Sharemind: a framework for fast privacy-preserving computations

Dan Bogdanov[1,2], Sven Laur[1], and Jan Willemson[1,2]

[1] University of Tartu
{db,swen,jan}@ut.ee
[2] AS Cybernetica

Abstract. Gathering and processing sensitive data is a difficult task. In fact, there is no common recipe for building the necessary information systems. In this paper, we present a provably secure and efficient general-purpose computation system to address this problem. Our solution—SHAREMIND—is a virtual machine for privacy-preserving data processing that relies on share computing techniques. This is a standard way for securely evaluating functions in a multi-party computation environment. The novelty of our solution is in the choice of the secret sharing scheme and the design of the protocol suite. We have made many practical decisions to make large-scale share computing feasible in practice. The protocols of SHAREMIND are information-theoretically secure in the honest-but-curious model with three computing participants. Although the honest-but-curious model does not tolerate malicious participants, it still provides significantly increased privacy preservation when compared to standard centralised databases.

1 Introduction

Recent years have significantly altered the methodology of collecting and processing information. The large-scale adoption of online information systems has made both the use and abuse of personal data easier than before. This has caused an increased awareness about privacy issues among individuals. In many countries, databases containing personal, medical or financial information about individuals are classified as sensitive and the corresponding laws specify who can collect and process sensitive information about a person.

On the other hand, the usage of sensitive information plays an essential role in medical, financial and social studies. Thus, one needs a methodology for conducting statistical surveys without compromising the privacy of individuals. The corresponding research area is commonly known as privacy-preserving data mining. So far the focus has been on randomised response techniques [AS00,AA01,ESAG02]. In a nutshell, recipients of the statistical survey apply a fixed randomisation method on their responses. As a result, each individual reply is erroneous, whereas the global statistical properties of the data are preserved. Unfortunately, such transformations can preserve privacy only on average and randomisation reduces the precision of the outcomes. Also, we cannot give security guarantees for individual records. In fact, the corresponding guarantees are rather weak and the use of extra information might significantly reduce the level of privacy.

Another option is to consider the problem as a multi-party computation task, where the data donors want to securely aggregate data without revealing their private inputs. However, the corresponding cryptographic solutions quickly become practically intractable when the number of participants grows beyond few hundreds. Moreover, data donors are often unwilling to stay online during the entire computation and their computers can be easily taken over by adversarial forces. As a way out, we propose a hierarchical solution, where all computations are done by dedicated *miner* parties who are less susceptible for external corruption. More precisely, we assume that only a few miner parties can be corrupted during the computation. Consequently, we can use secret sharing and share computing techniques for privacy-preserving data aggregation. In particular, data donors can safely submit their inputs by sending the corresponding shares to the miners. As a result, the miners can securely evaluate any aggregate statistic without further interaction with the data donors.

Our contribution. The presented theoretical solution does not form the core of this paper. Share computing techniques have been known for decades [BOGW88,CCD88,Bea91] and thus all important results are well established by now. Hence, we focused mainly on practical aspects and developed the SHAREMIND framework for privacy-preserving computations. The SHAREMIND framework is designed to be an efficient and easily programmable platform for developing and testing various privacy-preserving algorithms. More precisely, it consists of the computation runtime environment and a programming library for creating private data processing applications. As a result, one can develop secure multi-party protocols without the explicit knowledge of all implementation details. On the other hand, it is also possible to test and add your own protocols to the library, since the source code of SHAREMIND is freely available [SM007].

To assure maximal efficiency, we have made some non-standard choices. First, the SHAREMIND framework uses an additive secret sharing scheme over the ring $\mathbb{Z}_{2^{32}}$. Besides the direct computational gains, such a choice also simplifies many share computing protocols. When a secret sharing protocol is defined over a finite field $\mathbb{Z}_p$, then any overflow in computations causes modular reductions that corrupt the end result. In the SHAREMIND framework, all modular reductions occur modulo 2^{32} and thus results always coincide with the standard 32-bit integer arithmetic. On the other hand, standard share computing techniques are not applicable for the ring $\mathbb{Z}_{2^{32}}$. In particular, we were forced to roll out our own multiplication protocol, see Sect. 4.

Secondly, the current implementation of SHAREMIND supports the computationally most efficient setting, where only one of three miner nodes can be semi-honestly corrupted. As discussed in Sect. 3, the corresponding assumption can be enforced with a reasonable practical effort. Additionally, we discuss extending the system to more complex settings in Sect. 7.

To make the presentation more fluent, we describe the SHAREMIND framework step by step. Sect. 2 covers all essential cryptographic notions starting from secret sharing and ending with the necessary composability results. Sect. 3 gives a high-level description of the framework. Details of share computing are explained in Sect. 4. Next, Sect. 5 gives a brief overview of how the framework is implemented and how one could use it in privacy-preserving computations. In Sect. 6, we present and analyse the performance results. In particular, we compare our results with other implementations of privacy-preserving computations [MNPS04,BDJ$^+$06,YWS06]. Finally, we conclude our presentation with some improvement plans for future, see Sect. 7.

2 Cryptographic Preliminaries

Theoretical attack model. In this article, we state and prove all security guarantees in the *information-theoretical setting*, where each participant pair is connected with a private communication channel that provides asynchronous communication. In other words, a potential adversary can only delay or reorder messages without reading them. We also assume that the communication links are authentic, i.e., the adversary cannot send messages on behalf of non-corrupted participants. The adversary can corrupt participants during the execution of a protocol. In the case of *semi-honest* corruption, the adversary can only monitor the internal state of a corrupted participant, whereas the adversary has full control over *maliciously* corrupted participants. We consider only *threshold adversaries* that can adaptively corrupt up to t participants. Such an attack model is well established, see [BOCG93,BOKR94,HM00] for further details.

As a second restriction, we consider only self-synchronising protocols, where the communication can be divided into distinct rounds. A protocol is *self-synchronising* if the adversary cannot force (semi-)honest participants to start a new communication round until all other participants have completed the previous round. As a result, we can analyse the security of these protocols in the standard synchronised setting with a rushing adversary [CCD88,Bea91], where all communication rounds take place at pre-agreed time intervals and the corrupted participants can send out their messages after receiving messages form honest participants.

Secure multi-party computation. Assume that participants $\mathcal{P}_1, \ldots, \mathcal{P}_n$ want to compute outputs $y_i = f_i(x_1, \ldots, x_n)$ where $x_1, \ldots, x_n$ are corresponding private inputs. Then the security of a protocol π that implements the described functionality is defined by comparing the protocol with the ideal implementation π°, where all participants submit their inputs $x_1, \ldots, x_n$ securely to the trusted third party $\mathcal{T}$ that computes the necessary outputs $y_i = f_i(x_1, \ldots, x_n)$ and sends $y_1, \ldots, y_n$ securely back to the respective participants. A malicious participant $\mathcal{P}_i$ can halt the ideal protocol π° by submitting $x_i = \perp$. Then the trusted third party $\mathcal{T}$ sends $\perp$ as an output for all participants. Now a protocol π is secure if for any plausible attack $\mathcal{A}$ against the protocol π there exists a plausible attack $\mathcal{A}^\circ$ against the protocol π° that causes comparable damage.

For brevity, let us consider only the stand-alone setting, where only a single protocol instance is executed and all honest participants carry out no side computations. Let $\phi_i = (\sigma_i, x_i)$ denote the entire input state of $\mathcal{P}_i$ and let $\psi_i = (\phi_i, y_i)$ denote the entire output state. Similarly, let ϕ_{a} and ψ_{a} denote the inputs and outputs of the adversary and $\boldsymbol{\phi} = (\phi_1, \ldots, \phi_n, \phi_{\mathrm{a}})$, $\boldsymbol{\psi} = (\psi_1, \ldots, \psi_n, \psi_{\mathrm{a}})$ the corresponding input and output vectors. Then a protocol π is *perfectly secure* if for any plausible τ_{re}-time real world adversary $\mathcal{A}$ there exists a plausible τ_{id}-time ideal world adversary $\mathcal{A}^\circ$ such that for any input distribution $\boldsymbol{\phi} \leftarrow \mathfrak{D}$ the corresponding output distributions $\boldsymbol{\psi}$ and $\boldsymbol{\psi}^\circ$ in the real and ideal world coincide and the running times τ_{re} and τ_{id} are comparable.

The efficiency constraint has several interpretations. In the asymptotic setting, the running times are *comparable* if τ_{id} is polynomial in τ_{re}. For fixed time bound τ_{re}, one must decide an acceptable time bound τ_{id} by herself. In either case, all security proofs in this article are suitable, since they assure that $\tau_{\mathrm{id}} \leq c \cdot \tau_{\mathrm{re}}$ where c is relatively small constant.

In our setting, a real world attack $\mathcal{A}$ is plausible if it corrupts up to t participants. The corresponding ideal world attack $\mathcal{A}^\circ$ is plausible if it corrupts the same set of participants as the real world attack. For further details and standard results, see [Bea91,DM00,Can00,Can01].

Universal composability. Complex protocols are often designed by combining several low level protocols. Unfortunately, the stand-alone security is not enough to prove the security of the compound protocol and we must use more stringent security definitions. More formally, let $\varrho\langle\cdot\rangle$ be a global context that uses the functionality of a protocol π. Then we can compare real and ideal world protocols $\varrho\langle\pi\rangle$ and $\varrho\langle\pi^\circ\rangle$. Let $\boldsymbol{\phi}$, $\boldsymbol{\psi}$, $\boldsymbol{\psi}^\circ$ denote the input and output vectors of the compound protocols $\varrho\langle\pi\rangle$ and $\varrho\langle\pi^\circ\rangle$. Then a protocol π is *perfectly universally composable* if for any plausible τ_{re}-time attack $\mathcal{A}$ against $\varrho\langle\pi\rangle$ there exists a plausible τ_{id}-time attack $\mathcal{A}^\circ$ against $\varrho\langle\pi^\circ\rangle$ such that for any input distribution $\boldsymbol{\phi} \leftarrow \mathfrak{D}$ the output distributions $\boldsymbol{\psi}$ and $\boldsymbol{\psi}^\circ$ coincide and the running times τ_{re} and τ_{id} are comparable. See the manuscript [Can01] for a more formal and precise treatment.

Secret sharing schemes. Many solutions for secure multiparty computations are based on secret sharing. A *secret sharing scheme* is determined by a randomised sharing algorithm $\mathsf{Deal} : \mathcal{M} \rightarrow \mathcal{S}_1 \times \ldots \times \mathcal{S}_n$ and a recovery algorithm $\mathsf{Rec} : \mathcal{S}_1 \times \ldots \times \mathcal{S}_n \rightarrow \mathcal{M} \cup \{\perp\}$ where $\mathcal{M}$ is the set of possible secrets and $\mathcal{S}_1, \ldots, \mathcal{S}_n$ are the sets of possible shares. For brevity, we use a shorthand $[\![s]\!]$ to denote the shares $[s_1, \ldots, s_n]$ generated by the sharing algorithm $\mathsf{Deal}(s)$.

A *k-out-of-n threshold secret sharing scheme* must satisfy two additional requirements. First, the reconstruction algorithm must output a correct value for all shares $[\hat{s}_1, \ldots, \hat{s}_n]$ that are obtained from a valid sharing $[s_1, \ldots, s_n]$ by altering at most $n - k$ original shares. Second, the distribution of $k - 1$ shares $s_{i_1}, \ldots, s_{i_{k-1}}$ must be always independent of the secret s, i.e., a collection of $k - 1$ shares reveals no information about the secret s.

3 Privacy-Preserving Data Aggregation

As already emphasised in the introduction, one often needs to gather and process sensitive data that belongs to other persons or organisations. The latter can lead to serious security issues, as the organisations who collect and process the data may abuse it or reveal the data to third parties. As a result, people are unwilling to reveal sensitive information without strong security guarantees. In many countries, this issue is regulated by laws that specify who and how can gather and process sensitive data. Although proper legislation and auditing reduce the corresponding risks, data donors must still *unconditionally* trust government institutions that gather and process data. In the following, we introduce a framework for privacy-preserving computations that eliminates the need for unconditional trust. Our SHAREMIND framework uses secret sharing to split confidential information between several nodes (*miners*). As a result, data donors do not have to trust any of them. Instead, it is sufficient to believe that the number of collaborating corrupted nodes is below the prescribed threshold t. The latter can be achieved with physical and organisational security measures such as dedicated server rooms and software auditing.

By sending the shares of the data to the miners, data donors effectively delegate all rights over the data to the consortium of miners. Furthermore, it is relatively easy to assure that a set of miners fulfils the necessary security requirements, whereas it is essentially impossible to make justified assumptions about all data donors. As a result, protocols without *trust transfer* must be secure even if the majority of data donors are malicious. Although such protocols for honest minority do exist [Yao86,GL90], they have many drawbacks [CK89,CKL03]. First, we cannot construct protocols, where data donors just submit their inputs. Instead, they have to be active throughout the entire computational process. Secondly, we must use computationally expensive cryptographic primitives to assure provable security. Therefore, the corresponding protocols quickly become practically intractable if the number of analysed data items grows beyond few thousand records.

Note that physical and organisational security measures are sufficient to assure semi-honest behaviour if there are only a few miners. Indeed, it is not too far-fetched to assume that potential adversaries can install only passive logging programs into well-protected miner nodes. Classical limits proved by Chor and Kushilevitz [CK89] indicate that the use of computationally slow cryptographic primitives is unavoidable if semi-honestly corrupted participants are in majority. Hence, it is advantageous to consider settings, where the total number of miners n is larger than $2t$. Analogous theoretical limits for malicious corruption [CCD88,BOGW88] imply that $n > 3t$ or we again have to use relatively slow cryptographic primitives.

Although a larger number of miner nodes increases the level of tolerable corruption, it also makes assuring semi-honest behaviour much more difficult. In the reality, it is difficult to find more than five independent organisations that can provide adequate protection measures and are not motivated to collaborate with each other. Also, note that the communication complexity of multi-party computation protocols is roughly quadratic in the number of miners n. For these reasons, only models with three to five miner nodes are relevant in practice.

The main goal of the SHAREMIND framework is to provide efficient protocols for basic mathematical operations so that we could easily implement more complex tasks. In particular, one should be able to construct such protocols without any knowledge about underlying cryptographic techniques. For that reason, all implementations of basic operations in the SHAREMIND framework are universally composable. Secondly, many design choices are made to assure maximal efficiency. For instance, the current implementation of SHAREMIND consists of three miner nodes, as it is the most communication-efficient model that can handle semi-honest corruption of a single node.

To achieve maximal efficiency, we also use non-orthodox secret sharing and share computing protocols. Recall that most classical secret sharing schemes work over finite

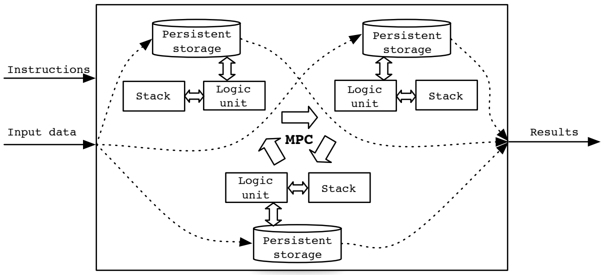

Fig. 1. Deployment diagram of Sharemind. The input data and instructions are delivered to miner nodes that use multi-party computation techniques to execute programs step by step and finally return the end results.

fields. As a result, it is easy to implement secure addition and multiplication modulo prime p or in the Galois field $\mathsf{GF}(2^k)$. However, the integer arithmetic in modern computers is done modulo 2^{32}. Hence, the most space- and time-efficient solution is to use additive secret sharing schemes over $\mathbb{Z}_{2^{32}}$. There is no need to implement modular arithmetic and we do not have to compensate the effect of modular reductions. On the other hand, we have to use non-standard methods for share computing, since Shamir secret sharing scheme does not work over $\mathbb{Z}_{2^{32}}$. We discuss these issues further in Sect. 4.

The high level description of the SHAREMIND framework is depicted in Fig. 1. Essentially, one can view SHAREMIND as a virtual processor that provides secure storage for shared inputs and performs privacy-preserving operations on them. Each miner node $\mathcal{P}_i$ has a local *database* for persistent storage and a local *stack* for storing intermediate results. All values in the database and stack are shared among all miners $\mathcal{P}_1, \ldots, \mathcal{P}_n$ by using an *additive secret sharing* over $\mathbb{Z}_{2^{32}}$. That is, each miner $\mathcal{P}_i$ has a local share s_i of a secret value s such that

$$s_1 + s_2 + \cdots + s_n \equiv s \mod 2^{32}$$

and any $n-1$ element subset $\{s_{i_1}, \ldots, s_{i_{n-1}}\}$ is uniformly distributed. Initially, the database is empty and data donors have to submit their inputs by sending the corresponding shares privately to miners who store them in the database. We describe this issue more thoroughly in Sect. 4.2.

After the input data is collected, a data analyst can start privacy-preserving computations by sending instructions to the miners. Each instruction is a command that either invokes a share computing protocol or just reorders shares. The latter allows a data analyst to specify complex algorithms without thinking about implementation details. More importantly, the corresponding complex protocol is guaranteed to preserve privacy, as long as the execution path in the program itself does not reveal private information. This restriction must be taken into account when choosing data analysis algorithms for implementation on SHAREMIND.

Each arithmetic instruction invokes a secure multi-party protocol that provides new shares. These shares are then stored on the stack. For instance, a unary stack instruction

f takes the top shares $[\![u]\!]$ of the stack and pushes the resulting shares $[\![f(u)]\!]$ to the stack top. Analogously, a binary stack instruction $\otimes$ takes two top most shares $[\![u]\!]$ and $[\![v]\!]$ and pushes $[\![u \otimes v]\!]$ to the stack. For efficiency reasons, we have also implemented vectorised operations to perform the same protocol in parallel with many inputs at once. This significantly reduces the number of rounds required for applying similar operations on large quantities of data.

The current implementation of SHAREMIND framework provides privacy preserving addition, multiplication and greater-than-or-equal comparison of two shared values. It can also multiply a shared value with a constant and extract its bits as shares. Share conversion from $\mathbb{Z}_2$ to $\mathbb{Z}_{2^{32}}$ and bitwise addition are mostly used as components in other protocols, but they are also available to the programmer. We emphasise here that many algorithms for data mining and statistical analysis do not use other mathematical operations and thus this instruction set is sufficient for most applications. Moreover, note that bit extraction and arithmetic primitives together are sufficient to implement any Boolean circuit with a linear overhead and thus the SHAREMIND framework is also Turing complete. We acknowledge here that there are more efficient ways to evaluate Boolean circuits like Yao circuit evaluation [Yao86,LP04] and we plan to include protocols with similar properties in the future releases of SHAREMIND.

We analyse the security of all share manipulation protocols in the information-theoretical attack model that was specified in Sect. 2. In particular, we assume that miners are connected to each other via secure channels. We can use strong symmetric encryption and authentication primitives to achieve a similar level of security in practical applications. As a result, we obtain only computational security guarantees in the real world. The latter is unavoidable if we want to achieve a cost-efficient and universal solution, as building dedicated secure channels is currently expensive. As a final detail, note that asynchronous communication allows us to omit the central synchronisation service and thus reduce network delays that have significant impact on the overall efficiency.

4 Share Computing Protocols

All computational instructions in the SHAREMIND framework are either unary or binary operations over unsigned integers represented as elements of $\mathbb{Z}_{2^{32}}$ or their vectorised counterparts. Hence, all protocols described in this section have the following structure. Each miner $\mathcal{P}_i$ uses shares u_i and v_i as inputs to the protocol to obtain a new share w_i such that $[\![w]\!]$ is a valid sharing of $f(u)$ or $u \otimes v$. In the corresponding idealised implementation, all miners send their input shares to the trusted third party $\mathcal{T}$ who restores all inputs, computes the corresponding output w and sends back newly computed shares $[\![w]\!] \leftarrow \mathsf{Deal}(w)$. Hence, the output shares $[\![w]\!]$ are independent of input shares and thus no information is leaked about the input shares if we publish all output shares $[\![w]\!]$.

Since the current implementation of SHAREMIND framework consists of three miner nodes, we present here only three-party protocols that are secure against a semi-honest corruption of a single node. Similar constructions are possible for other settings, see [CFIK03] for a generic solution.

Although share computing protocols are often used as elementary steps in more complex protocols, they themselves can be composed from even smaller atomic operations. Many of these atomic sub-protocols produce output shares that are never published. Hence, it makes sense to introduce another security notion that is weaker than universal composability. We say that a share computing protocol is *perfectly simulatable* if there exists an efficient universal non-rewinding simulator $\mathcal{S}$ that can simulate all protocol messages to any real world adversary $\mathcal{A}$ so that for all input shares the output distributions of $\mathcal{A}$ and $\mathcal{S}\langle\mathcal{A}\rangle$ coincide. Most importantly, perfect simulatability is closed under concurrent composition. The corresponding proof is straightforward.

Lemma 1. *A protocol is perfectly simulatable if all its sub-protocols are perfectly simulatable.*

Proof (Sketch). Since all simulators $\mathcal{S}_i$ of sub-protocols are non-rewinding, we can construct a single compound simulator $\mathcal{S}_*$ that runs simulators $\mathcal{S}_i$ in parallel to provide the missing messages to $\mathcal{A}$. As each simulator $\mathcal{S}_i$ is perfect, the final view of $\mathcal{A}$ is also perfectly simulated. □

However, perfect simulatability alone is not sufficient for universal composability. Namely, output shares of a perfectly simulatable protocol may depend on input shares. As a result, published shares may reveal more information about inputs than necessary. Therefore, we must often re-share the output shares at the end of each protocol. The corresponding ideal functionality is modelled as follows. Initially, the miners send their shares $[\![u]\!]$ to the trusted third party $\mathcal{T}$. Then the party $\mathcal{T}$ recovers $u \leftarrow \mathsf{Rec}([\![u]\!])$ and sends new shares $[\![w]\!] \leftarrow \mathsf{Deal}(u)$ back to the miners.

For the basic setting with three miners, the simplest re-sharing protocol is the following. First, each miner $\mathcal{P}_i$ generates a random mask $r_i \leftarrow \mathbb{Z}_{2^{32}}$ and sends it to the right neighbour $\mathcal{P}_{i+1}$ (the miner $\mathcal{P}_3$ sends r_3 to $\mathcal{P}_1$). Next, each miner $\mathcal{P}_i$ outputs $w_i \leftarrow u_i + r_{i-1} - r_i$. As a result, the output shares have indeed the correct distribution $\mathsf{Deal}(u)$. Moreover, for fixed values u_i, w_i, r_i, there is only a single consistent value r_{i-1} in the real execution. Hence, we can construct a non-rewinding *interface* $\mathcal{I}_0$ between the ideal world and a real world adversary $\mathcal{A}$. The interface $\mathcal{I}_0$ forwards the input share u_i of corrupted miner $\mathcal{P}_i$ to the trusted third party $\mathcal{T}$, provides randomness $r_i \leftarrow \mathbb{Z}_{2^{32}}$ to $\mathcal{P}_i$, and given w_i form $\mathcal{T}$ sends $r_{i-1} \leftarrow w_i - u_i + r_i$ to the corrupted party $\mathcal{P}_i$. As the trusted third party $\mathcal{T}$ draws the output shares from the correct distribution, the simulation is perfect and the output distributions $\boldsymbol{\psi}$ and $\boldsymbol{\psi}^\circ$ in the real and the ideal world coincide. Moreover, the output distributions $\boldsymbol{\psi}$ and $\boldsymbol{\psi}^\circ$ remain identical even if we execute the re-sharing protocol as a sub-task in larger computational context $\varrho\langle\cdot\rangle$, as the interface $\mathcal{I}_0$ is non-rewinding (see the manuscript [Can01] for further details.). Also, note that the construction of the interface makes sense only in the semi-honest model, where the adversary cannot alter the value of r_i.

The next lemma shows that perfect simulatability together with re-sharing assures universal composability in the semi-honest model. In the malicious model, one needs extractability of inputs and additional correctness guarantees against malicious behaviour.

Lemma 2. *A perfectly simulatable share computing protocol that ends with perfectly secure re-sharing of output shares is perfectly universally composable.*

Proof. Let $\mathcal{S}$ be the perfect simulator for the share computing phase and $\mathcal{I}_0$ the interface for the re-sharing protocol. Then we can construct a new non-rewinding interface $\mathcal{I}$ for the whole protocol:

1. It first submits the inputs of the corrupted miners $\mathcal{P}_i$ to the trusted third party $\mathcal{T}$ and gets back the output shares w_i.
2. Next, it runs possibly in parallel the simulator $\mathcal{S}$ and the interface $\mathcal{I}_0$ with the output shares w_i to simulate the missing protocol messages.

Now the output distributions $\boldsymbol{\psi}$ and $\boldsymbol{\psi}^\circ$ coincide, since the sub-routines $\mathcal{S}$ and $\mathcal{I}_0$ perfectly simulate protocol messages and $\mathcal{I}_0$ assures that the output shares of corrupted parties are indeed w_i. The latter assures that the adversarial output ψ°_{a} is correctly matched together with the outputs of honest parties. Since the interface $\mathcal{I}$ is non-rewinding, the claim holds even if the protocol is executed in a larger computational context $\varrho\langle\cdot\rangle$. □

4.1 Protocols for Atomic Operations

As the SHAREMIND framework is based on additive sharing, it is straightforward to implement share addition and multiplication with a public constant c by doing only local operations, since $[u_1 + v_1, \ldots, u_n + v_n]$ and $[cu_1, \ldots, cu_n]$ are valid shares of $u + v$ and cu. However, note that these operations are only perfectly simulatable, since the output shares depend on input shares.

A share multiplication protocol is another important atomic primitive. Unfortunately, we cannot use the standard solutions based on polynomial interpolation and re-sharing, since Shamir secret sharing scheme fails in the ring $\mathbb{Z}_{2^{32}}$. Hence, we must roll out our own multiplication protocol. By the definition of the additive secret sharing scheme

$$uv = \sum_{i=1}^{n} u_i v_i + \sum_{j \neq i}^{n} u_i v_i \mod 2^{32} \tag{1}$$

and thus we need sub-protocols for computing shares of $u_i v_j$. For clarity and brevity, we consider only a sub-protocol, where $\mathcal{P}_1$ has an input x_1 and $\mathcal{P}_2$ has an input x_2 and the miner $\mathcal{P}_3$ helps the others to obtain shares of $x_1 x_2$. Du and Atallah were the first to publish the corresponding protocol [DA00] although similar reduction techniques have been used earlier. Fig. 2 depicts the corresponding protocol. Essentially, the correctness of the protocol relies on the observation

$$x_1 x_2 = -(x_1 + \alpha_1)(x_2 + \alpha_2) + x_1(x_2 + \alpha_2) + (x_1 + \alpha_1)x_2 + \alpha_1\alpha_2 \enspace .$$

The security follows from the fact that for uniformly and independently generated $\alpha_1, \alpha_2 \leftarrow \mathbb{Z}_{2^{32}}$ the sums $x_1 + \alpha_1$ and $x_2 + \alpha_2$ have also uniform distribution.

Lemma 3. *The Du-Atallah protocol depicted in Fig. 2 is perfectly simulatable.*

Proof. Let us fix inputs x_1 and x_2. Then $\mathcal{P}_1$ receives two independent uniformly distributed values and $\mathcal{P}_2$ receives two independent uniformly distributed values. $\mathcal{P}_3$ receives no values at all. Hence, it is straightforward to construct a simulator $\mathcal{S}$ that simulates the view of a semi-honest participant. □

1. $\mathcal{P}_3$ generates $\alpha_1, \alpha_2 \leftarrow \mathbb{Z}_{2^{32}}$ and sends α_1 to $\mathcal{P}_1$ and α_2 to $\mathcal{P}_2$.
2. $\mathcal{P}_1$ computes $x_1 + \alpha_1$ and sends the result to $\mathcal{P}_2$.
   $\mathcal{P}_2$ computes $x_2 + \alpha_2$ and sends the result to $\mathcal{P}_1$.
3. Parties compute shares of $x_1 x_2$:
   (a) $\mathcal{P}_1$ computes its share $w_1 = -(x_1 + \alpha_1)(x_2 + \alpha_2) + x_1(x_2 + \alpha_2)$
   (b) $\mathcal{P}_2$ computes its share $w_2 = (x_1 + \alpha_1)x_2$
   (c) $\mathcal{P}_3$ computes its share $w_3 = \alpha_1 \alpha_2$

Fig. 2. Du-Atallah multiplication protocol

Execute the following protocols concurrently:

- Compute locally shares $u_1 v_1$, $u_2 v_2$ and $u_3 v_3$.
- Use six instances of the Du-Atallah protocol for computing shares of $u_i v_j$ where $i \neq j$.
- Re-share the final sum of all previous sub-output shares.

Fig. 3. High-level description of the share multiplication protocol.

Fig. 3 depicts a share multiplication protocol that executes six instances of the Du-Atallah protocol in parallel to compute the right side of the equation (1). Since the protocols are executed concurrently, the resulting protocol has only three rounds, see App. B for the explicit description.

Theorem 1. *The multiplication protocol depicted in Fig. 3 is perfectly universally composable.*

Proof. Lemma 1 assures that the whole protocol is perfectly simulatable, as the local computations and the instances of Du-Atallah protocol are perfectly simulatable. Since the output shares are re-shared, Lemma 2 assures that the protocol is also universally composable. □

The described multiplication protocol requires a total of three rounds and 27 messages. For comparison, note that the classical multiplication protocol based on Shamir secret sharing has two rounds and 9 messages. On the other hand, recall that Shamir secret sharing is not possible over $\mathbb{Z}_{2^{32}}$ and computing over $\mathbb{Z}_p$ requires additional safeguarding measures against modular reductions. Moreover, the first round of our multiplication protocol is independent of inputs and can be precomputed. Hence, the effective round complexity of both protocols is still comparable.

4.2 Protocol for Input Gathering

Many protocols can be directly built on the atomic operations described in the previous sub-section. As the first example, we discuss methods for input validation. Recall that initially the database of shared inputs is empty in the SHAREMIND framework and the data donors have to fill it. There are two aspects to note. First, the data donors might be malicious and try to construct fraudulent inputs to influence data aggregation procedures. For instance, some participants of polls might be interested in artificially increasing the support of their favourite candidate. Secondly, the data donors want to submit their data as fast as possible without extra work. In particular, they are unwilling to prove that their inputs are in the valid range.

There are two principal ways to address these issues. First, the miners can use multiparty computation protocols to detect and eliminate fraudulent entries. This is computationally expensive, since the evaluation of correctness predicates is a costly operation. Hence, it is often more advantageous to use such an input gathering procedure that guarantees validity by design. For instance, many data tables consist of binary inputs (yes-no answers). Then we can gather inputs as shares over $\mathbb{Z}_2$ to avoid fraudulent inputs and later use share conversion to get the corresponding shares over $\mathbb{Z}_{2^{32}}$.

Let $[u_1, u_2, u_3]$ be a valid additive sharing over $\mathbb{Z}_2$. Then we can express the shared value u through the following equation over integers:

$$f(u_1, u_2, u_3) := u_1 + u_2 + u_3 - 2u_1u_2 - 2u_1u_3 - 2u_2u_3 + 4u_1u_2u_3 = u \enspace .$$

Consequently, if we treat u_1, u_2, u_3 as inputs and compute the shares of $f(u_1, u_2, u_3)$ over $\mathbb{Z}_{2^{32}}$, then we obtain the desired sharing of u. More precisely, we can use the Du-Atallah protocol to compute the shares $[\![u_1u_2]\!]$, $[\![u_1u_3]\!]$, $[\![u_2u_3]\!]$ over $\mathbb{Z}_{2^{32}}$. To get the shares $[\![u_1u_2u_3]\!]$, we use the share multiplication protocol to multiply $[\![u_1u_2]\!]$ and the shares $[\![u_3]\!]$ created by $\mathcal{P}_3$. Finally, all parties use local addition and multiplication routines to obtain the shares of $f(u_1, u_2, u_3)$ and then re-share them to guarantee the universal composability. The resulting protocol has only four rounds, since we can start the first round of all multiplication protocols simultaneously, see App. C for further details. The total number of exchanged messages is 50.

Theorem 2. *The share conversion protocol is perfectly universally composable.*

Proof. The proof follows again directly from Lemmata 1 and 2, since all sub-protocols are perfectly simulatable and the output shares are re-shared at the end of the protocol. □

As a final detail, note that input gathering can even be an off-line event, if we make use of public-key encryption. Namely, if everybody knows the public keys of the miners, then they can encrypt the shares with the corresponding public keys and then store them in a public database. Later miners can fetch and decrypt their individual shares and fill their input database.

4.3 Protocols for Bit Extraction and Comparison

Various routines for bit manipulations form another set of important operations. In particular, note that for signed representation of $\mathbb{Z}_{2^{32}} = \{-2^{31}, \ldots, 0, \ldots, 2^{31} - 1\}$ the highest bit indicates the sign and thus the evaluation of greater-than-or-equal (GTE) predicate can be reduced to bit extraction operations. In the following, we mimic the generic scheme proposed by Damgård et al [DFK$^+$06] for implementing bit-level operations. As this construction is given in terms of atomic primitives, it can be used also for settings where there are more than three miners.

As the first step, note that it is easy to generate shares of a random number $r \leftarrow \mathbb{Z}_{2^{32}}$ such that we also know the corresponding bit shares $\llbracket r^{(31)} \rrbracket, \ldots, \llbracket r^{(0)} \rrbracket$. Each miner $\mathcal{P}_i$ first chooses 32 random bits $\overline{r}_i^{(31)}, \ldots, \overline{r}_i^{(0)}$. After that the miners use the share conversion protocol to convert the corresponding shares $\llbracket \overline{r}^{(31)} \rrbracket, \ldots, \llbracket \overline{r}^{(0)} \rrbracket$ over $\mathbb{Z}_2$ to the shares of $\llbracket r^{(31)} \rrbracket, \ldots, \llbracket r^{(0)} \rrbracket$ over $\mathbb{Z}_{2^{32}}$. Finally, the miners use local operations to compute shares

$$\llbracket r \rrbracket = 2^{31} \cdot \llbracket r^{(31)} \rrbracket + 2^{30} \cdot \llbracket r^{(31)} \rrbracket + \cdots + 2^{1} \cdot \llbracket r^{(1)} \rrbracket + 2^{0} \cdot \llbracket r^{(0)} \rrbracket \enspace .$$

As the second step, we construct a bit extraction protocol that given input shares $\llbracket u \rrbracket$ produces shares of its bits $\llbracket u^{(31)} \rrbracket, \ldots, \llbracket u^{(0)} \rrbracket$. This is a versatile operation that can be used as a building block for many bit-based operations. The corresponding protocol is following. First, the miners generate shares $\llbracket r \rrbracket$ of a random number with known bit decomposition. Then they compute the difference $\llbracket a \rrbracket = \llbracket u \rrbracket - \llbracket r \rrbracket$ and publish the corresponding shares. Since the difference has uniform distribution, the value a leaks no information about inputs. As a result, we have reduced the problem of extracting the bits of u to computing the bitwise sum of $a + r$, as each miner knows the public value a and has the shares of random bits $r^{(31)}, \ldots, r^{(0)}$.

Computing bitwise sum is a more complicated procedure, since we need to take care of carry bits. The bitwise addition protocol takes in k element share vectors $\llbracket u^{(k)} \rrbracket, \ldots, \llbracket u^{(0)} \rrbracket$ and $\llbracket v^{(k)} \rrbracket, \ldots, \llbracket v^{(0)} \rrbracket$ and outputs shares $\llbracket w^{(k)} \rrbracket, \ldots, \llbracket w^{(0)} \rrbracket$ such that $w^{(i)}$ is the i-th bit of the sum $u + v$. As all inputs are bits, we can iteratively compute the shares $\llbracket w^{(i)} \rrbracket$ using the classical addition algorithm. However, the corresponding protocol has $\Theta(k)$ rounds, since we cannot compute carry bits locally.

A more round efficient alternative is to use the standard look-ahead carry construction to perform the carry computations in parallel. By the recursive application of the carry look-ahead principle, we can reduce the number of rounds to $\Theta(\log k)$. The corresponding iterative algorithm together with the explanation of the carry look-ahead principle is given in App. A

Again, we can execute some of the sub-protocols in parallel and thus save some rounds. Namely, the resulting bitwise addition protocol has 8 rounds and 87 messages. The corresponding bit extraction protocol has 12 rounds and sends out 139 messages. Note that although each of these messages contains more than one value, we can consider them as a single message for performance considerations. Both protocols are also universally composable, since all sub-protocols are universally composable. Hence we can see that the following theorem holds.

Theorem 3. *The bitwise addition protocol is perfectly universally composable. The share extraction protocol is perfectly universally composable.*

Our approach has many virtues compared to solutions using Shamir scheme. For example, the solution presented in [DFK$^+$06] requires a complex sampling procedure to construct uniformly chosen values $r \leftarrow \mathbb{Z}_p$ which still have a known bit decomposition. Also, we do not have to eliminate the effects of modular reduction when the variables are shared over $\mathbb{Z}_{2^{32}}$. In particular, we can safely compute the bitwise sum of $a + r - p$, even if $a + r \geq p$. Hence, our share extraction protocol is simpler than the protocol proposed in [DFK$^+$06].

Finally, we describe how to implement GTE predicate if both arguments are guaranteed to be in $\mathbb{Z}_{2^{31}} \subseteq \mathbb{Z}_{2^{32}}$. The latter allows us to define the predicate GTE : $\mathbb{Z}_{2^{31}} \times \mathbb{Z}_{2^{31}} \to \{0, 1\}$ as follows:

$$\textsc{Gte}(x, y) = \begin{cases} 1, & \text{if the last bit of the difference } x - y \text{ is 1,} \\ 0, & \text{otherwise.} \end{cases}$$

It is straightforward to see that the definition is correct for unsigned and signed interpretation of the arguments as long as both arguments are in the range $\mathbb{Z}_{2^{31}}$. Since the range $\mathbb{Z}_{2^{31}}$ is sufficient for most practical computations, we have not implemented the extended protocol yet.

Theorem 4. *The greater-than-or-equal protocol is perfectly universally composable.*

Proof. The protocol is universally composable, since the bit extraction protocol that is used to split $x - y$ into bit shares is universally composable. □

5 Practical Implementation

The main goal of the SHAREMIND project is to provide an easily programmable and flexible platform for developing and testing various privacy preserving algorithms based on share computing. The implementation of the SHAREMIND framework provides a library of the most important mathematical primitives described in the previous section. Since these protocols are universally composable, we can use them in any order, possibly in parallel, to implement more complex algorithms. These algorithms remain secure as long as the execution path is independent from input data. Hence, one cannot use explicit if-then constructions. However, this shortcoming is not essential, since all if-then constructions can be rewritten using the following oblivious selection technique:

$$\textbf{if } a \textbf{ then } x \leftarrow y \textbf{ else } x \leftarrow z \quad \Longleftrightarrow \quad x \leftarrow a \cdot y + (1 - a) \cdot z \ .$$

As a result, one can implement a wide range of algorithms.

The software implementation of SHAREMIND is written in the C++ programming language and is tested on Linux, Mac OS X and Windows XP. The “virtual processor” of SHAREMIND consists of the *miner* application which performs the duties of a secure multiparty computation party and the *controller* library for developing controller applications that work with the miners.

In SHAREMIND development environment the miners start with a minimal configuration. The identities of the miners are decided when a controller application connects to them. The controller application will have to know the addresses of the miners and the miners will have to know nothing, as the controller will configure them automatically. All parties know how to exchange messages over a computer network and perform secret sharing operations. Additionally, the miner application knows how to run privacy preserving operations. The controller application executes a program by asking the miners to sequentially execute operations described by the program.

Miners have an operation scheduler for ordering and executing received operations. When a computational operation is requested from the miner, it is scheduled for execution. When the operation is ready to be executed, the miners run the secure multi-party computation protocols necessary for completing the operation. Like in a standard stack machine, all operations read their input data from the stack and write output data to the stack upon completion.

We stress that the described way of running programs on SHAREMIND is perfect for testing and developing algorithms. For real-world applications, we propose a slightly different architecture. To avoid unauthorised commands, the miners must be configured with the identities of each other and all possible controllers. This can be achieved by using public-key infrastructure.

In the development scenario, the controller library interprets programs and passes each subsequent command to the miners. In a real-world application, programs will be signed by an appropriate authority and run by the miners. The controller will request a complete analysis instead of a single operation. This gives the miners more control over program execution and publication of aggregation results and avoids attacks, where the controller can construct malicious queries from basic operations.

6 Performance Results

To evaluate the limitations and bottlenecks of share computing, we have measured the performance of the SHAREMIND framework on various computational tasks. In the following, we present the performance results for scalar product and vectorised comparison. These tests are chosen for two reasons. First, together these benchmarks cover the most important primitives of SHAREMIND: addition, multiplication and comparison. More importantly, scalar product is one of the most fundamental operations used in data mining algorithms. Second, other secure multi-party computation systems [MNPS04,BDJ^{+}06,YWS06] have also implemented at least one of these benchmarks.

Both tests got two n-element vectors $a_1, \ldots, a_n$ and $b_1, \ldots, b_n$ as inputs. In the scalar product test, we computed the sum $a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$. In the comparison test, we evaluated comparisons $a_i \geq b_i$ for $i = 1, \ldots, n$. All input datasets were randomly generated and the corresponding shares were stored in the share databases of SHAREMIND. For each chosen vector size, we performed a number of experiments and measured the results. To identify performance bottlenecks, we measured the time complexity in these three computation phases:

1. the local computation time including scheduling overhead;
2. time spent while sending data to other parties;
3. time spent while waiting for data from other parties.

During the analysis, the timing results were grouped and added to produce the experimental data in this section. The timings were determined at the miners to minimise the impact of the overhead, like arranging input and output data on the stack. This allows us to determine exactly, how long an operation runs before another operation can be executed after it. The tests were performed on four desktop computers in a computing cluster. Each machine contained a dual-core Opteron 175 processor and 2 GB of RAM, and ran Scientific Linux CERN 4.5. The computers were connected by a local switched network allowing communication speeds up to 1 gigabit per second.

As one would expect, the initial profiling results showed that network delays have significant impact on the performance. In fact, the number of rounds almost completely determines the running time provided that the messages sent over the network have reasonable size. Consequently, it is advantageous to execute as many operations in parallel as possible.

The SHAREMIND framework allows the developer to perform several instances of the same operation at once. For example, the developer can multiply or compare two vectors elementwise. The messages exchanged by the parties will be larger, but the computation will be more efficient, because the networking layer can send larger packages with relative ease. Still, vectors may sometimes become too large and this may start to hinder the performance of the networking layer. To balance the effects of vectorisation, we implemented a load balancing system into the SHAREMIND framework. We fixed a certain threshold after which miners start batch processing of large vectorised queries. In each sub-round, a miner processes a fragment of its inputs and sends the results to the other miners before continuing with the next fragment of input data.

Vectorisation and batch processing decrease the time which would otherwise be spent waiting for inputs from the other miners. Note that this does not cause any additional security issues as the composition theorems that apply to our protocols also cover parallel executions. Fig. 4 shows how the optimisations have minimised the penalty caused by network delays.

As seen on Fig. 4 the impact of network delay during scalar product computation is already small—the miners do not waste too many CPU cycles while waiting for inputs. Note, that the performance of the scalar product operation depends directly on the multiplication protocol, because addition is a local operation and therefore very fast. Further optimisations to the implementation of the multiplication protocol can only lead to marginal improvements. For the parallel comparison, the effect of network delay is more important and further scheduling optimisations may decrease the time wasted while waiting for messages. In both benchmarks, the time required to send and receive messages is significant and thus the efficiency of networking can significantly influence performance results.

6.1 Experimental Results and Analysis

The performance data was gathered by running a series of experiments on a data table of the given size. Fig. 4 depicts average running times for test vectors with up to $100,000$ elements in $10,000$-element increments. For the comparison protocol the running times were rather stable, the average standard deviation was approximately 6% from the mean running time. The scalar computation execution time was significantly more unstable, as the average standard deviation over all experiments was 24% of the mean. As most of the variation was in the network delay component of the timings, the fluctuations can be attributed to low-level tasks of the operating system. This is further confirmed by the fact that all scalar product timings are smaller than a second so even

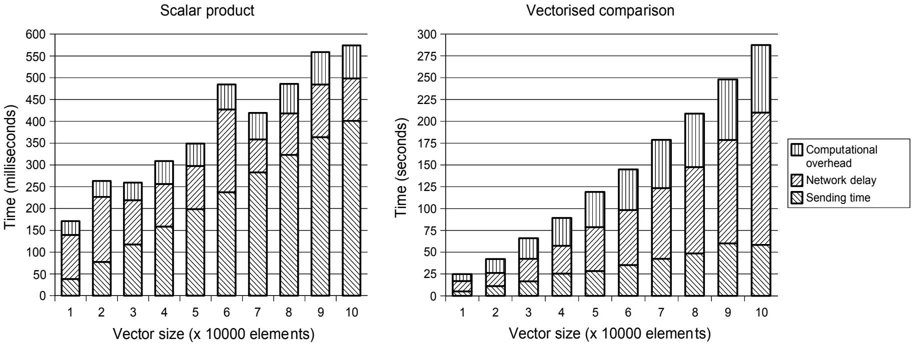

Fig. 4. Performance of the SHAREMIND framework.

relatively small delays made an impact on our execution time. We remind here that the benchmark characterises near-ideal behaviour of the SHAREMIND framework, since no network congestion occurred during the experiments and the physical distance between the computers was small. In real world applications, the effect of network delays and the variance of running times can be considerably larger.

We compared the performance of SHAREMIND with some known implementations of privacy-preserving computations [MNPS04,BDJ$^+$06,YWS06]. The FAIRPLAY system [MNPS04] is a general framework for secure function evaluation with two parties that is based on garbled circuit evaluation. According to the authors, single comparison operation for 32-bit integers takes 1.25 seconds. A single SHAREMIND comparison takes, on average, 500 milliseconds. If we take into account the improvements in hardware we can say that the performance is similar when evaluating single comparisons. The authors of FAIRPLAY noticed that parallel execution gives a speedup factor of up 2.72 times in a local network. Experiments with SHAREMIND have shown that parallel execution can increase execution up to 27 times. Hence, SHAREMIND can perform parallel comparison more efficiently. The experimental scalar product implementation in [YWS06] also works with two parties. However, due to the use of more expensive cryptographic primitives, it is slower than SHAREMIND even with precomputation. For example, computing scalar product of two 100000-element binary vectors takes a minimum of 5 seconds without considering the time of precomputation.

The SCET system used in [BDJ$^+$06] is similar to SHAREMIND as it is also based on share computing. Although SCET supports more than three computational parties, our comparison is based on results with three parties. The authors have presented performance results for multiplication and comparison operations as fitted linear approximations. The approximated time for computing products of x inputs is $3x+41$ milliseconds and the time for evaluating comparisons is $674x + 90$ milliseconds (including precomputation). The performance of SHAREMIND can not be easily linearly approximated, because for input sizes up to 5000 elements parallel execution increases performance significantly more than for inputs with more than 5000 elements. However, based on the presented approximations and our own results we claim that SHAREMIND achieves better performance with larger input vectors in both scalar product and vectorised comparison. A SHAREMIND multiplication takes, on the average, from 0.006 to 57 milliseconds, depending on the size of the vector. More precisely, multiplication takes less than 3 milliseconds for every input vector with more than 50 elements. The timings for comparison range from 3 milliseconds to about half a second which is significantly less than 674 milliseconds per operation.

From the results we see that vectorisation significantly improves the throughput of the system. Performing many similar operations in parallel is more optimal than performing the same number of individual operations. The multiplication protocol scales better than the GTE protocol because of the smaller round and message count. We acknowledge that it might not be possible to rewrite all algorithms to perform operations in parallel, but we note that algorithms which make use of vectorisation will run faster when implemented in the SHAREMIND framework.

7 Conclusion and Future Work

In this paper, we have proposed a novel approach for developing privacy-preserving applications. The SHAREMIND framework relies on secure multi-party computation, but it also introduces several new ideas for improving the efficiency of both the applications and their development process. The main theoretical contribution of the framework is a suite of computation protocols working over elements in the ring of 32-bit integers instead of the standard finite field. This non-standard choice allowed us to build simple and efficient protocols.

We have also implemented a fully functional prototype of SHAREMIND and showed that it offers enhanced performance when compared to other similar frameworks. Besides that, SHAREMIND also has an easy to use application development interface allowing the programmer to concentrate on the implementation of data mining algorithms and not to worry about the privacy issues.

However, the current implementation has several restrictions, most notably it can use only three computing parties and can deal with just one semi-honest adversary. Hence the main direction for future research is relaxing these restrictions by developing computational primitives for more than three parties. We will also need to study the possibilities for providing security guarantees against active adversaries. Another aspect needing further improvement is the application programmer's interface. A compiler from a higher-level language to our current assembly-like instruction set is definitely needed. Implementing and benchmarking a broad range of existing data-mining algorithms will remain the subject for further development as well.

References

[AA01] Dakshi Agrawal and Charu C. Aggarwal. On the design and quantification of privacy preserving data mining algorithms. In *PODS*. ACM, 2001.

[AS00] Rakesh Agrawal and Ramakrishnan Srikant. Privacy-preserving data mining. In Weidong Chen, Jeffrey F. Naughton, and Philip A. Bernstein, editors, *SIGMOD Conference*, pages 439–450. ACM, 2000.

[BDJ+06] Peter Bogetoft, Ivan Damgård, Thomas Jakobsen, Kurt Nielsen, Jakob Pagter, and Tomas Toft. A practical implementation of secure auctions based on multiparty integer computation. In *Financial Cryptography*, volume 4107, 2006.

[Bea91] Donald Beaver. Secure multiparty protocols and zero-knowledge proof systems tolerating a faulty minority. *J. Cryptology*, 4(2):75–122, 1991.

[Bih03] Eli Biham, editor. *Advances in Cryptology - EUROCRYPT 2003, International Conference on the Theory and Applications of Cryptographic Techniques, Warsaw, Poland, May 4-8, 2003, Proceedings*, volume 2656 of *Lecture Notes in Computer Science*. Springer, 2003.

[BOCG93] Michael Ben-Or, Ran Canetti, and Oded Goldreich. Asynchronous secure computation. In *STOC*, pages 52–61, 1993.

[BOGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *STOC* [DBL88], pages 1–10.

[BOKR94] Michael Ben-Or, Boaz Kelmer, and Tal Rabin. Asynchronous secure computations with optimal resilience (extended abstract). In *PODC*, pages 183–192, 1994.

[Can00] Ran Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 13(1):143–202, 2000.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, pages 136–145, 2001.

[CCD88] David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty unconditionally secure protocols (extended abstract). In *STOC* [DBL88], pages 11–19.

[CFIK03] Ronald Cramer, Serge Fehr, Yuval Ishai, and Eyal Kushilevitz. Efficient multi-party computation over rings. In Biham [Bih03], pages 596–613.

[CK89] B. Chor and E. Kushilevitz. A zero-one law for boolean privacy. In *STOC '89: Proceedings of the twenty-first annual ACM symposium on Theory of computing*, pages 62–72, New York, NY, USA, 1989. ACM Press.

[CKL03] Ran Canetti, Eyal Kushilevitz, and Yehuda Lindell. On the limitations of universally composable two-party computation without set-up assumptions. In Biham [Bih03], pages 68–86.

[DA00] Wenliang Du and Mikhail J. Atallah. Protocols for secure remote database access with approximate matching. 7th ACM Conference on Computer and Communications Security (ACMCCS 2000), The First Workshop on Security and Privacy in E-Commerce, Nov. 1-4, 2000. Athens, Greece.

[DBL88] *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing, 2-4 May 1988, Chicago, Illinois, USA*. ACM, 1988.

[DFK+06] Ivan Damgård, Matthias Fitzi, Eike Kiltz, Jesper Buus Nielsen, and Tomas Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In Shai Halevi and Tal Rabin, editors, *TCC*, volume 3876 of *Lecture Notes in Computer Science*, pages 285–304. Springer, 2006.

[DM00] Yevgeniy Dodis and Silvio Micali. Parallel reducibility for information-theoretically secure computation. In Mihir Bellare, editor, *CRYPTO*, volume 1880 of *Lecture Notes in Computer Science*, pages 74–92. Springer, 2000.

[ESAG02] Alexandre V. Evfimievski, Ramakrishnan Srikant, Rakesh Agrawal, and Johannes Gehrke. Privacy preserving mining of association rules. In *KDD*, pages 217–228. ACM, 2002.

[GL90] Shafi Goldwasser and Leonid A. Levin. Fair computation of general functions in presence of immoral majority. In Alfred Menezes and Scott A. Vanstone, editors, *CRYPTO*, volume 537 of *Lecture Notes in Computer Science*, pages 77–93. Springer, 1990.

[HM00] Martin Hirt and Ueli M. Maurer. Player simulation and general adversary structures in perfect multiparty computation. *Journal of Cryptology*, 13(1):31–60, 2000.

[LP04] Yehuda Lindell and Benny Pinkas. A proof of yao's protocol for secure two-party computation. Cryptology ePrint Archive, Report 2004/175, 2004. `http://eprint.iacr.org/`.

[MNPS04] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay - secure two-party computation system. In *USENIX Security Symposium*, pages 287–302. USENIX, 2004.

[SM007] The SHAREMIND project. The official webpage is `http://sharemind.cs.ut.ee`. The source code is published online at `http://www.sourceforge.net/projects/sharemind/`, 2007.

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *FOCS*, pages 162–167, 1986.

[YWS06] Zhiqiang Yang, Rebecca N. Wright, and Hiranmayee Subramaniam. Experimental analysis of a privacy-preserving scalar product protocol. *Comput. Syst. Sci. Eng.*, 21(1), 2006.

A Fast Bitwise Addition Algorithm

The main drawback of the textbook addition algorithm is slow propagation of carries. Let $u^{(k)} \dots u^{(0)}$ and $v^{(k)} \dots v^{(0)}$ be the bit representations of k-bit integers and let $c^{(k)}, \dots, c^{(0)}$ represent the corresponding carry bits. The textbook addition algorithms computes carries iteratively

$$c^{(i)} \leftarrow u^{(i)} \cdot v^{(i)} + (1 - u^{(i)} \cdot v^{(i)}) \cdot c^{(i-1)} \cdot (u^{(i)} + v^{(i)})$$

with the base case $c^{(0)} \leftarrow 0$. However, carries do not always propagate through the addition. In fact, a pair $u^{(i)} = 0$ and $v^{(i)} = 0$ always kills the propagating carry. Similarly, the carry bit is propagated further only if one of the digits is 1. Therefore, we can significantly speed up the carry computations if we split the number into smaller chunks and use flags $p^{(k)}, \dots, p^{(0)}$ that indicate whether the next carry update can propagate to the carry $c^{(i)}$.

In a nutshell, the carry look-ahead algorithm computes set and propagate flags by recursively merging adjacent blocks of bits until we get a single block. Let $c^{(p)}, \dots, c^{(q)}$ and $c^{(q-1)}, \dots, c^{(r)}$ denote carry bits set so far. Let $p^{(i+q)}$ denote that the carry propagates from the beginning of the block $c^{(p)}, \dots, c^{(q)}$ to the ith position, i.e., $u^{(q+i-1)} \dots u^{(q)} + v^{(q+i-1)} \dots v^{(q)} = 2^i - 1$. Then we must set

$$s^{(i+q)} \leftarrow s^{(i+q)} + p^{(i+q)} s^{(q-1)} \qquad \text{and} \qquad p^{(i+q)} \leftarrow p^{(i+q)} p^{(q-1)}$$

to merge the corresponding blocks. In the base case, blocks are one bit wide and we must set

$$s^{(i)} \leftarrow u^{(i)} v^{(i)} \qquad \text{and} \qquad p^{(i)} \leftarrow u^{(i)} + v^{(i)} - 2s^{(i)} \enspace .$$

If the carry bits are known then computing the corresponding bit representation is easy:

$$w^{(i)} \leftarrow u^{(i)} + v^{(i)} - 2s^{(i)} + s^{(i-1)} \enspace .$$

The corresponding share computing algorithm is depicted below.

```
Algorithm 1: Protocol for the bitwise addition of two vectors
Data: Shared bit vectors $[\![u^{(n-1)}]\!], \dots, [\![u^{(0)}]\!]$ and $[\![u^{(n-1)}]\!], \dots, [\![u^{(0)}]\!]$.
Result: Shared bit vector $[\![w^{(n-1)}]\!], \dots, [\![w^{(0)}]\!]$ such that $w = u + v$.
Round 1
  for $j \leftarrow 0$ to $n - 1$ do
    Compute shares $[\![p^{(j)}]\!] \leftarrow [\![u^{(j)}]\!] + [\![v^{(j)}]\!]$.
    Compute shares $[\![s^{(j)}]\!] \leftarrow [\![u^{(j)}]\!] \cdot [\![v^{(j)}]\!]$.
Rounds 2...$\log_2 n + 1$
  for $k \leftarrow 0$ to $\log_2 n - 1$ do
    for $\ell \leftarrow 0$ to $2^k - 1$ do
      for $m \leftarrow 0$ to $\frac{n}{2^k+1} - 1$ do
        Compute shares
        $[\![s^{(2^k+\ell+2^{k+1}m)}]\!] \leftarrow [\![s^{(2^k+\ell+2^{k+1}m)}]\!] + [\![p^{(2^k+\ell+2^{k+1}m)}]\!] \cdot [\![s^{(2^k+2^{k+1}m-1)}]\!]$.
        Compute shares $[\![p^{(2^k+\ell+2^{k+1}m)}]\!] \leftarrow [\![p^{(2^k+\ell+2^{k+1}m)}]\!] \cdot [\![p^{(2^k+2^{k+1}m-1)}]\!]$.
  Compute shares $[\![w^{(0)}]\!] \leftarrow [\![u^{(0)}]\!] + [\![v^{(0)}]\!] - 2 \cdot [\![s^{(0)}]\!]$.
  for $j \leftarrow 1$ to $n - 1$ do
    Compute shares $[\![w^{(j)}]\!] \leftarrow [\![u^{(j)}]\!] + [\![v^{(j)}]\!] + [\![s^{(j-1)}]\!] - 2 \cdot [\![s^{(j)}]\!]$.
```

B Share multiplication protocol

Algorithm 2: Protocol for multiplying two shared values

```
Data: Shared values $[\![u]\!]$ and $[\![v]\!]$.
Result: Shared value $[\![w]\!]$ such that $w = uv$.
Round 1
  $\mathcal{P}_1$ generates $r_{12}, r_{13}, s_{12}, s_{13}, t_{12} \leftarrow \mathbb{Z}_{2^{32}}$.
  $\mathcal{P}_2$ generates $r_{23}, r_{21}, s_{23}, s_{21}, t_{23} \leftarrow \mathbb{Z}_{2^{32}}$.
  $\mathcal{P}_3$ generates $r_{31}, r_{32}, s_{31}, s_{32}, t_{31} \leftarrow \mathbb{Z}_{2^{32}}$.
  All values $*_{ij}$ are sent from $\mathcal{P}_i$ to $\mathcal{P}_j$.
Round 2
  $\mathcal{P}_1$ computes $\hat{a}_{12} \leftarrow u_1 + r_{31}$, $\hat{b}_{12} \leftarrow v_1 + s_{31}$, $\hat{a}_{13} \leftarrow u_1 + r_{21}$,
  $\hat{b}_{13} \leftarrow v_1 + s_{21}$.
  $\mathcal{P}_2$ computes $\hat{a}_{23} \leftarrow u_2 + r_{12}$, $\hat{b}_{23} \leftarrow v_2 + s_{12}$, $\hat{a}_{21} \leftarrow u_2 + r_{32}$,
  $\hat{b}_{21} \leftarrow v_2 + s_{32}$.
  $\mathcal{P}_3$ computes $\hat{a}_{31} \leftarrow u_3 + r_{23}$, $\hat{b}_{31} \leftarrow v_3 + s_{23}$, $\hat{a}_{32} \leftarrow u_3 + r_{13}$,
  $\hat{b}_{32} \leftarrow v_3 + s_{13}$.
  All values $*_{ij}$ are sent from $\mathcal{P}_i$ to $\mathcal{P}_j$.
Round 3
  $\mathcal{P}_1$ computes:
    $c_1 \leftarrow$
    $u_1\hat{b}_{21}+u_1\hat{b}_{31}+v_1\hat{a}_{21}+v_1\hat{a}_{31}-\hat{a}_{12}\hat{b}_{21}-\hat{b}_{12}\hat{a}_{21}+r_{12}s_{13}+s_{12}r_{13}-t_{12}+t_{31}$,
    $w_1 \leftarrow c_1 + u_1v_1$.
  $\mathcal{P}_2$ computes:
    $c_2 \leftarrow$
    $u_2\hat{b}_{32}+u_2\hat{b}_{12}+v_2\hat{a}_{32}+v_2\hat{a}_{12}-\hat{a}_{23}\hat{b}_{32}-\hat{b}_{23}\hat{a}_{32}+r_{23}s_{21}+s_{23}r_{21}-t_{23}+t_{12}$,
    $w_2 \leftarrow c_2 + u_2v_2$.
  $\mathcal{P}_3$ computes:
    $c_3 \leftarrow$
    $u_3\hat{b}_{13}+u_3\hat{b}_{23}+v_3\hat{a}_{13}+v_3\hat{a}_{23}-\hat{a}_{31}\hat{b}_{13}-\hat{b}_{31}\hat{a}_{13}+r_{31}s_{32}+s_{31}r_{32}-t_{31}+t_{23}$,
    $w_3 \leftarrow c_3 + u_3v_3$.
```

C Share conversion protocol

```
Algorithm 3: Protocol for converting shares from $\mathbb{Z}_2$ to $\mathbb{Z}_{2^{32}}$
  Data: Bit shares $\llbracket u \rrbracket$ of a bit $u$ over $\mathbb{Z}_2$.
  Result: Shares $\llbracket w \rrbracket$ of $u$ over $\mathbb{Z}_{2^{32}}$.
  Round 1
    $\mathcal{P}_1$ generates $r_{12}, r_{13}, s_{12}, s_{13}, t_{12} \leftarrow \mathbb{Z}_{2^{32}}$.
    $\mathcal{P}_1$ computes $s_1 \leftarrow r_{12}r_{13} - s_{12} - s_{13}$.
    $\mathcal{P}_2$ generates $r_{23}, r_{21}, s_{23}, s_{21}, t_{23} \leftarrow \mathbb{Z}_{2^{32}}$.
    $\mathcal{P}_2$ computes $s_2 \leftarrow r_{23}r_{21} - s_{23} - s_{21}$.
    $\mathcal{P}_3$ generates $r_{31}, r_{32}, s_{31}, s_{32}, t_{31} \leftarrow \mathbb{Z}_{2^{32}}$.
    $\mathcal{P}_3$ computes $s_3 \leftarrow r_{31}r_{32} - s_{31} - s_{32}$.
    All values $*_{ij}$ are sent from $\mathcal{P}_i$ to $\mathcal{P}_j$.
  Round 2
    $\mathcal{P}_1$ computes $\hat{b}_{12} \leftarrow r_{31} + u_1$, $\hat{b}_{13} \leftarrow r_{21} + u_1$.
    $\mathcal{P}_2$ computes $\hat{b}_{23} \leftarrow r_{12} + u_2$, $\hat{b}_{21} \leftarrow r_{32} + u_2$.
    $\mathcal{P}_3$ computes $\hat{b}_{31} \leftarrow r_{23} + u_3$, $\hat{b}_{32} \leftarrow r_{13} + u_3$.
    Additionally, $\mathcal{P}_3$ shares $u_3$: $c_{31}, c_{32} \leftarrow \mathbb{Z}_{2^{32}}$, $c_{33} \leftarrow u_3 - c_{31} - c_{32}$.
    All values $*_{ij}$, $i \neq j$ are sent from $\mathcal{P}_i$ to $\mathcal{P}_j$.
  Round 3
    $\mathcal{P}_1$ computes $ab_1 \leftarrow s_{31} - r_{31}\hat{b}_{21}$, $ac_1 \leftarrow \hat{b}_{31}\hat{b}_{13} + s_{21} - \hat{b}_{31}r_{21}$, $bc_1 \leftarrow s_1$,
    $c_1 \leftarrow c_{31}$.
    $\mathcal{P}_2$ computes $ab_2 \leftarrow \hat{b}_{12}\hat{b}_{21} + s_{32} - \hat{b}_{12}r_{32}$, $ac_2 \leftarrow s_2$, $bc_2 \leftarrow s_{12} - r_{12}\hat{b}_{32}$,
    $c_2 \leftarrow c_{32}$.
    $\mathcal{P}_3$ computes $ab_3 \leftarrow s_3$, $ac_3 \leftarrow s_{23} - r_{23}\hat{b}_{13}$, $bc_3 \leftarrow \hat{b}_{23}\hat{b}_{32} + s_{13} - \hat{b}_{23}r_{13}$,
    $c_3 \leftarrow c_{33}$.
    Multiply shares of $\llbracket ab \rrbracket$ and $\llbracket c \rrbracket$ to get $\llbracket abc \rrbracket$.
  Round 4
    $\mathcal{P}_1$ computes $w_1 \leftarrow u_1 - 2ab_1 - 2bc_1 - 2ac_1 + 4abc_1 - t_{12} + t_{31}$
    $\mathcal{P}_2$ computes $w_2 \leftarrow u_2 - 2ab_2 - 2bc_2 - 2ac_2 + 4abc_2 - t_{23} + t_{12}$
    $\mathcal{P}_3$ computes $w_3 \leftarrow u_3 - 2ab_3 - 2bc_3 - 2ac_3 + 4abc_3 - t_{31} + t_{23}$
```

D Bit extraction protocol

Algorithm 4: Protocol for extracting bit shares from a shared value

```
Data: Shared value $\llbracket u \rrbracket$.
Result: Shares $\llbracket u^{(31)} \rrbracket, \ldots, \llbracket u^{(0)} \rrbracket$ of all bits of $u$.
Round 1
  $\mathcal{P}_1$ generates $\overline{r}_1^{(0)}, \ldots, \overline{r}_1^{(31)} \leftarrow \mathbb{Z}_2$.
  $\mathcal{P}_2$ generates $\overline{r}_2^{(0)}, \ldots, \overline{r}_2^{(31)} \leftarrow \mathbb{Z}_2$.
  $\mathcal{P}_3$ generates $\overline{r}_3^{(0)}, \ldots, \overline{r}_3^{(31)} \leftarrow \mathbb{Z}_2$.
  Convert shares $\llbracket \overline{r}^{(i)} \rrbracket$ over $\mathbb{Z}_2$ to shares $\llbracket r^{(i)} \rrbracket$ over $\mathbb{Z}_{2^{32}}$.
Round 2
  Compute shares $\llbracket r \rrbracket \leftarrow 2^{31} \cdot \llbracket r^{(31)} \rrbracket + 2^{30} \cdot \llbracket r^{(30)} \rrbracket + \cdots + 2^{0} \cdot \llbracket r^{(0)} \rrbracket$.
  Compute shares $\llbracket v \rrbracket \leftarrow \llbracket u \rrbracket - \llbracket r \rrbracket$ and broadcast shares $v_1, v_2, v_3$.
Round 3
  $\mathcal{P}_1$ computes $a_1 \leftarrow v_1 + v_2 + v_3$ and finds its bits $a_1^{(0)}, \ldots, a_1^{(31)}$.
  $\mathcal{P}_2$ initialises $a_2 \leftarrow 0$ and sets $a_2^{(0)}, \ldots, a_2^{(31)} \leftarrow 0$.
  $\mathcal{P}_3$ initialises $a_3 \leftarrow 0$ and sets $a_3^{(0)}, \ldots, a_3^{(31)} \leftarrow 0$.
  Add $\llbracket a^{(31)} \rrbracket, \ldots, \llbracket a^{(0)} \rrbracket$ and $\llbracket r^{(31)} \rrbracket, \ldots, \llbracket r^{(0)} \rrbracket$ bitwise to compute
  $\llbracket u^{(31)} \rrbracket, \ldots, \llbracket u^{(0)} \rrbracket$.
```

E Greater-than-or-equal comparison protocol

Algorithm 5: Protocol for evaluating the greater-than predicate.

```
Data: Shared values $\llbracket u \rrbracket$ and $\llbracket v \rrbracket$.
Result: Shares $\llbracket w \rrbracket$ such that $w = \textsc{Gte}(u, v)$.
All parties $\mathcal{P}_i$ compute $d_i \leftarrow u_i - v_i$.
All parties extract bits $\llbracket d^{(0)} \rrbracket, \ldots, \llbracket d^{(31)} \rrbracket$ of $\llbracket d \rrbracket$.
All parties $\mathcal{P}_i$ learn the result share $\llbracket w \rrbracket \leftarrow 1 - \llbracket d^{(31)} \rrbracket$.
```