangle, \dots, \langle\langle v_{n-1} \rangle\rangle]$ and $[\langle\langle u_0 \rangle\rangle, \dots, \langle\langle u_{n-1} \rangle\rangle]$.

The verifiers execute two-party instance of SubXor (Algorithm 4, using two-party sharing $\langle\langle \cdot \rangle\rangle$ instead of three-party sharing $[\![\cdot]\!]$), applying it to the bit vectors $[\langle\langle v_0 \rangle\rangle, \dots, \langle\langle v_{n-1} \rangle\rangle]$ and $[\langle\langle u_0 \rangle\rangle, \dots, \langle\langle u_{n-1} \rangle\rangle]$. The XOR operations of SubXor are computed locally on shares. For the $(n-1)$ AND operations, the verifiers use the $(n-1)$ precomputed multiplication triples over $\mathbb{Z}_2$. The shared bits $[\langle\langle y_0 \rangle\rangle, \dots, \langle\langle y_{n-1} \rangle\rangle]$ resulting from SubXor protocol are the bits of $x = x^1 - (-x^2) = x^1 + x^2$. Instead of applying SubXor to compute $x^1 - (-x^2) = x^1 + x^2$, the verifiers may directly use addition of XOR-shared numbers from [41, Sec. 3.1], which has essentially the same complexity.

Let us now consider the other conversion direction. The verifiers hold shared bits $[\langle\langle x_0 \rangle\rangle, \dots, \langle\langle x_{n-1} \rangle\rangle]$ and want to convert them to additive shares $\langle\langle y \rangle\rangle$. In that case, they need one more hint from the prover. The prover will commit y itself as a hint, sharing it among the verifiers. The verifiers proceed in their verification using provided $\langle\langle y \rangle\rangle$. They also use the procedure $\mathbb{Z}_{2^n} \rightarrow \mathbb{Z}_2^n$ (described above) to decompose $\langle\langle y \rangle\rangle$ to bits $[\langle\langle y_0 \rangle\rangle, \dots, \langle\langle y_{n-1} \rangle\rangle]$. They will then check that $x_i \oplus y_i = 0$ for all $i \in \{0, \dots, n-1\}$. The final check does not contribute to computation complexity, since it becomes a part of the succinct zero check described in Sec. 3.2.

Compared to the old verification method, using multiplication triples requires prover to open $a_i \oplus s_i$ and $a_i \oplus t_i$ for all values s_i and t_i that participate in the multiplications of SubXor protocol, so now there are $2(n-1)$ opened bits instead of n bits of the old verification method. Moreover, if the transition goes from $\mathbb{Z}_2^n$ to $\mathbb{Z}_{2^n}$, then the verifiers additionally need an n-bit hint to be committed to. The communication complexity of the verification phase may thus increase up to three times.

On the other hand, instead of generating n trusted bits in $O(n^2)$ communication, the parties now generate n AND triples in $O(n)$ communication, while the number of rounds stays the same. Hence, the complexity of the preprocessing phase decreases n times. We believe this to be a reasonable trade-off, since the preprocessing phase is much more expensive and is a bottleneck for the covert/active security in general. This aspect gives practical advantage to our protocols.

Preprocessing.

For each share conversion, generate $(n-1)$ trusted multiplication triples $(a_i, b_i, c_i) \in \mathbb{Z}_2$ (i.e. $c_i = a_i \wedge b_i$).

Verification.

- $\mathbb{Z}_{2^n} \to \mathbb{Z}_2^n$: Let the verifiers V_1 and V_2 hold additive shares x^1 and x^2 respectively, such that $x = x^1 + x^2 \in \mathbb{Z}_{2^n}$. They want to locally convert $\langle\langle x \rangle\rangle$ to shared bits $[\langle\langle y_0 \rangle\rangle, \dots, \langle\langle y_{n-1} \rangle\rangle]$.
  (1) V_1 locally computes bit decomposition $[x_0^1, \dots, x_{n-1}^1] \leftarrow x^1$, and V_2 computes $[x_0^2, \dots, x_{n-1}^2] \leftarrow (-x^2)$.
  (2) Let $v_i, u_i \in \mathbb{Z}_2$ be shared as $v_i = x_i^1 \oplus 0$ and $u_i = 0 \oplus x_i^2$.
  (3) The verifiers execute SubXor (Algorithm 4), applying it to the bit vectors $[\langle\langle v_0 \rangle\rangle, \dots, \langle\langle v_{n-1} \rangle\rangle]$ and $[\langle\langle u_0 \rangle\rangle, \dots, \langle\langle u_{n-1} \rangle\rangle]$. The XOR operations of SubXor are computed locally. For the $(n-1)$ AND operations, the verifiers use the $(n-1)$ precomputed multiplication triples over $\mathbb{Z}_2$.
  (4) The verifiers proceed with the shares $[\langle\langle y_0 \rangle\rangle, \dots, \langle\langle y_{n-1} \rangle\rangle]$ resulting from SubXor protocol.
- $\mathbb{Z}_2^n \to \mathbb{Z}_{2^n}$: Let the verifiers hold shared bits $[\langle\langle x_0 \rangle\rangle, \dots, \langle\langle x_{n-1} \rangle\rangle]$ that they want to convert to additive shares of $\langle\langle y \rangle\rangle$. In this case, the prover has to additionally commit y itself as a hint, sharing it among the verifiers. The verifiers proceed in their verification using provided $\langle\langle y \rangle\rangle$. They also use the procedure described above to decompose y to bits $[y_0, \dots, y_{n-1}]$. They will then check that $x_i \oplus y_i = 0$ for all $i \in \{0, \dots, n-1\}$.

Figure 1: New verification of ring conversions $\mathbb{Z}_{2^n} \leftrightarrow \mathbb{Z}_2^n$

4.3 Generalization to n parties

We note that the verification mechanism of [30] is not constrained to 3-party protocols (although 3-party case is the most efficient), but can be generalized to any number n of parties, as long as there is an honest majority (the number of corrupted parties is $t < n/2$). Addition and subtraction of two xor-shared numbers (protocol SubXor) can be generalized directly, taking n-party protocol versions of underlying basic operations. In XorToAdd protocol, we would need to generate r_t shared random numbers, and execute t instances of n-party XorToAdd to compute $s = b - r_1 - \cdots - r_t$. The final result would be shared among some $t+1$ parties as $r_1, \dots, r_t, s$, so that no coalition of t parties would learn the secret. In the verification procedure, each of $t+1$ verifiers would convert its additive share locally to bits, and t instances of bitwise addition protocol would need to be executed. We leave detailed security and performance analysis of n-party generalization out of this paper.

5 EVALUATION

We benchmarked our implementation on three 2× Intel Xeon E5-2640 v3 2.6 GHz/8GT/20M servers, with 125GB RAM running on a 1Gbps LAN, similarly to the benchmarks reported in [30]. We run a large number of instances of protocol verification in parallel, and report the amortized time for a single protocol.

Table 1: Running times of the verification phase for 32-bit AddToXor protocol

# runs	time (s)					
	Old verification [30]			New verification (Fig. 1)		
	P_1	P_2	P_3	P_1	P_2	P_3
10^3	0.481	0.491	0.484	0.478	0.489	0.492
10^4	0.886	0.896	0.920	0.865	0.916	0.912
10^5	2.35	2.73	2.77	2.32	2.84	2.90
10^6	14.8	18.1	18.7	14.6	19.0	19.5

Differently from [30], we have chosen to not measure the running time of the execution phase of new protocols. The performance profile of the new XorToAdd protocol is very different from the old one, with larger number of rounds but smaller communication. The actual execution-phase benefits of the new protocol depend a lot on the other operations that the privacy-preserving application using this protocol performs, as well as on the amount of data it processes. The increase of the number of rounds may have a noticeable effect on the execution-time performance, if everything else in the application has very small round complexity and data sizes are small. But if some other protocol requiring a significant number of rounds is run in parallel with XorToAdd, or if the amount of processed data is large, then its round complexity does not matter and the reduced bandwidth brings performance benefits. We thus evaluate the number of rounds and communicated bits of the execution phase of our protocols, as these are more robust over different applications. Additionally, the execution phase has the smallest run-time of the three phases, and has the least effect on the total running time. Also note that the execution times of preprocessing and verification phases depend much less on context.

Laud et al. [30] have evaluated the older versions of the AddToXor and XorToAdd protocols that have been available in Sharemind implementation at that time. Those protocols had not been optimized, although more efficient versions have already been developed on theory level. The domain-specific language [32], enables to re-implement the same AddToXor and XorToAdd protocols, and apply some automatic circuit optimizations to them. It reduces the total communication time of the execution phase, as well as of pre- and postprocessing phases. In this work, we report the times of these optimized protocols, and show that our new verification improves them even more.

For the AddToXor protocol, the comparison of the old and new verification methods is given in Table 1. For the XorToAdd protocol, the comparison is given in Table 2. Since the protocols are asymmetric, we evaluate time needed to verify the parties P_1, P_2, P_3 separately. The three proofs have been run in parallel, so the total running time is the maximum of these. The verification time for the new implementation slightly increases, as we expected. However, the difference is not significant. The verification times of the new XorToAdd protocol are more evenly distributed between the provers, since the protocol itself is more symmetric.

Table 2: Running times of the verification phase for 32-bit XorToAdd protocol

# runs	time (s)								
	old prot. (Alg. 1), old verification [30]			new prot. (Alg. 3), old verification			new prot. (Alg. 3) new verification (Fig. 1)		
	P_1	P_2	P_3	P_1	P_2	P_3	P_1	P_2	P_3
10^3	0.450	0.516	0.503	0.510	0.515	0.504	0.490	0.506	0.503
10^4	0.760	1.06	1.07	0.956	0.961	0.967	0.964	0.962	0.975
10^5	2.31	4.39	4.34	3.06	3.52	3.51	2.97	3.32	3.38
10^6	14.1	27.4	27.9	21.0	25.6	25.7	21.0	23.9	23.8

Table 3: Time to generate $u = 10^8$ verified 32-bit tuples [30]

tuple	mult. triples	trusted bits	bitwise AND triples
time	236 s	72 s	236 s

The total cost of actively secure bit conversion protocols is given in Table 4. To estimate the preprocessing time, we counted the total number of precomputed tuples required for verification, doing it separately for the old and the new verification. We then estimated the preprocessing time using the timings in Table 3, where one XOR-shared AND triple is equivalent to 32 multiplication triples over $\mathbb{Z}_2$. The execution phase has been actually measured only for old XorToAdd protocol, since it has not been improved compared to the current version of this protocol inside Sharemind. The protocol AddToXor has been optimized and can be only faster than its older version inside Sharemind, so we may take the running time of that older protocol as an upper bound. The situation is more complicated for the new version of XorToAdd protocol. The total communication, as well as the number of rounds, have been increased. We see that these numbers are not very different from AddToXor protocol, which itself has a similar structure, and uses SubXor as a subprotocol. We may assume that the protocol will not be much slower, and take "around 2 times slower than the old XorToAdd protocol" as a conservative upper bound.

From the results of Table 4, we see that if we apply the new verification method, the total time for the 32-bit AddToXor protocol reduces from 206 µs to 164 µs. For the 32-bit XorToAdd protocol, we see no improvement.

To see how well our new methods would work in general, we have counted the bit communication of all three phases for larger Sharemind protocols. These numbers are given in Table 5 and Table 6 for integer protocols, and in Table 7 for floating point protocols. For each protocol, the results are presented in the form $\frac{x:y:z}{1:a:\boldsymbol{b}}$. The upper line lists the total communication cost (in bits): x for the execution of the protocol, y for its verification in the post-execution phase, and z for the generation of precomputed tuples in the preprocessing phase. The suffixes k and M denote the multipliers 10^3 and 10^6, respectively. The lower line is computed directly from the upper line, and it shows how many times more expensive each phase is, compared to the execution phase, i.e. $a = y/x$, $\boldsymbol{b} = z/x$. The only difference is the *new* XorToAdd protocol since we compare it with its *old* version, and also the private shift right protocol ($[\![a]\!] \gg [\![b]\!]$), for which we also get a new version since it uses XorToAdd as subroutine. The lower line of these two protocols is of the form $c{:}a{:}\boldsymbol{b}$, where $c = x/x'$, $a = y/x'$, $\boldsymbol{b} = z/x'$, and x' is the cost of the execution phase of the *old version*. We see that, while the new XorToAdd protocol is indeed worse for 8, 16, 32 bits, it actually becomes better starting from 64 bits, and $[\![a]\!] \gg [\![b]\!]$ also does benefit from it.

Although some floating point protocols also use XorToAdd as subroutine, replacing it with a new one does not give much improvement, so we have not included the new versions of these protocols into Table 7.

For all protocols, we compare the cost using old [30] and new (Fig. 1) verification methods. This choice does not affect the execution phase at all, but the costs of pre- and postprocessing are different. As we expected, for the new verification method, the cost of the verification phase itself has increased, since now each bit that undergoes conversion requires two or three bits of hint from the prover. However, the increase is not too large compared to the decrease in the overhead of the preprocessing phase, which has been reduced by an order of magnitude for division and left shift protocols. Hence, the total cost of active and covert security for these protocols becomes much smaller.

The private shift right protocol ($[\![a]\!] \gg [\![b]\!]$) still has a relatively high preprocessing phase. The reason for this is its *overly efficient* execution phase, using tricks similar to [28, Alg. 4], causing the amount of communication (the cost of execution phase) to be asymptotically smaller than the amount of local computations (which have to be verified). We believe that a special form of precomputed triples could be used to verify these local computations. We do not explore this direction in this paper.

6 CONCLUSION

In this work, we have improved the performance of active and covert security of Sharemind protocols. In particular, we have proposed a new method for verification of computation of conversion beween rings $\mathbb{Z}_{2^n}$ and $\mathbb{Z}_2^n$, which is in turn sufficient to compute conversion between any two rings $\mathbb{Z}_{2^n}$ and $\mathbb{Z}_{2^m}$. We have evaluated both methods and compared them. Since Sharemind protocols contain many such conversions, all of which need to be verified to achieve active or covert security,

Table 4: Total cost of actively secure bit conversion protocols

	32-bit AddToXor		32-bit XorToAdd		
protocol (old – Alg. 1, new – Alg. 3)	old	old	old	new	new
verification (old – [30], new – Fig. 1)	old	new	old	old	new
# AND triples	58	60	0	90	92
# MULT triples	0	0	64	0	0
# trusted bits	64	0	96	64	0
running time of preprocessing phase (µs)	183	142	220	258	217
# rounds of exec.phase	7	7	1	7	7
# comm.bits of exec.phase	1024	1024	1088	1216	1216
running time of execution phase (µs)	< 2.3	< 2.3	5.1	< 10	< 10
max. verification time (µs)	18.6	19.5	27.8	25.8	23.9
total time (µs) ≈	206	164	252	295	250

Table 5: Total bit communication of different phases for 8 and 16-bit integer protocols

Integer operation	8 bits		16 bits	
	Old verif. [30]	New verif. (Fig. 1)	Old verif. [30]	New verif. (Fig. 1)
$[\![x]\!] \cdot [\![y]\!]$	48 : 192 : 1008 *1 : 4 :* ***21***	48 : 192 : 1008 *1 : 4 :* ***21***	96 : 384 : 2017 *1 : 4 :* ***21***	96 : 384 : 2017 *1 : 4 :* ***21***
$[\![x]\!] \;/\; [\![y]\!]$	4306 : 36k : 1.1M *1 : 8 :* ***249***	4306 : 46.5k : 225k *1 : 10 :* ***52***	10.0k : 85.7k : 4.9M *1 : 8 :* ***486***	10k : 113k : 542.2k *1 : 11 :* ***54***
$[\![x]\!] \;/\; y$	404 : 5172 : 92.3k *1 : 12 :* ***228***	404 : 7120 : 34.3k *1 : 17 :* ***85***	948 : 12.1k : 330.6k *1 : 12 :* ***349***	948 : 16.0k : 77.6k *1 : 16 :* ***82***
$[\![x]\!] \ll [\![y]\!]$	144 : 818 : 5234 *1 : 5 :* ***36***	144 : 914 : 4706 *1 : 6 :* ***33***	400 : 2648 : 17.7k *1 : 6 :* ***44***	400 : 2840 : 14.7k *1 : 7 :* ***37***
$[\![x]\!] \overset{old}{\gg} [\![y]\!]$	312 : 4050 : 22.7k *1 : 12 :* ***73***	312 : 4082 : 22.4k *1 : 13 :* ***72***	848 : 14.5k : 82.2k *1 : 17 :* ***97***	848 : 14.6k : 80.2k *1 : 17 :* ***95***
$[\![x]\!] \overset{new}{\gg} [\![y]\!]$	472 : 4738 : 26.1k *1.5 : 15.2 :* ***84***	472 : 4818 : 25.4k *1.5 : 15.4 :* ***81***	1120 : 15.6k : 87k *1.3 : 18.4 :* ***103***	1120 : 15.8k : 83.0k *1.3 : 18.6 :* ***98***
$[\![x]\!] \gg y$	180 : 1690 : 15.2k *1 : 9 :* ***84***	180 : 2224 : 10.8k *1 : 12 :* ***60***	468 : 4218 : 53.5k *1 : 9 :* ***114***	468 : 5504 : 26.6k *1 : 11 :* ***57***
$[\![x]\!] = [\![y]\!]$	50 : 200 : 1571 *1 : 4 :* ***31***	50 : 232 : 1219 *1 : 4 :* ***24***	106 : 424 : 4549 *1 : 4 :* ***43***	106 : 488 : 2563 *1 : 4 :* ***24***
$[\![x]\!] < [\![y]\!]$	280 : 2748 : 16.0k *1 : 9 :* ***57***	280 : 2844 : 14.9k *1 : 10 :* ***53***	719 : 7440 : 46.0k *1 : 10 :* ***64***	719 : 7632 : 40.1k *1 : 10 :* ***56***
AddToXor	160 : 1120 : 6403 *1 : 7 :* ***40***	160 : 1152 : 6050 *1 : 7 :* ***38***	416 : 3008 : 18.1k *1 : 7 :* ***44***	416 : 3072 : 16.1k *1 : 7 :* ***39***
$\overset{old}{\text{XorToAdd}}$	80 : 560 : 3722 *1 : 7 :* ***47***	80 : 560 : 3722 *1 : 7 :* ***47***	288 : 2144 : 14.7k *1 : 7 :* ***51***	288 : 2144 : 14.7k *1 : 7 :* ***51***
$\overset{new}{\text{XorToAdd}}$	208 : 1184 : 6739 *2.6 : 15 :* ***84***	208 : 1232 : 6387 *2.6 : 15 :* ***80***	512 : 3136 : 18.8k *1.8 : 11 :* ***65***	512 : 3232 : 16.8k *1.8 : 11 :* ***58***

the performance overheads of some larger protocols has been reduced by an order of magnitude.

While we have managed to significantly reduce the preprocessing phase, we have slightly increased the verification phase. We conclude that the old method can be still prefered in the cases where the preprocessing time is of less importance. Nevertheless, in this paper we have achieved better overall performance.

With the protocols proposed in this paper, we see a general disappearance of huge verification overheads from actively secure Sharemind protocols, compared to passively secure ones. While other actively secure SMC protocol sets may give a better performance on multiplications, real privacy-preserving applications contain a mix of operations with private values and we believe the Sharemind protocol set to be the most suitable choice for them.

ACKNOWLEDGMENTS

The authors would like to thank anonymous reviewers for helpful suggestions. This work is supported by Estonian Research Council, grant no. IUT27-1, and Excellence in IT in Estonia (EXCITE).

Table 6: Total bit communication of different phases for 32 and 64-bit integer protocols

Integer operation	32 bits		64 bits	
	Old verif. [30]	New verif. (Fig. 1)	Old verif. [30]	New verif. (Fig. 1)
$[\![x]\!] \cdot [\![y]\!]$	192 : 768 : 4034 *1 : 4 :* ***21***	192 : 768 : 4034 *1 : 4 :* ***21***	384 : 1536 : 8067 *1 : 4 :* ***21***	384 : 1536 : 8067 *1 : 4 :* ***21***
$[\![x]\!] \, / \, [\![y]\!]$	31.7k : 275k : 28M *1 : 8 :* ***884***	31.7k : 358k : 1.7M *1 : 11 :* ***54***	88.6k : 793k : 180M *1 : 8 :* ***2029***	88.6k : 1.1M : 5.0M *1 : 11 :* ***56***
$[\![x]\!] \, / \, y$	2180 : 27.5k : 1.2M *1 : 12 :* ***563***	2180 : 35.4k : 173.4k *1 : 16 :* ***80***	4932 : 62.0k : 4.7M *1 : 12 :* ***950***	4932 : 77.7k : 383.2k *1 : 15 :* ***78***
$[\![x]\!] \ll [\![y]\!]$	1296 : 9374 : 64.6k *1 : 7 :* ***50***	1296 : 9758 : 50.9k *1 : 7 :* ***39***	4624 : 35.1k : 245.8k *1 : 7 :* ***53***	4624 : 35.9k : 187.8k *1 : 7 :* ***41***
$[\![x]\!] \overset{old}{\gg} [\![y]\!]$	2384 : 53k : 303.6k *1 : 22 :* ***127***	2384 : 53.1k : 294.5k *1 : 22 :* ***124***	7120 : 199.2k : 1.1M *1 : 27 :* ***161***	7120 : 199.5k : 1.1M *1 : 28 :* ***156***
$[\![x]\!] \overset{new}{\gg} [\![y]\!]$	2608 : 52.7k : 297k *1.1 : 22.1 :* ***125***	2608 : 53.0k : 278.8k *1.1 : 22.2 :* ***117***	5968 : 185.3k : 1.1M *0.8 : 26 :* ***154***	5968 : 185.9k : 977k *0.8 : 26.1 :* ***137***
$[\![x]\!] \gg y$	1092 : 9946 : 184.1k *1 : 9 :* ***169***	1092 : 12.5k : 61.2k *1 : 11 :* ***56***	2564 : 22.9k : 661k *1 : 8 :* ***258***	2564 : 28.2k : 138.5k *1 : 10 :* ***54***
$[\![x]\!] = [\![y]\!]$	218 : 872 : 14.3k *1 : 4 :* ***66***	218 : 1000 : 5252 *1 : 4 :* ***24***	442 : 1768 : 49.3k *1 : 4 :* ***112***	442 : 2024 : 10.6k *1 : 4 :* ***24***
$[\![x]\!] < [\![y]\!]$	1750 : 18.7k : 127k *1 : 10 :* ***73***	1750 : 19.0k : 100k *1 : 10 :* ***57***	4109 : 44.7k : 354.7k *1 : 10 :* ***86***	4109 : 45.4k : 238.6k *1 : 11 :* ***58***
AddToXor	1024 : 7552 : 49.4k *1 : 7 :* ***48***	1024 : 7680 : 40.3k *1 : 7 :* ***39***	2432 : 18.2k : 135.5k *1 : 7 :* ***56***	2432 : 18.4k : 96.8k *1 : 7 :* ***40***
$\overset{old}{\text{XorToAdd}}$	1088 : 8384 : 58.7k *1 : 7 :* ***54***	1088 : 8384 : 58.7k *1 : 7 :* ***54***	4224 : 33.2k : 234.2k *1 : 7 :* ***55***	4224 : 33.2k : 234.2k *1 : 7 :* ***55***
$\overset{new}{\text{XorToAdd}}$	1216 : 7808 : 50.8k *1.1 : 7 :* ***47***	1216 : 8000 : 41.7k *1.1 : 7 :* ***38***	2816 : 18.7k : 138.2k *0.7 : 4 :* ***33***	2816 : 19.1k : 99.5k *0.7 : 4 :* ***24***

Table 7: Total bit communication of different phases for floating point protocols

Float operation	32-bit mantissa, 16-bit exponent		64-bit mantissa, 16-bit exponent	
	Old verif. [30]	New verif.	Old verif. [30]	New verif.
$[\![x]\!] + [\![y]\!]$	26.4k : 374.8k : 3.4M *1 : 14 :* ***128***	26.4k : 391.2k : 2.1M *1 : 14 :* ***81***	72.3k : 1.2M : 11.8M *1 : 16 :* ***163***	72.3k : 1.2M : 7.0M *1 : 17 :* ***96***
$[\![x]\!] \cdot [\![y]\!]$	4857 : 44.7k : 945.5k *1 : 9 :* ***195***	4857 : 54.2k : 267.8k *1 : 11 :* ***55***	10.7k : 96.5k : 3.1M *1 : 9 :* ***294***	10.7k : 114.4k : 569k *1 : 10 :* ***53***
$[\![x]\!] = [\![y]\!]$	560 : 3488 : 168.1k *1 : 6 :* ***300***	560 : 5632 : 26.6k *1 : 10 :* ***47***	1008 : 6048 : 469.7k *1 : 6 :* ***466***	1008 : 9600 : 45.4k *1 : 9 :* ***45***
$[\![x]\!] < [\![y]\!]$	4337 : 43.8k : 440.6k *1 : 10 :* ***102***	4337 : 46.8k : 242.5k *1 : 10 :* ***56***	9503 : 98.3k : 1.2M *1 : 10 :* ***126***	9503 : 103.5k : 539k *1 : 10 :* ***57***
$[\![x]\!]^{-1}$	11.3k : 109k : 7.8M *1 : 9 :* ***687***	11.3k : 144k : 695k *1 : 12 :* ***61***	31.3k : 305k : 47.8M *1 : 9 :* ***1528***	31.3k : 398k : 1.9M *1 : 12 :* ***62***
$\sqrt{[\![x]\!]}$	12.2k : 122k : 5.2M *1 : 9 :* ***421***	12.2k : 152k : 746.4k *1 : 12 :* ***61***	46.5k : 476k : 39.8M *1 : 10 :* ***856***	46.5k : 583k : 2.9M *1 : 12 :* ***62***
$\exp([\![x]\!])$	17.6k : 263k : 5.4M *1 : 14 :* ***308***	17.6k : 291k : 1.5M *1 : 16 :* ***86***	55.3k : 941k : 37.3M *1 : 17 :* ***674***	55.3k : 1.0M : 5.4M *1 : 18 :* ***97***
$\ln([\![x]\!])$	96.6k : 1.3M : 15.4M *1 : 13 :* ***159***	96.6k : 1.4M : 7.4M *1 : 14 :* ***76***	276k : 4.2M : 76.6M *1 : 15 :* ***277***	276k : 4.4M : 24.2M *1 : 15 :* ***88***
$\sin([\![x]\!])$	76.0k : 784k : 11.6M *1 : 10 :* ***152***	76.0k : 866k : 4.6M *1 : 11 :* ***60***	244k : 2.7M : 68.3M *1 : 10 :* ***280***	244k : 2.9M : 15.7M *1 : 12 :* ***64***
$\text{erf}([\![x]\!])$	25.7k : 207k : 7.5M *1 : 8 :* ***293***	25.7k : 251k : 1.2M *1 : 9 :* ***48***	89.3k : 730k : 49.8M *1 : 8 :* ***558***	89.3k : 870k : 4.3M *1 : 9 :* ***48***

REFERENCES

[1] Joy Algesheimer, Jan Camenisch, and Victor Shoup. 2002. Efficient Computation Modulo a Shared Secret with Application to the Generation of Shared Safe-Prime Products. In *Advances in Cryptology - CRYPTO 2002, 22nd Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 2002, Proceedings (Lecture Notes in Computer Science)*, Moti Yung (Ed.), Vol. 2442. Springer, 417–432. https://doi.org/10.1007/3-540-45708-9_27

[2] Yonatan Aumann and Yehuda Lindell. 2010. Security Against Covert Adversaries: Efficient Protocols for Realistic Adversaries. *J. Cryptology* 23, 2 (2010), 281–343. https://doi.org/10.1007/s00145-009-9040-7

[3] Carsten Baum, Ivan Damgård, and Claudio Orlandi. 2014. Publicly Auditable Secure Multi-Party Computation. In *Security and Cryptography for Networks - 9th International Conference, SCN 2014. Proceedings (LNCS)*, Michel Abdalla and Roberto De Prisco (Eds.), Vol. 8642. Springer, 175–196. https://doi.org/10.1007/978-3-319-10879-7_11

[4] Dan Bogdanov, Peeter Laud, Sven Laur, and Pille Pullonen. 2014. From Input Private to Universally Composable Secure Multiparty Computation Primitives. In *IEEE 27th Computer Security Foundations Symposium, CSF 2014*. IEEE, 184–198.

[5] Dan Bogdanov, Peeter Laud, and Jaak Randmets. 2014. Domain-Polymorphic Programming of Privacy-Preserving Applications. In *Proceedings of the Ninth Workshop on Programming Languages and Analysis for Security, PLAS@ECOOP 2014, Uppsala, Sweden, July 29, 2014*, Alejandro Russo and Omer Tripp (Eds.). ACM, 53. https://doi.org/10.1145/2637113.2637119

[6] Dan Bogdanov, Sven Laur, and Jan Willemson. 2008. Sharemind: A Framework for Fast Privacy-Preserving Computations. In *ESORICS (LNCS)*, Sushil Jajodia and Javier López (Eds.), Vol. 5283. Springer, 192–206.

[7] Dan Bogdanov, Margus Niitsoo, Tomas Toft, and Jan Willemson. 2012. High-performance secure multi-party computation for data mining applications. *Int. J. Inf. Sec.* 11, 6 (2012), 403–418.

[8] Justin Brickell, Donald E. Porter, Vitaly Shmatikov, and Emmett Witchel. 2007. Privacy-preserving remote diagnostics. In *Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007, Alexandria, Virginia, USA, October 28-31, 2007*, Peng Ning, Sabrina De Capitani di Vimercati, and Paul F. Syverson (Eds.). ACM, 498–507. https://doi.org/10.1145/1315245.1315307

[9] Martin Burkhart, Mario Strasser, Dilip Many, and Xenofontas A. Dimitropoulos. 2010. SEPIA: Privacy-Preserving Aggregation of Multi-Domain Network Events and Statistics. In *19th USENIX Security Symposium, Washington, DC, USA, August 11-13, 2010, Proceedings*. USENIX Association, 223–240. http://www.usenix.org/events/sec10/tech/full_papers/Burkhart.pdf

[10] Octavian Catrina and Amitabh Saxena. 2010. Secure Computation with Fixed-Point Numbers. In *Financial Cryptography and Data Security (LNCS)*, Radu Sion (Ed.), Vol. 6052. Springer, 35–50.

[11] David Chaum, Claude Crépeau, and Ivan Damgård. 1988. Multiparty Unconditionally Secure Protocols (Extended Abstract). In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA*, Janos Simon (Ed.). ACM, 11–19. https://doi.org/10.1145/62212.62214

[12] Ivan Damgård, Matthias Fitzi, Eike Kiltz, Jesper Nielsen, and Tomas Toft. 2006. Unconditionally secure constant-rounds multiparty computation for equality, comparison, bits and exponentiation. In *Proceedings of The Third Theory of Cryptography Conference, TCC 2006 (LNCS)*, Vol. 3876. Springer.

[13] Ivan Damgård, Martin Geisler, Mikkel Krøigaard, and Jesper Buus Nielsen. 2009. Asynchronous Multiparty Computation: Theory and Implementation. In *Public Key Cryptography - PKC 2009, 12th International Conference on Practice and Theory in Public Key Cryptography, Irvine, CA, USA, March 18-20, 2009. Proceedings (Lecture Notes in Computer Science)*, Stanislaw Jarecki and Gene Tsudik (Eds.), Vol. 5443. Springer, 160–179. https://doi.org/10.1007/978-3-642-00468-1_10

[14] Ivan Damgård, Martin Geisler, and Jesper Buus Nielsen. 2010. From Passive to Covert Security at Low Cost. In *Theory of Cryptography, 7th Theory of Cryptography Conference, TCC 2010, Zurich, Switzerland, February 9-11, 2010. Proceedings (LNCS)*, Daniele Micciancio (Ed.), Vol. 5978. Springer, 128–145.

[15] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. 2013. Practical Covertly Secure MPC for Dishonest Majority - Or: Breaking the SPDZ Limits. In *Computer Security - ESORICS 2013 - 18th European Symposium on Research in Computer Security, Egham, UK, September 9-13, 2013. Proceedings (LNCS)*, Jason Crampton, Sushil Jajodia, and Keith Mayes (Eds.), Vol. 8134. Springer, 1–18.

[16] Ivan Damgård, Claudio Orlandi, and Mark Simkin. 2017. Yet Another Compiler for Active Security or: Efficient MPC Over Arbitrary Rings. Cryptology ePrint Archive, Report 2017/908. https://eprint.iacr.org/2017/908.

[17] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. 2012. Multiparty Computation from Somewhat Homomorphic Encryption, See [39], 643–662.

[18] Ivan Damgård and Sarah Zakarias. 2013. Constant-Overhead Secure Computation of Boolean Circuits using Preprocessing. In *TCC*. 621–641. https://doi.org/10.1007/978-3-642-36594-2_35

[19] Daniel Demmler, Thomas Schneider, and Michael Zohner. 2015. ABY - A Framework for Efficient Mixed-Protocol Secure Two-Party Computation. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*. The Internet Society. https://www.ndss-symposium.org/ndss2015/aby---framework-efficient-mixed-protocol-secure-two-party-computation

[20] Vassil Dimitrov, Liisi Kerik, Toomas Krips, Jaak Randmets, and Jan Willemson. 2016. Alternative Implementations of Secure Real Numbers, See [45], 553–564. https://doi.org/10.1145/2976749.2978348

[21] Jun Furukawa, Yehuda Lindell, Ariel Nof, and Or Weinstein. 2017. High-Throughput Secure Three-Party Computation for Malicious Adversaries and an Honest Majority. In *Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part II (Lecture Notes in Computer Science)*, Jean-Sébastien Coron and Jesper Buus Nielsen (Eds.), Vol. 10211. 225–255. https://doi.org/10.1007/978-3-319-56614-6_8

[22] Oded Goldreich, Silvio Micali, and Avi Wigderson. 1987. How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA*, Alfred V. Aho (Ed.). ACM, 218–229. https://doi.org/10.1145/28395.28420

[23] Wilko Henecka, Stefan Kögl, Ahmad-Reza Sadeghi, Thomas Schneider, and Immo Wehrenberg. 2010. TASTY: tool for automating secure two-party computations. In *CCS '10: Proceedings of the 17th ACM conference on Computer and Communications Security*. ACM, 451–462.

[24] Marcel Keller, Emmanuela Orsini, and Peter Scholl. 2016. MASCOT: Faster Malicious Arithmetic Secure Computation with Oblivious Transfer, See [45], 830–842. https://doi.org/10.1145/2976749.2978357

[25] Liisi Kerik, Peeter Laud, and Jaak Randmets. 2016. Optimizing MPC for robust and scalable integer and floating-point arithmetic. In *Proceedings of WAHC'16 - 4th Workshop on Encrypted Computing and Applied Homomorphic Cryptography*.

[26] Eizen Kimura, Koki Hamada, Ryo Kikuchi, Koji Chida, Kazuya Okamoto, Shirou Manabe, Tomohiro Kuroda, Yasushi Matsumura, Toshihiro Takeda, and Naoki Mihara. 2016. Evaluation of Secure Computation in a Distributed Healthcare Setting. In *Exploring Complexity in Health: An Interdisciplinary Systems Approach - Proceedings of MIE2016 at HEC2016, Munich, Germany, 28 August - 2 September 2016. (Studies in Health Technology and Informatics)*, Alexander Hörbst, Werner O. Hackl, Nicolette de Keizer, Hans-Ulrich Prokosch, Mira Hercigonja-Szekeres, and Simon de Lusignan (Eds.), Vol. 228. IOS Press, 152–156. https://doi.org/10.3233/978-1-61499-678-1-152

[27] Toomas Krips and Jan Willemson. 2014. Hybrid Model of Fixed and Floating Point Numbers in Secure Multiparty Computations. In *Information Security - 17th International Conference, ISC 2014, Hong Kong, China, October 12-14, 2014. Proceedings (Lecture Notes in Computer Science)*, Sherman S. M. Chow, Jan Camenisch, Lucas Chi Kwong Hui, and Siu-Ming Yiu (Eds.), Vol. 8783. Springer, 179–197. https://doi.org/10.1007/978-3-319-13257-0_11

[28] Peeter Laud. 2014. A Private Lookup Protocol with Low Online Complexity for Secure Multiparty Computation. In *Information and Communications Security - 16th International Conference, ICICS 2014, Hong Kong, China, December 16-17, 2014, Revised Selected Papers (Lecture Notes in Computer Science)*, Lucas Chi Kwong Hui, S. H. Qing, Elaine

Shi, and Siu-Ming Yiu (Eds.), Vol. 8958. Springer, 143–157. https://doi.org/10.1007/978-3-319-21966-0_11
[29] Peeter Laud and Alisa Pankova. 2014. Verifiable Computation in Multiparty Protocols with Honest Majority. In *Provable Security - 8th International Conference, ProvSec 2014. Proceedings (LNCS)*, Sherman S. M. Chow, Joseph K. Liu, Lucas Chi Kwong Hui, and Siu-Ming Yiu (Eds.), Vol. 8782. Springer, 146–161. https://doi.org/10.1007/978-3-319-12475-9_11
[30] Peeter Laud, Alisa Pankova, and Roman Jagomägis. 2017. Preprocessing Based Verification of Multiparty Protocols with Honest Majority. *PoPETs* 2017, 4 (2017), 23–76. https://doi.org/10.1515/popets-2017-0038
[31] Peeter Laud, Alisa Pankova, Liina Kamm, and Meilof Veeningen. 2015. Basic Constructions of Secure Multiparty Computation. In *Applications of Secure Multiparty Computation*, Peeter Laud and Liina Kamm (Eds.). Cryptology and Information Security Series, Vol. 13. IOS Press, 1–25.
[32] Peeter Laud and Jaak Randmets. 2015. A Domain-Specific Language for Low-Level Secure Multiparty Computation Protocols. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*, Indrajit Ray, Ninghui Li, and Christopher Kruegel (Eds.). ACM, 1492–1503. https://doi.org/10.1145/2810103.2813664
[33] Sven Laur, Riivo Talviste, and Jan Willemson. 2013. From Oblivious AES to Efficient and Secure Database Join in the Multiparty Setting. In *Applied Cryptography and Network Security - 11th International Conference, ACNS 2013, Banff, AB, Canada, June 25-28, 2013. Proceedings (Lecture Notes in Computer Science)*, Michael J. Jacobson Jr., Michael E. Locasto, Payman Mohassel, and Reihaneh Safavi-Naini (Eds.), Vol. 7954. Springer, 84–101. https://doi.org/10.1007/978-3-642-38980-1_6
[34] Sven Laur, Jan Willemson, and Bingsheng Zhang. 2011. Round-Efficient Oblivious Database Manipulation. In *Proceedings of the 14th International Conference on Information Security. ISC'11*. 262–277.
[35] Yehuda Lindell and Ariel Nof. 2017. A Framework for Constructing Fast MPC over Arithmetic Circuits with Malicious Adversaries and an Honest-Majority. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM, 259–276. https://doi.org/10.1145/3133956.3133999
[36] Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. 2012. A New Approach to Practical Active-Secure Two-Party Computation, See [39], 681–700. https://doi.org/10.1007/978-3-642-32009-5_40
[37] Takashi Nishide and Kazuo Ohta. 2007. Multiparty Computation for Interval, Equality, and Comparison Without Bit-Decomposition Protocol. In *Public Key Cryptography - PKC 2007, 10th International Conference on Practice and Theory in Public-Key Cryptography, Beijing, China, April 16-20, 2007, Proceedings (Lecture Notes in Computer Science)*, Tatsuaki Okamoto and Xiaoyun Wang (Eds.), Vol. 4450. Springer, 343–360. https://doi.org/10.1007/978-3-540-71677-8_23
[38] Martin Pettai and Peeter Laud. 2015. Automatic Proofs of Privacy of Secure Multi-party Computation Protocols against Active Adversaries. In *IEEE 28th Computer Security Foundations Symposium, CSF 2015, Verona, Italy, 13-17 July, 2015*, Cédric Fournet, Michael W. Hicks, and Luca Viganò (Eds.). IEEE, 75–89. https://doi.org/10.1109/CSF.2015.13
[39] Reihaneh Safavi-Naini and Ran Canetti (Eds.). 2012. *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*. LNCS, Vol. 7417. Springer.
[40] Bharath K. K. Samanthula, Hu Chun, and Wei Jiang. 2013. An Efficient and Probabilistic Secure Bit-decomposition. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security (ASIA CCS '13)*. ACM, New York, NY, USA, 541–546. https://doi.org/10.1145/2484313.2484386
[41] Thomas Schneider and Michael Zohner. 2013. GMW vs. Yao? Efficient Secure Two-Party Computation with Low Depth Circuits. In *Financial Cryptography and Data Security - 17th International Conference, FC 2013, Okinawa, Japan, April 1-5, 2013, Revised Selected Papers (Lecture Notes in Computer Science)*, Ahmad-Reza Sadeghi (Ed.), Vol. 7859. Springer, 275–292. https://doi.org/10.1007/978-3-642-39884-1_23
[42] Berry Schoenmakers and Pim Tuyls. 2006. Efficient Binary Conversion for Paillier Encrypted Values. In *Advances in Cryptology - EUROCRYPT 2006, 25th Annual International Conference on the Theory and Applications of Cryptographic Techniques, St. Petersburg, Russia, May 28 - June 1, 2006, Proceedings (Lecture Notes in Computer Science)*, Serge Vaudenay (Ed.), Vol. 4004. Springer, 522–537. https://doi.org/10.1007/11761679_31
[43] Adi Shamir. 1979. How to Share a Secret. *Commun. ACM* 22, 11 (1979), 612–613.
[44] Tomas Toft. 2009. Constant-Rounds, Almost-Linear Bit-Decomposition of Secret Shared Values. In *Topics in Cryptology - CT-RSA 2009, The Cryptographers' Track at the RSA Conference 2009, San Francisco, CA, USA, April 20-24, 2009. Proceedings (Lecture Notes in Computer Science)*, Marc Fischlin (Ed.), Vol. 5473. Springer, 357–371. https://doi.org/10.1007/978-3-642-00862-7_24
[45] Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi (Eds.). 2016. *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. ACM. http://dl.acm.org/citation.cfm?id=2976749