Sharemind: a framework for fast privacy-preserving computations

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

[1] University of Tartu, Liivi 2, 50409 Tartu, Estonia.
{db,swen,jan}@ut.ee
[2] Cybernetica AS, Akadeemia tee 21, 12618 Tallinn, Estonia

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

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 use 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. Privacy-preserving data mining techniques try to address such problems. So far the focus has been on randomised response techniques [2,1,13]. 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 alternative is to consider this problem as a multi-party computation task, where the data donors want to securely aggregate data without revealing their private

⋆ This research has been supported by Estonian Science Foundation grant number 7081.

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 to external corruption. Consequently, we can assume that only a few miner parties can be corrupted during the computation. Thus, 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 and thus all important results are well established by now, see [3,7] for further references. 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. 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 [17].

We have made some non-standard choices to assure maximal efficiency. First, the SHAREMIND framework uses 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.

Second, 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. Also, it is possible to extend the framework for other settings. For example, one can implement generic methodology given in [9].

To make the presentation more fluent, we describe the SHAREMIND framework step by step through Sect. 2–5. Performance results are presented and analysed in Sect. 6. In particular, we compare our results with other implementations of privacy-preserving computations [16,6,18]. Finally, we conclude our presentation with some improvement plans for future, see Sect. 7.

Some of the details of this work have been omitted because of space limitations. The full version of this article that covers all these details can be found on the homepage of SHAREMIND project [17] and in the IACR ePrint Archive [5].

2 Cryptographic Preliminaries

Theoretical attack model. In this article, we state and prove all security guarantees in the *information-theoretical setting*, where each pair of participants 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 [4,14] for further details.

Secondly, 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, this setting becomes equivalent to the standard synchronised network model with a rushing adversary.

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.

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 him- or herself. All security proofs in this article are suitable for both security models, since they assure that $\tau_\mathrm{id} \leq c \cdot \tau_\mathrm{re}$ where c is a 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. Further details and standard results can be found in the manuscripts [3,11,7,8].

Universal composability. Complex protocols are often designed by combining several low level protocols. Unfortunately, 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. We refer to the manuscript [8] for a more formal and precise treatment.

Secret sharing schemes. Secret sharing schemes are used to securely distribute private values to a group of participants. More precisely, let $\mathcal{M}$ be the set of possible secrets and let $\mathcal{S}_1, \ldots, \mathcal{S}_n$ be the sets of possible shares. Then shares for the participants are created with a randomised sharing algorithm $\mathsf{Deal} : \mathcal{M} \to \mathcal{S}_1 \times \ldots \times \mathcal{S}_n$. Participants can use a recovery algorithm $\mathsf{Rec} : \mathcal{S}_1 \times \ldots \times \mathcal{S}_n \to \mathcal{M} \cup \{\perp\}$ to restore the secret form shares. For brevity, we use a shorthand $[\![s]\!]$ to denote the shares $[s_1, \ldots, s_n]$ generated by the sharing algorithm $\mathsf{Deal}(s)$.

Secret sharing schemes can have different security properties depending on the exact details of Deal and Rec algorithms. The SHAREMIND framework uses *additive sharing* over $\mathbb{Z}_{2^{32}}$, where a secret value s is split to shares $s_1, \ldots, s_n \in \mathbb{Z}_{2^{32}}$ 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. As a result, participants cannot learn anything about s unless all of them join their shares.

3 Privacy-Preserving Data Aggregation

As already emphasised in the introduction, organisations who collect and process 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. Although proper legislation and auditing reduces the corresponding risks, data donors must often unconditionally trust institutions that gather and process data. In the following, we show how to use cryptographic techniques to avoid such unconditional trust.

The SHAREMIND framework for privacy-preserving computations uses secret sharing to split confidential information between several nodes (*miners*). By sending the shares of the data to the miners, data donors effectively delegate all rights over the data to the consortium of miners. Let t be the prescribed corruption threshold such that no information can be learnt about the inputs if the number of collaborating corrupted parties is below t. We allow some miner nodes to be corrupted, but require that the total number of corrupted nodes is below the threshold t. The latter can be achieved with physical and organisational security measures such as dedicated server rooms and software auditing. This is achievable, since the framework needs only a few miner nodes. In practice, each miner node should be hosted by a separate respected organisation.

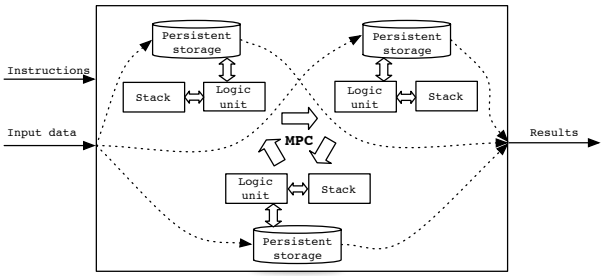

Fig. 1. In SHAREMIND, input data and instructions are sent to miner nodes that use multi-party computation to execute the algorithm. The result is returned when the computation is complete.

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 $\mathbb{Z}_{2^{32}}$. The framework provides efficient protocols for basic mathematical operations so that one 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 perfectly universally composable.

The current version of SHAREMIND framework is based on three miner nodes and tolerates semi-honest corruption of a single node, i.e., no information is leaked unless two miner nodes collaborate. The latter is a compromise between efficiency and security. Although a larger number of miner nodes increases the level of tolerable corruption, it also makes assuring semi-honest behaviour much more difficult. Secondly, the communication complexity of multi-party computation protocols is roughly quadratic in the number of miners n and thus three is the optimal choice. Besides, it is difficult to find more than a handful of independent organisations that can provide adequate protection measures and are not motivated to collaborate with each other.

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 fields. As a result, it is easy to implement secure addition and multiplication modulo prime p or in the Galois field $\mathbb{F}_{2^k}$. However, the integer arithmetic in modern computers is done modulo 2^{32}. Consequently, the most space- and time-efficient solution is to use additive secret sharing 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.

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 fixed 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. This significantly reduces the number of rounds required for applying similar operations on many inputs.

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 many 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 (see [15]) 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. How to build such a network form standard cryptographic primitives is detailed in Sect. 5. Also, note that the next section provides only a general description of all protocols, detailed technical description of all protocols can be found in the full version of this article [5].

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 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.

Although share computing protocols are often used as elementary steps in more complex protocols, they themselves can be composed from even smaller atomic oper-

1. Each party $\mathcal{P}_i$ sends a random mask $r_i \leftarrow \mathbb{Z}_{2^{32}}$ to the right neighbour $\mathcal{P}_{i+1}$.
2. Each party $\mathcal{P}_i$ uses the input share u_i to compute the output $w_i \leftarrow u_i + r_{i-1} - r_i$.

Fig. 2. Re-sharing protocol for three parties.

ations. 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. *If all sub-protocols of a protocol are perfectly simulatable, then the protocol is 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 $\mathcal{T}$ who recovers the input $u \leftarrow \mathsf{Rec}([\![u]\!])$ and sends new shares $[\![w]\!] \leftarrow \mathsf{Deal}(u)$ back to the miners. The simplest universally composable re-sharing protocol is given in Fig. 2. Indeed, we can construct a non-rewinding *interface* $\mathcal{I}_0$ between the ideal world and a real world adversary $\mathcal{A}$ such that for any input distribution the output distributions ψ and ψ° coincide. The corresponding interface $\mathcal{I}_0$ forwards the input share u_i of a corrupted miner $\mathcal{P}_i$ to $\mathcal{T}$, 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 $\mathcal{P}_i$.

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 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

Due to the properties of additive sharing, we can implement share addition and multiplication by a public constant c with local operations only, as $[u_1 + v_1, \ldots, u_n + v_n]$ and $[cu_1, \ldots, cu_n]$ are valid shares of $u + v$ and cu. However, 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. Shamir secret sharing just 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, $\mathcal{P}_2$ 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 [12] although similar reduction techniques have been used earlier. Fig. 3 depicts the corresponding protocol. Essentially, the correctness of the protocol relies on the observation

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

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.

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. 3. Du-Atallah multiplication protocol.

Execute the following protocols concurrently:

1. Compute locally shares u_1v_1, u_2v_2 and u_3v_3.
2. Use six instances of the Du-Atallah protocol for computing shares of u_iv_j where $i \neq j$.
3. Re-share the final sum of all previous sub-output shares.

Fig. 4. High-level description of the share multiplication protocol.

Lemma 3. *The Du-Atallah protocol depicted in Fig. 3 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. □

Fig. 4 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.

Theorem 1. *The multiplication protocol is perfectly universally composable.*

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

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:

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

1. Generate random bit shares $[\![r^{(31)}]\!], \ldots, [\![r^{(0)}]\!]$ over $\mathbb{Z}_{2^{32}}$.
2. Compute the corresponding shares $[\![r]\!] = 2^{31} \cdot [\![r^{(31)}]\!] + \cdots + 2^0 \cdot [\![r^{(0)}]\!]$.
3. Compute and publish the shares of the difference $[\![a]\!] = [\![u]\!] - [\![r]\!]$.
4. Mimic bitwise addition algorithm to compute bit shares $[\![u^{(31)}]\!], \ldots, [\![u^{(0)}]\!]$ from the known bit representation of a and the bit shares $[\![r^{(31)}]\!], \ldots, [\![r^{(0)}]\!]$.

Fig. 5. High-level description of the bit extraction protocol.

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.

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. □

Note that input gathering can even be an off-line event, if we make use of public-key encryption. If everybody knows the public keys of the miners, they can encrypt the shares with the corresponding keys and then store the encryptions in a public database. Miners can later fetch and decrypt their individual shares to fill their input databases.

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 [10] 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, see Fig. 5.

For the first step in the algorithm, miners can create random shares over $\mathbb{Z}_2$ and then convert them to the shares over $\mathbb{Z}_{2^{32}}$. The second step can be computed locally. The third step is secure, since the difference $a = u - r$ has uniform distribution over $\mathbb{Z}_{2^{32}}$ and thus one can always simulate the shares of a. For the final step, note that addition and multiplication protocols are sufficient to implement all logic gates when all inputs are guaranteed to be in the range $\{0, 1\}$. Hence, we can use the classical bitwise addition algorithm to compute $[\![u^{(31)}]\!], \ldots, [\![u^{(0)}]\!]$. However, the number of rounds in the corresponding protocol is linear in the number of bits, since we cannot compute carry bits locally. To minimise the number of rounds, we used standard look-ahead carry construction to perform the carry computations in parallel. The latter provides

logarithmic round complexity. More precisely, the final bitwise addition protocol has 8 rounds and the corresponding bit extraction protocol has 12 rounds. Both protocols are also universally composable, since all sub-protocols are universally composable.

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

As a simple extension, we describe how to implement greater-than-or-equal predicate if both arguments are guaranteed to be in $\mathbb{Z}_{2^{31}} \subseteq \mathbb{Z}_{2^{32}}$. This allows us to define

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

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 for the full range $\mathbb{Z}_{2^{32}} \times \mathbb{Z}_{2^{32}}$, 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. To hide the execution path of the algorithm, we can replace if-then branches with oblivious selection clauses. For instance, we can represent **if** a **then** $x \leftarrow y$ **else** $x \leftarrow z$ as $x \leftarrow a \cdot y + (1 - a) \cdot z$.

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. Secure channels between the miners are implemented using standard symmetric encryption and authentication algorithms. 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 prohibitively expensive.

One of the biggest advances of the framework is its modularity. At the highest abstraction level, the framework behaves as a virtual processor with a fixed set of commands. However, the user can design and experiment with new cryptographic protocols. On this level, the framework hides all technical details, such as network setup and exact

details of message delivery. Finally, the user can explicitly change networking details at the lowest level, although we have put a lot of effort into optimising network behaviour.

To facilitate fast testing and algorithm development, we implemented the most obvious execution strategy, where the controller application executes a program by asking the miners to sequentially execute operations described by the program. 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. The shares of the final results are sent back to the controller.

Of course, such a simplistic approach neglects many practical security concerns. In particular, the controller has full control over the miners and thus we have a single point of failure. Therefore, real-world applications must be accompanied with auxiliary mechanisms to avoid such high level attacks. For instance, the miners must be configured with the identities of each other and all possible controllers to avoid unauthorised commands. This can be achieved by using public-key infrastructure. Similarly, the complete code should be analysed and signed by an appropriate authority to avoid unauthorised data manipulation. However, the time-complexity of these operations is constant and thus our execution strategy is still valid for performance testing.

6 Performance Results

We have measured the performance of the SHAREMIND framework on two computational tasks—scalar product and vectorised comparison. These tests are chosen to cover the most important primitives of SHAREMIND: addition, multiplication and comparison. More importantly, it also allowed us to compare SHAREMIND to other secure multi-party computation systems [16,6,18].

The input datasets were randomly generated and the corresponding shares were stored in local databases. For each vector size, we ran the computation many times and measured the results for each execution. To identify performance bottlenecks, we measured the local computation time, the time spent on sending data, and the time spent on waiting. The time was measured at the miners to minimise the impact of overhead from communication with the controller. The tests were performed on four computers in a computing cluster. Each machine had 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 roundtrip time has significant impact on the performance. Consequently, it is advantageous to execute many operations in parallel and thus the use of vectorised operations can lead to significant performance gains. The latter is a promising result, since many data mining algorithms are based on highly parallelisable matrix operations.

Nevertheless, we also observed that sometimes data vectors become too large and this starts to hinder the performance of the networking layer. To balance the effects of vectorisation, we implemented a load balancing system. We fixed a certain threshold vector size after which the miners start batch processing of large vectorised queries. In

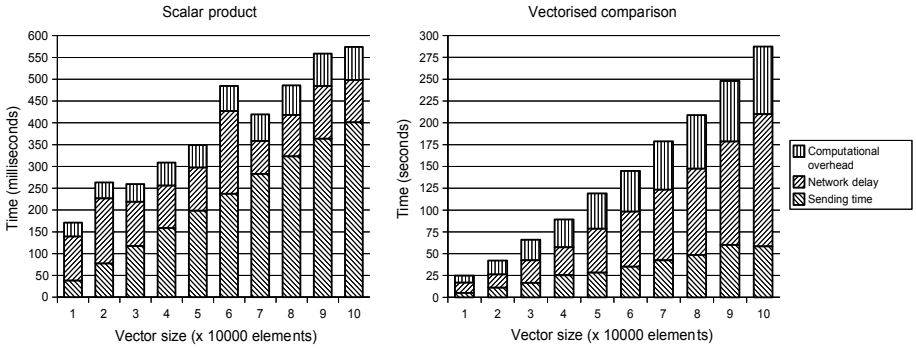

Fig. 6. Performance of the SHAREMIND framework. Left and right pane depict average running times for test vectors with $10,000$–$100,000$ elements in $10,000$-element increments.

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.

Fig. 6 shows the impact of our optimisations on the waiting time caused by network delays. In particular, note that the impact of network delays is small during scalar product computation—the miners do not waste too many CPU cycles while waiting for inputs. Consequently, further optimisations can only lead to marginal improvements. The same is true for the multiplication protocol, since the performance characteristics of the scalar product operation practically coincide with the multiplication protocol: addition as a local operation is very fast. For the parallel comparison, the effect of network delays 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 layer can significantly influence performance results.

Besides measuring the average running time, we also considered variability of timings. For the comparison protocol, the running times were rather stable. The average standard deviation was approximately 6% from the average running time. The scalar computation execution time was significantly more fluctuating, 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 small, so even 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 practice, the effect of network delays and the variability of running times can be considerably larger.

We also compared the performance of SHAREMIND with other known implementations of privacy-preserving computations. Our first candidate was the FAIRPLAY system [16], which is a general framework for secure function evaluation with two parties that is based on garbled circuit evaluation. According to the authors, a single comparison operation for 32-bit integers takes 1.25 seconds. A single SHAREMIND compari-

son 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 [18] 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 the 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 [6] 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.

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 standard finite fields.

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 without worrying about the details of cryptographic protocols.

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

1. Dakshi Agrawal and Charu C. Aggarwal. On the design and quantification of privacy preserving data mining algorithms. In *Proc. of PODS '01*, pages 247–255, 2001.
2. Rakesh Agrawal and Ramakrishnan Srikant. Privacy-preserving data mining. *SIGMOD Rec.*, 29(2):439–450, 2000.
3. Donald Beaver. Secure multiparty protocols and zero-knowledge proof systems tolerating a faulty minority. *Journal of Cryptology*, 4(2):75–122, 1991.
4. Michael Ben-Or, Boaz Kelmer, and Tal Rabin. Asynchronous secure computations with optimal resilience (extended abstract). In *Proc. of PODC '94*, pages 183–192, 1994.
5. Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: a framework for fast privacy-preserving computations. Cryptology ePrint Archive, Report 2008/289, 2008.
6. 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 *Proc. of Financial Cryptography*, LNCS 4107, 2006.
7. Ran Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 13(1):143–202, 2000.
8. Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Proc. of FOCS '01*, pages 136–145, 2001.
9. Ronald Cramer, Serge Fehr, Yuval Ishai, and Eyal Kushilevitz. Efficient multi-party computation over rings. In *Proc. of EUROCRYPT '03*, LNCS 4107, pages 596–613, 2003.
10. 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 *Proc. of TCC 2006*, LNCS 3876, pages 285–304, 2006.
11. Yevgeniy Dodis and Silvio Micali. Parallel reducibility for information-theoretically secure computation. In *Proc. of CRYPTO '00*, LNCS 1880, pages 74–92, 2000.
12. Wenliang Du and Mikhail J. Atallah. Protocols for secure remote database access with approximate matching. ACMCCS 2000, Nov. 1-4, 2000. Athens, Greece.
13. Alexandre V. Evfimievski, Ramakrishnan Srikant, Rakesh Agrawal, and Johannes Gehrke. Privacy preserving mining of association rules. In *Proc. of KDD '02*, pages 217–228, 2002.
14. Martin Hirt and Ueli M. Maurer. Player simulation and general adversary structures in perfect multiparty computation. *Journal of Cryptology*, 13(1):31–60, 2000.
15. Yehuda Lindell and Benny Pinkas. A proof of Yao's protocol for secure two-party computation. Cryptology ePrint Archive, Report 2004/175, 2004.
16. Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay - secure two-party computation system. In *Proc. of USENIX Security Symposium*, pages 287–302, 2004.
17. The SHAREMIND project web page. `http://sharemind.cs.ut.ee`, 2007.
18. Zhiqiang Yang, Rebecca N. Wright, and Hiranmayee Subramaniam. Experimental analysis of a privacy-preserving scalar product protocol. *Comput. Syst. Sci. Eng.*, 21(1), 2006.