CYBERNETICA
Institute of Information Security

The design and implementation of a two-party protocol suite for SHAREMIND 3

Pille Pullonen, Dan Bogdanov, Thomas Schneider

T-4-17 / 2012

Pille Pullonen[1,2], Dan Bogdanov[1,2], Thomas Schneider[3].
[1] Cybernetica, Institute of Information Security,
[2] University of Tartu, Institute of Computer Science,
[3] European Center for Security and Privacy by Design (EC SPRIDE)

The research reported here was supported by:

1. the European Regional Development Fund through the Estonian Center of Excellence in Computer Science, EXCS

2. Estonian Science foundation, grant(s) No. 8124

Cybernetica research reports are available online at
`http://research.cyber.ee/`

Mailing address:
Cybernetica AS
Mäealuse 2
12618 Tallinn
Estonia

The design and implementation of a two-party protocol suite for SHAREMIND 3

Pille Pullonen, Dan Bogdanov, Thomas Schneider

August 30, 2012

Abstract

This report introduces the basics of two-party secret sharing protocols based on additive secret sharing implemented in SHAREMIND. The main contribution is the multiplication protocol which uses Paillier's additively homomorphic cryptosystem. One approach is to directly use this protocol for multiplication and the other is to generate Beaver's triples and use them for the multiplication. The benchmarking results show that the protocol that directly uses Paillier's cryptosystem is slow compared to the native SHAREMIND protocol, but would be feasible as a precomputation phase. The online protocol using Beaver's triples outperforms the three-party SHAREMIND multiplication.

1 Introduction

SHAREMIND [BLW08] is a framework for secure multi-party computation using additive secret sharing. The traditional protection domain model consists of three miner servers and a client or a controller application. The controller is responsible for sharing the inputs and scheduling computations on these inputs as well as receiving the results. The servers and the controller communicate with each other over encrypted network channels and all computation protocols are designed to protect the privacy of inputs. One of the currently active research areas for SHAREMIND 3 is developing protocol suites for settings other than additively shared secrets and three miners. This work describes a protection domain, that is similar to the traditional one, but uses only two miners. Some of the three-party protocols are easily usable with two parties and the main contribution of this work is the development of the multiplication protocol that uses an additively homomorphic cryptosystem. We also explore the possibility to use the proposed protocol as a precomputation phase to generate Beaver's triples and use these triples in the multiplication protocol.

This report is structured as follows. Section 2 introduces the concepts necessary for understanding the developed protocols. More precisely, it covers Paillier's cryptosystem as an example of an additively homomorphic cryptosystem, basics of the additive secret sharing scheme and the main ideas for proving the security of the SHAREMIND protocols. The following Section 3 describes the main protocols of the two party system and provides argumentation for their security. Section 4 explains the implementation details and choices, concentrating on the multiplication protocols and Paillier's cryptosystem. Finally, Section 5 gives the performance results of the implemented protocols and compares them to the native SHAREMIND protocols. Section 6 concludes this report.

2 Preliminaries

2.1 Paillier's cryptosystem

Paillier's public-key cryptosystem [Pai99] uses an RSA modulus $N = pq$ where the secret components p and q are large primes with equal bit length. In fact, it must hold that $gcd(pq, (p-1)(q-1)) = 1$, which is easily satisfied if p and q are of equal bit length. The public key is (N, g), where $g \in \mathbb{Z}^*_{N^2}$ and the private key is $\lambda = lcm(p-1, q-1)$. The encryption function $E(m, r)$, where $m \in \mathbb{Z}_N$, requires a randomness $r \in \mathbb{Z}^*_N$ and defines the ciphertext as $c = E(m, r) = g^m r^N \bmod N^2$. The decryption function $D(c) = \frac{L(c^\lambda \bmod N^2)}{L(g^\lambda \bmod N^2)} \bmod N$ uses a helper function $L(x) = \frac{x-1}{N}$ that is evaluated as an integer division.

Paillier's cryptosystem is additively homomorphic, allowing to compute the sum of the messages under encryption $E(m_1 + m_2, r_1 \cdot r_2) = E(m_1, r_1) \cdot E(m_2, r_2)$. This property also allows to evaluate the multiplication of an encrypted message and a plain value k under encryption $E(km, r^k) = E(m, r)^k$.

The modulus N must be difficult to factor and for medium term security the length of N should be at least 2048 bits, as can be seen from various recommendations at [Gir]. Paillier's cryptosystem is indistinguishable under chosen plaintext attacks (IND-CPA) under the Decisional Composite Residuosity Assumption. We say that the Paillier's cryptosystem is (t, ε)-indistinguishable if for two known messages m_0 and m_1 the probability of distinguishing between the encryptions of these messages

$$Pr[IND] = |Pr[A(E(m_0)) = 0] - Pr[A(E(m_1)) = 0]| \leq \varepsilon$$

for any t-time adversary A.

There are several known improvements to the basic definition of Paillier's cryptosystem, for example in [Pai99] and [DJ01]. The decryption can be simplified by precomputing the constant $L(g^\lambda \bmod N^2)^{-1} \bmod N$ and reducing the complexity of the helper function $L(x)$ by defining $L(x) = (x-1) \cdot u \bmod 2^{|N|}$, where $|N|$ denotes the bit length of N and $u = N^{-1} \bmod 2^{|N|}$ can be precomputed. Furthermore, it is possible to use the Chinese Remainder Theorem (CRT) to reduce the decryption workload by computing separately modulo p and q and combining the results. Using the CRT requires the definitions of two helper functions $L_p(x) = \frac{x-1}{p}$ and $L_q(x) = \frac{x-1}{q}$ that can be simplified as the original $L(x)$, resulting in $L_p(x) = \frac{x-1}{p} = (x-1) \cdot u_p \bmod 2^{|p|}$, where $u_p = p^{-1} \bmod 2^{|p|}$. The corresponding decryption constants can also be precomputed when using CRT, giving $h_p = L_p(g^{p-1} \bmod p^2)^{-1} \bmod p$ and $h_q = L_q(g^{q-1} \bmod q^2)^{-1} \bmod q$. Finally, values $m_p = L_p(c^{p-1} \bmod p^2) \cdot h_p \bmod p$ and $m_q = L_q(c^{q-1} \bmod q^2) \cdot h_q \bmod q$ are computed to decrypt the ciphertext c and restore the message as $m = CRT(m_p, m_q) \bmod N$.

The part g of the public key can always be a constant value without lessening the security of the system, a small constant could be used to simplify the computation. Alternatively, Damgård and Jurik [DJ01] propose that g could be always defined as $g = N+1$ which allows for faster encryption $E(m, r) = (N+1)^m r^N \bmod N^2 = (Nm+1)r^N \bmod N^2$. If the randomness r^N is precomputed before encryption, then this choice of parameters can reduce the encryption complexity from two modular exponentiations to two multiplications. In addition, it is possible to define a second encryption function that uses the private key and the CRT to at first compute $c_p = (Nm+1)r^N \bmod p^2$ and $c_q = (Nm+1)r^N \bmod q^2$ and then combine them to obtain $c = CRT(c_p, c_q) \bmod N^2$.

2.2 Additive secret sharing

By the definition of the additive secret sharing scheme a secret a is represented by several shares a_i that satisfy the equality

$$a = \sum_{i=1}^{n} a_i.$$

For n participants the scheme is a (n, n) threshold scheme meaning that a secret is split into n shares and all n shares are needed to restore the secret. Restoring is done according to the previous equation by summing all the shares. A sharing algorithm can be easily obtained by fixing a_1 to a_{n-1} at random and computing

$$a_n = a - \sum_{i=1}^{n-1} a_i.$$

Additive secret sharing is information theoretically secure meaning that any set of less than n shares reveals nothing about the secret value.

Additive secret sharing is in general defined over a ring, but SHAREMIND only considers the cases of $\mathbb{Z}_{2^k}$, mostly using $\mathbb{Z}_{2^{32}}$. All operations on the shares are performed with respect to the ring the shares belong to. In the following, $a_i \in \mathbb{Z}_{2^k}$ marks the share of CP_i and for a shared value $a \in \mathbb{Z}_{2^k}$ it holds that $a = a_1 + a_2$ as we are working in the two-party protection domain. A secret shared value is denoted by $[\![a]\!]$.

2.3 Proving security

The security goal of the current SHAREMIND framework is to be secure against a semi-honest adversary, meaning that the adversary will follow the protocol but might try to learn something extra from the messages it receives in the protocols. We expect that at most one of the two miners can be corrupted at a time. Additive secret sharing provides information theoretic security and it is possible to prove that also for protocols that use only additive secret sharing. However, the proposed multiplication protocol uses additively homomorphic encryption and hence is only secure in the computational model. This is not a serious drawback because if we consider the real implementations of the protocols, then, for example, the secure network traffic also uses computationally secure cryptographic measures and implemented protocols are only computationally secure.

The following security proofs follow the framework of general SHAREMIND security proofs as specified in [BLW08]. The protocols must be universally composable [Can01] so that they can be used in combination with one another. We can achieve universal composability in the semi-honest model by having perfectly simulatable protocols where the output shares are independent from the input shares. The latter can typically be achieved with ending each protocol with a perfectly secure re-sharing of the output shares.

A secure computation protocol is perfectly simulatable if there exists an efficient universal non-rewinding simulator S that can simulate all protocol messages received by any real world adversary A so that for all inputs, the output distributions of A and $S(A)$ coincide. For protocols that are symmetric to all participants we just need to show the security for one party, however asymmetric protocols require separate constructions for both parties.

In addition, given a message $a \pm r$ for $a, r \in \mathbb{Z}_{2^k}$, where r is a uniform random element, we can replace that message with r. If r is a random uniform element then so is $a \pm r$. We can use this equality to transform the views until they are trivially secure and easy to simulate.

3 Protocols

We have three different classes of participating parties: input parties (IP), computing parties (CP) and result parties (RP). For the described two-party protection domain we always have two computing parties denoted CP_1 and CP_2 accordingly. The current model supports any number of input and result parties, whereas all three classes can overlap.

The general setup defines additive secret sharing between two parties. However, for multiplication purposes, party CP_1 has defined a keypair for Paillier's cryptosystem and party CP_2 has learned the public key of party CP_1. We assume the existence of secure communication channels between the participants.

3.1 Resharing

The ideal functionality of the resharing protocol would have a trusted third party (TTP) who collects the shares $[\![a]\!]$ of both parties, restores this secret value and then sends both parties uniformly distributed new shares $[\![b]\!]$ of this value. The real functionality is described in Algorithm 1 and uses the randomization of the existing shares. We need to prove that this protocol is perfectly secure so that it can be used to achieve universal composability of other protocols. For this we must show that all attacks against this protocol can be carried out against the ideal model of the protocol.

Algorithm 1 Resharing protocol

Input: $[\![a]\!]$
Output: $[\![b]\!] = [\![a]\!]$, where $b_1,\ b_2$ are uniformly distributed and independent from $a_1,\ a_2$
CP_1 : generates random $r_1 \leftarrow \mathbb{Z}_{2^k}$
CP_2 : generates random $r_2 \leftarrow \mathbb{Z}_{2^k}$
CP_1 : sends r_1 to CP_2
CP_2 : sends r_2 to CP_1
CP_1 : computes $b_1 \leftarrow a_1 + r_1 - r_2$
CP_2 : computes $b_2 \leftarrow a_2 + r_2 - r_1$
return $[\![b]\!]$

Theorem 3.1.1. *The resharing Algorithm 1 is correct.*

Proof. To prove the correctness we need to show that $b = a$. For that we can expand the left hand side of the equation:

$$b = b_1 + b_2 = a_1 + r_1 - r_2 + a_2 + r_2 - r_1 = a_1 + a_2 = a.$$

□

Theorem 3.1.2. *The resharing Algorithm 1 is perfectly secure against a passive adversary.*

Proof. The resharing protocol is symmetric for both parties, hence, we can consider a general case of corrupted CP_i, for simplicity consider a corrupted CP_1. We can construct a non-rewinding interface between the real world adversary CP_1 and the ideal functionality. Firstly, the interface receives the randomness r_1 from the adversary. It then forwards the value $-r_1$ to the TTP, who also receives a_2 as an input from CP_2. The TTP restores the value $a_2 - r_1$ and sends the uniformly distributed

share b'_1 back to the interface, as well as forwards b'_2 to CP_2. The interface forwards $-b'_1$ to the party CP_1 who defines its share as $b_1 = a_1 + r_1 - (-b'_1)$. As party CP_2 received b'_2 from the trusted third party then the final shares $b = b_1 + b'_2 = a_1 + r_1 + b'_1 + b'_2 = a_1 + r_1 + a_2 - r_1 = a_1 + a_2 = a$. Both of the final shares b_1 and b'_2 are uniformly distributed and, therefore, the distributions in the real and ideal world coincide and the simulation is perfect. The non-rewinding property of the described interface ensures the universal composability of this protocol. □

3.2 Classify and declassify

The purpose of the classifying Algorithm 2 is to share the secret input of one party IP_i among both computing parties. Correspondingly the declassify Algorithm 3 defines how the shares are collected to allow the result parties RP_i to restore the secret value. The trivial version of declassify where parties forward their shares to each result party directly would leak information about the initial shares and therefore we use a protocol where the value is reshared before making it public.

Algorithm 2 Sharing of private inputs

Input: a from IP_i
Output: $[\![a]\!]$

IP_i : generates random $a_1 \leftarrow \mathbb{Z}_{2^k}$
IP_i : computes $a_2 = a - a_1$
IP_i : sends a_1 to CP_1
IP_i : sends a_2 to CP_2
return $[\![a]\!]$

Theorem 3.2.1. *The sharing Algorithm 2 is correct.*

Proof. This algorithm is trivially correct by the definition of the additive secret sharing as

$$a = a_1 + a_2 = a_1 + a - a_1 = a.$$

□

Algorithm 3 Restoring the shared value

Input: $[\![a]\!]$
Output: a

CP_1 & CP_2 : perform resharing Algorithm 1 with input $[\![a]\!]$ to obtain $[\![t]\!]$
CP_1 : sends t_1 to all RP_i
CP_2 : sends t_2 to all RP_i
for all RP_i **do**
  RP_i : computes $a = t_1 + t_2$
end for
return a

Theorem 3.2.2. *The declassify Algorithm 3 is correct.*

Proof. For correctness we need that $a = a_1 + a_2$:

$$a = t_1 + t_2 = a_1 + r_1 - r_2 + a_2 + r_2 - r_1 = a_1 + a_2$$

□

The declassifying Algorithm 3 is perfectly simulatable because both parties CP_i only receive uniformly distributed r_i and t_i values. However, it is clear that learning the value a will reveal the share a_1 to party CP_2 and vice versa a_2 to CP_1 as $a_1 = a - a_2$ if either of them belongs to the result parties. Note that the resharing performed as a part of the declassify protocol does not protect against this information leak and avoiding it is essentially impossible in the two party setting. The reason behind computing the values t_1 and t_2 is that the SHAREMIND framework supports the participation of parties who do not participate in the computation but learn the declassified results of the protocols. In this case, the miners send the value t_i also to the non-computing nodes. Hiding the real shares with randomness to get t_1 and t_2 assures that these non-computing nodes do not learn the shares of the computing miners. It is clear that t_i does not reveal anything about the share a_i because of the uniformly distributed randomness used to blind the value a_i when computing t_i also makes t_i a uniformly distributed value.

3.3 Addition and subtraction

The addition protocol in Algorithm 4 and the subtraction protocol (comments in Algorithm 4) can be computed locally based on the shares. The subtraction protocol is analogous to the addition protocol in Algorithm 4, where only the addition operation has been replaced by subtraction.

Algorithm 4 Addition protocol

Input: $[\![a]\!]$, $[\![b]\!]$
Output: $[\![c]\!] = [\![a + b]\!]$ {subtraction $[\![c]\!] = [\![a - b]\!]$}
  CP_1 : computes $c_1 = a_1 + b_1$ {subtraction $c_1 = a_1 - b_1$}
  CP_2 : computes $c_2 = a_2 + b_2$ {subtraction $c_2 = a_2 - b_2$}
  return $[\![c]\!]$

Theorem 3.3.1. *The addition and subtraction protocols in Algorithm 4 are correct.*

Proof. For correctness we need to show that $c = a + b$ and $c = a - b$ correspondingly:

$$c = c_1 + c_2 = a_1 + b_1 + a_2 + b_2 = a_1 + a_2 + b_1 + b_2 = a + b,$$
$$c = c_1 + c_2 = a_1 - b_1 + a_2 - b_2 = a_1 + a_2 - (b_1 + b_2) = a - b.$$

□

Theorem 3.3.1. *The addition and subtraction protocols in Algorithm 4 are secure against a passive adversary.*

Proof. The protocol run is perfectly simulatable as there is no communication, however, addition and subtraction protocols are not universally composable because the output shares depend on the inputs. For universal composability we need to combine the addition protocol with resharing Algorithm 1. □

3.4 Multiplication

Let σ denote a statistical security parameter, we use $\sigma = 112$ for medium term security. The following protocol uses an additively homomorphic public-key cryptosystem, for instance Paillier's cryptosystem, where $E(m)$ denotes encryption and $D(c)$ denotes decryption. The corresponding keys have been omitted, because we always use the key pair of party CP_1. Values computed under encryption are not computed modulo 2^k as the usual arithmetic operations on shares and therefore require modulo reduction after decryption.

Firstly, let CP_1 have the shares a_1 and b_1 whereas CP_2 has a_2 and b_2, where $a = a_1 + a_2$ and $b = b_1 + b_2$. The multiplication protocol enables these parties to respectively learn c_1 and c_2 such that $c = c_1 + c_2 = a \cdot b$ as described in Algorithm 5. Here, k denotes the length of the shared value data type and secret sharing is computed in ring $\mathbb{Z}_{2^k}$. The party CP_2 computes $E(a_1 \cdot b_2 + b_1 \cdot a_2 + r)$ under encryption using the homomorphic properties. Firstly, CP_2 uses constant multiplication to obtain $E(a_1 \cdot b_2)$ and $E(b_1 \cdot a_2)$ and, secondly, CP_2 encrypts $E(r)$ and uses the addition under encryption twice to compute the end result.

Algorithm 5 Multiplication of single values

Input: $[\![a]\!], [\![b]\!]$
Output: $[\![c]\!] = [\![a \cdot b]\!]$
CP_1 : sends $E(a_1)$ to CP_2
CP_1 : sends $E(b_1)$ to CP_2
CP_2 : generates random $r \leftarrow \{0,1\}^{2 \cdot k + 1 + \sigma}$
CP_2 : computes $c_2 = a_2 \cdot b_2 - r$
CP_2 : sends $E(v) = E(a_1 \cdot b_2 + b_1 \cdot a_2 + r)$ to CP_1
CP_1 : decrypts $v = D(E(v))$
CP_1 : computes $c_1 = (a_1 \cdot b_1 + v) \bmod 2^k$
return $[\![c]\!]$

Theorem 3.4.1. *The multiplication Algorithm 5 for single values is correct.*

Proof. To show that the multiplication is correct we need to prove that $c = a \cdot b$. For that we can expand c according to the values in the protocol as follows.

$$\begin{aligned} c &= c_1 + c_2 = a_1 \cdot b_1 + v + a_2 \cdot b_2 - r = a_1 \cdot b_1 + a_1 \cdot b_2 + b_1 \cdot a_2 + r + a_2 \cdot b_2 - r \\ &= a_1 \cdot b_1 + a_1 \cdot b_2 + b_1 \cdot a_2 + a_2 \cdot b_2 = (a_1 + a_2) \cdot (b_1 + b_2) = a \cdot b \end{aligned}$$

□

Theorem 3.4.2. *The multiplication Algorithm 5 for single values is secure against a passive adversary.*

Proof. The view of CP_2 is simulatable so that the simulation and real protocol runs are computationally indistinguishable. CP_2 sees two incoming messages in the defined protocol – $E(a_1)$ and $E(b_1)$. From the IND-CPA security of Paillier's cryptosystem we know that these encryptions are indistinguishable from encryptions of random elements. Therefore we could build a simulator that sends $E(r_1)$ and $E(r_2)$, where r_1 and r_2 are chosen uniformly, and the input in the simulation does not depend on the private values of party CP_2. If Paillier's cryptosystem is (t, ε)-indistinguishable

under chosen plaintext attacks, then the outputs of t-time CP_2 differ at most by 2ε in the simulation and the real protocol run.

The view of CP_1 is more complicated because the simulator can not just send a random encrypted element. We know that CP_1 can distinguish the simulator from the real life if the message $v < a_1 + b_1$ because the shares are all positive elements and $a_1b_2 + a_2b_1 \geq a_1 + b_1$ because these computations can not overflow the encryption modulus. However, we know that a_2 and b_2 are uniformly distributed values and we may choose uniform values also in the simulation. In addition, the simulator gets the randomness r as in the real protocol and calculates the response v as in the actual protocol. The values a_2 and b_2 have uniform distribution in both the simulation and the real protocol run and so does the final value v. This part of the simulation is perfect.

The second aspect from the view of CP_1 is that the value v may leak information about the length of a_2 and b_2 if it is shorter than $2k+1$ bits. However, the chance of choosing the randomness r so that all σ uppermost bits are 0 is $2^{-\sigma}$ which is negligible in the statistical security parameter σ.

The output share of CP_2 is independent from the input shares of CP_2 due to the randomness r that is used to hide the shares and has uniform distribution over $\mathbb{Z}_{2^k}$. The output share of the CP_1 is also uniformly distributed because of the randomness r that is indirectly included to the output through the input from party CP_2. □

The one value multiplication protocol in Algorithm 5 can be trivially extended for vector multiplication by just executing this protocol independently for all corresponding pairs in the vectors. However, as the length of the encrypted values and possible plaintexts exceeds the usual share value, then we could reduce the cost of communication by packing some values into one ciphertext. More precisely, if the secrets are in a ring $\mathbb{Z}_{2^k}$ and the Paillier modulus N has length $|N|$, then we can pack $|N|/l$ share values into one ciphertext in party CP_2 response, where $l = (2k + 1 + \sigma)$ or $l = (2k+2+\sigma)$. Variable l stands for the length of packed values and we consider two cases because the first may produce an error but the latter may be less efficient. Furthermore, CP_2 can encrypt the blinding value only once for the packed response. The details of this protocol for vectors of length $m \leq |N|/l$ are in Algorithm 6, which can be run multiple times to generalize it for longer vectors.

Theorem 3.4.3. *The multiplication Algorithm 6 is correct.*

Proof. To show that the multiplication is correct we need to prove that $c^i = a^i \cdot b^i$ for all i. For this it is necessary to understand what happens in the packing. The final value of $r = r_1||r_2||\ldots||r_m$ and the final value for $e = v_1||v_2||\ldots||v_m$, where each element v_i and r_i takes l bits and is padded with beginning zeros if necessary. The maximal actual length for each v_i is $2k+1$ bits if both inputs $[\![a]\!], [\![b]\!]$ are in the ring $\mathbb{Z}_2^k$. If $l = 2k + 2 + \sigma$ then the result of addition $r_i + v_i$ is also at most l bits and summing $e + r = vr_1||vr_2||\ldots||vr_m$ means that always $vr_i = v_i + r_i$ is computed as the single v in the single value multiplication. The correctness of the whole computation results from the correctness of each vr_i and the proof of theorem 3.4.1.

If $l = 2k + 1 + \sigma$, then there is a possibility $\epsilon < 2^{-\sigma}$ that the sum $r_i + v_i$ overflows the intended length and unpacking the response from CP_2 results in wrongly calculated vr_i. The error occurs if the σ most significant bits of r_i are all set and the sum of $2k + 1$ least significant bits of r_i and v_i overflows $2k + 1$ bits. However, this is acceptable in practice as the probability for this is negligible in the statistical security parameter σ. For all other cases the protocol is correct for the same reasons as for the longer l value.

□

Algorithm 6 Pairwise multiplication of vectors of length $m \leq |N|/l$

Input: $\langle [\![a^{(1)}]\!], .., [\![a^{(m)}]\!] \rangle$, $\langle [\![b^{(1)}]\!], .., [\![b^{(m)}]\!] \rangle$, where $(1), \ldots, (m)$ are indices
Output: $\langle [\![c^{(1)}]\!], .., [\![c^{(m)}]\!] \rangle$, where $[\![c^{(i)}]\!] = [\![a^{(i)} \cdot b^{(i)}]\!]$

CP_1 : sends $E(a_1^{(1)}), .., E(a_1^{(m)})$ to CP_2
CP_1 : sends $E(b_1^{(1)}), .., E(b_1^{(m)})$ to CP_2
CP_2 : fixes $r = 0$, $e = 0$, $E(e)$ as unbounded integers
for all $i \in \{1, \ldots, m\}$ **do**
  CP_2 : generates random $r_i \leftarrow \{0,1\}^{2 \cdot k + 1 + \sigma}$
  CP_2 : computes $E(v_i) = E(a_1^{(i)} \cdot b_2^{(i)} + b_1^{(i)} \cdot a_2^{(i)})$
  CP_2 : computes $r = r \cdot 2^l + r_i$
  CP_2 : computes $E(e) = E(e \cdot 2^l + v_i)$
  CP_2 : computes $c_2^{(i)} = a_2^{(i)} \cdot b_2^{(i)} - r_i \bmod 2^k$
end for
CP_2 : encrypts $E(r)$
CP_2 : sends $E(v) = E(e + r)$ to CP_1
CP_1 : decrypts and unpacks single values $vr_1 || vr_2 || \ldots || vr_m = D(E(v))$
for all $i \in \{1, \ldots, m\}$ **do**
  CP_1 : computes $c_1^{(i)} = a_1^{(i)} \cdot b_1^{(i)} + vr_i \bmod 2^k$
end for
return $[\![c]\!]$

Theorem 3.4.4. *The multiplication Algorithm 6 is secure against a passive adversary.*

Proof. The security results from the universal composability of the Algorithm 5 for multiplying a single value. The security of Algorithm 5 is shown in Theorem 3.4.2 where the non-rewinding simulators and independence of the output shares ensure universal composability. □

3.5 Multiplication using Beaver's triples

In the following we define a two-party multiplication protocol based on the ideas from [Bea91] where the multiplication triples were introduced. At high level the precomputation produces shared triples $\langle [\![a]\!], [\![b]\!], [\![c]\!] \rangle$ where $c = a \cdot b$ and these stored triples can be used to perform fast multiplication. In general the following protocol for multiplication is the same as described in [BDOZ10] except that it has been stripped of the measures against an active adversary — the zero-knowledge proofs and message authentication codes. Furthermore, the general protocol for triple generation as introduced in [BDOZ10] can be simplified for the two party case so that it becomes the multiplication protocol previously described in Algorithm 6. The resulting multiplication algorithm consists of the precomputation phase in Algorithm 7 and the online computation phase in Algorithm 8. Each precomputed triple can be used only once and must be discarded after use.

Theorem 3.5.1. *The triple generation Algorithm 7 is correct.*

Proof. For correctness we need that $[\![x^{(i)}]\!] \cdot [\![y^{(i)}]\!] = [\![z^{(i)}]\!]$ for $i \in \{1, \ldots, m\}$. The correctness proof 3.4.3 for the multiplication Algorithm 6 also proves that the Algorithm 7 for precomputing the Beaver's triples results in correct triples where $z = x \cdot y$. □

Algorithm 7 Precomputing the Beaver's triples for multiplication

Input: number of triples m
Output: triples $\langle [\![x^{(i)}]\!], [\![y^{(i)}]\!], [\![z^{(i)}]\!] \rangle$ where $z = x \cdot y$ for $i \in \{1, \dots m\}$

CP_1 : generate random $x_1^{(1)}, \dots, x_1^{(m)}, y_1^{(1)}, \dots, y_1^{(m)} \leftarrow \{0,1\}^k$
CP_2 : generate random $x_2^{(1)}, \dots, x_2^{(m)}, y_2^{(1)}, \dots, y_2^{(m)} \leftarrow \{0,1\}^k$
run multiplication Algorithm 6 with inputs $[\![x]\!], [\![y]\!]$ and output $[\![z]\!]$
return triples $\langle [\![x^{(i)}]\!], [\![y^{(i)}]\!], [\![z^{(i)}]\!] \rangle$

Algorithm 8 Pairwise multiplication of vectors of length m using Beaver's triples

Input: $\langle [\![a^{(1)}]\!], .., [\![a^{(m)}]\!] \rangle$, $\langle [\![b^{(1)}]\!], .., [\![b^{(m)}]\!] \rangle$, where $(1), \dots, (m)$ are indexes
Output: $\langle [\![c^{(1)}]\!], .., [\![c^{(m)}]\!] \rangle$, where $[\![c^{(i)}]\!] = [\![a^{(i)} \cdot b^{(i)}]\!]$

for all $i \in \{1, .., m\}$ **do**
parties choose a triple $\langle [\![x^{(i)}]\!], [\![y^{(i)}]\!], [\![z^{(i)}]\!] \rangle$
parties execute the subtraction protocol $[\![e]\!] = [\![a^{(i)}]\!] - [\![x^{(i)}]\!]$
parties execute the subtraction protocol $[\![w]\!] = [\![b^{(i)}]\!] - [\![y^{(i)}]\!]$
parties execute the declassify protocol for $[\![e]\!]$ and $[\![w]\!]$
CP_1 : computes $c_1^{(i)} = z_1^{(i)} + e \cdot y_1^{(i)} + w \cdot x_1^{(i)} + e \cdot w$
CP_2 : computes $c_1^{(i)} = z_2^{(i)} + e \cdot y_2^{(i)} + w \cdot x_2^{(i)}$
end for
return $[\![c]\!]$

Theorem 3.5.2. *The multiplication Algorithm 8 that generated the Beaver's triple with Algorithm 7 is correct.*

Proof. To prove the correctness of Algorithm 8 we need to consider if $c = a \cdot b$ for each single value in the vector.

$$\begin{aligned} c &= c_1 + c_2 = z_1 + e \cdot y_1 + w \cdot x_1 + e \cdot w + z_2 + e \cdot y_2 + w \cdot x_2 \\ &= z + e \cdot y + w \cdot x + e \cdot w = z + (a - x) \cdot y + (b - y) \cdot x + (a - x) \cdot (b - y) \\ &= z + a \cdot y - x \cdot y + b \cdot x - x \cdot y + a \cdot b - a \cdot y - x \cdot b + x \cdot y = z - x \cdot y + a \cdot b \\ &= z - z + a \cdot b = a \cdot b \end{aligned}$$

□

Theorem 3.5.3. *The Algorithm 7 for computing Beaver's triples is secure against a passive adversary.*

Proof. The security of the triple precomputation results from the security of the multiplication Algorithm 6 stated in Theorem 3.4.4. □

Theorem 3.5.4. *The multiplication Algorithm 8 using Beaver's triples is secure against a passive adversary.*

Proof. The multiplication protocol is mostly combined from the previously introduced protocols for subtraction and declassification. The subtraction protocol is perfectly simulatable according

to Theorem 3.3.1. The value that is declassified has been blinded using the values from the triple and does not reveal anything about the input shares of the party. The remaining part of the protocol requires only local computations and is therefore easy to simulate, as a result all parts of the protocol are simulatable. In addition, as the triple is used only once and is discarded after the protocol then we can view it as the randomness and therefore the output shares are independent from the input shares and this protocol is universally composable. □

4 Implementation details

The SHAREMIND framework is implemented in C++ and so is the additional protection domain described in this paper. There are several additional libraries needed for SHAREMIND, but the given two-party protection domain only adds the need for the NTL number theory library [Sho].

The current implementation has a hard-coded statistical security parameter $\sigma = 112$ which stands for medium term security and the demo keys therefore have a 2048-bit modulus. A further restriction is that currently the *uint32* is the only secret data type provided in the implementation of the protection domain. However, adding other integer types should not be difficult as they are supported by the implementation of Paillier's cryptosystem and would work with all the proposed protocols without modifications.

The current encoding of ciphertext vectors where elements are of type NTL::ZZ only allows to use ciphertexts with size of 65536 bytes, because the length of the ciphertext is encoded using two bytes. This is sufficient for the chosen 2048-bit modulus that results in 4096 bits of ciphertext length. Furthermore, the current encoding would also suit a modulus of 4096 bits.

4.1 Paillier's cryptosystem

Paillier's cryptosystem is implemented using the NTL [Sho] number theoretic library for C++ that was developed by Victor Shoup. This library was mainly chosen for the readability of the code written with NTL. However, as there were no efficiency comparisons between the number theoretic libraries for C++ before this choice, then it is likely that there are libraries that would prove more efficient. Furthermore, NTL can be compiled using GNU Multi-Precision library [Gra] to achieve better performance of long integer arithmetic.

The keys for the cryptosystem are read from PEM encoded RSA key files using the OpenSSL [YH] API for C++. Suitable keys can, for example, be generated using the OpenSSL tool. Keeping keys as RSA key files is not space efficient as the RSA keys have many extra values that are not necessary for the Paillier's cryptosystem. However, this representation has been chosen for the simplicity of both, generating new strong keys and parsing the key files. There is no standard representation of the Paillier's keys yet, if such a standard is created then is should be used instead of currently used RSA keys.

The Paillier's cryptosystem implementation has all of the aforementioned improvements except the precomputation of the encryption randomness. However, precomputing the randomness should be added to achieve considerable speed-ups for the encryption and therefore also for the multiplication protocol. Each miner can compute and store randomness for its own use by performing only local operations. Ideally, the miner could use the times when it is otherwise idle to compute the randomness and use the precomputed randomness during encryption. However, this would also mean, that in case of no idle time and active encryption the miner might occasionally still need to

compute the randomness during encryption because there may not be enough precomputed values available.

4.2 Multiplication

The multiplication protocol puts a lot of workload on party CP_1 who has to separately encrypt all of its input elements. Considerable speed-ups can only be achieved if the encryption function is as fast as possible. Although precomputation can shorten the protocol execution time, it is important to notice that it does not reduce the overall workload of the miners, just enables to distribute the workload more evenly during the miner uptime.

Nevertheless, taking the large workload on CP_1 side into account can help to speed up the protocol on the side of CP_2, who can perform precomputations before waiting for inputs from CP_1 and can reduce the workload when actually computing on the inputs from CP_2. Analysing the protocol in Algorithm 6 reveals that party CP_2 can precompute its final share and the randomness r without knowing the inputs from CP_1. As the main improvement, CP_2 can also perform the expensive encryption operation at the precomputation phase. This can improve the running time because it is highly likely that CP_2 finished all the precomputations before it even could receive the vectors. However, even in the case of a very fast CP_1 the party CP_2 would still need to perform all these computations and such ordering should not have a big negative effect on the running time.

The vector multiplication Algorithm 6 introduced the possibility to pack several secret values into one ciphertext. For the currently chosen 2048-bit modulus, $\sigma = 112$ and 32-bit secret values we can pack 11 elements regardless of the exact value for l used for packing. For long term security, we should choose a 3072-bit modulus and $\sigma = 128$ which will result in the possibility to pack 15 elements into one ciphertext.

Multiplication is currently implemented using batches so that long vectors get broken to smaller sections. This segmentation mainly aims to reduce the size of network messages as instead of sending the whole vector at once we send a suitably sized batch. The batch size should in the future be determined from actual network settings and can be fixed in the configuration file of the protection domain. As long as there is a single secret value type it would make sense to choose the batch size as a multiple of the number of elements that can be packed together to achieve maximum efficiency from the packing.

4.2.1 Possible improvements

CP_2 may gain from finding one random element of length $l \times \textit{packing count}$ to add to the concatenated ciphertext and then chopping it up to get all random r_i values. Using the NTL randomness generator once is faster than calling it packing count times to get all r_i separately. Splitting this one randomness to different r_i values only requires cheaper truncating and bitshifting operations. However, this optimization can only be implemented easily if we expect l to be the same as the security constant, otherwise the randomness should have fixed zeros in places of every l'th bits to avoid overflowing one packed element and the size of r_i. We need to find a shorter final randomness if we are processing a shorter array than the maximum amount we can pack to one ciphertext.

In addition, we may try to make the algorithm symmetric by introducing also a key pair of CP_2 and divide the inputs to two parts so that both participants encrypt half of the inputs. Both parties would process the encrypted values received from the other miner as CP_2 processes the values from CP_1 in Algorithm 6. This could speed up the general protocol approximately 2 times. Using two

Table 1: Comparison of arithmetic operations between protection domains (milliseconds)

	3 miners		2 miners			
length	add	multiply	add	triples	multiply CP_1	multiply CP_2
1000	0.033	27.538	0.030	37402.992	21.061	0.201
10000	0.309	59.717	0.298	376045.833	25.239	5.277
20000	0.603	91.396	0.571	754546.683	30.649	13.340
100000	2.556	414.367	2.548	3759854.983	94.252	81.104

sets of keys do not affect the further usage of the multiplication results or the triples in case we perform this as a precomputation protocol.

4.3 Multiplication using Beaver's triples

This protocol requires a precomputation mechanism for SHAREMIND as this protocol is composed of two distinct steps - the precomputation phase and the online multiplication phase. However, this precomputation differs considerably from the precomputation of encryption randomness because it requires the two miners to perform an interactive protocol during the precomputation. Hence, precomputation requires the two computing parties to decide when to run this protocol.

The current test implementation just performs precomputation at the beginning of the multiplication protocol. SHAREMIND needs a mechanism to perform precomputation at start-up and during idle times, to fully integrate this protocol idea. This may also require tests to estimate what is the amount of precomputed triples that should be calculated initially and stored during the uptime.

The online phase of the protocol is very simple and can be implemented as described in Algorithm 8. However, as the values that are declassified are blinded with the random triple then there is no need to actually blind these values as in the full declassify Algorithm 3. In addition, there is no need to declassify these values to non-computing nodes as they provide no information about the protocol result.

5 Performance measurements

All measurements in this section are performed while running all miners on a single machine. This setup reduces the time for network communication, but also means that the miners compete for the processor time and the results in a distributed system might differ considerably from the results presented here. These results are given for the purpose of comparing the protection domains with two or three parties and unless specified differently, the times are average over all the participating miners.

The measurements were performed on a Intel(R) Core(TM)2 Duo CPU 2.66 GHz Linux machine with 4 GB of RAM. The fragment size for three-party multiplication is 100000 and for two parties it is 500 unless specified differently. The online phase of the triple multiplication does not use fragmentation at the moment, however it may be beneficial to add it in order to keep the network messages smaller. There is an artificial gap between the online and offline phase of the multiplication so that the running times of these separate parts should not affect each other.

Table 1 compares the average running times of addition and multiplication algorithms in the two protection domains. The addition algorithm is the same for the two domains and the slightly

slower times for the three party version should result from the fact that all miners used the same machine and therefore three-party version had more competition for the processor time. As addition is performed locally and miners do the same amount of work in both domains then in real life, the running times for the addition protocol should coincide. Nevertheless, the ideas for multiplication algorithms are different and therefore form the more interesting part of the comparison.

The times for triple generation increase linearly as the input length increases. This is expected because most of the protocol time is taken by the party CP_1 who has to encrypt all of its inputs separately. In addition, triple generation is slightly more time consuming for CP_1 who must also process the last network message from CP_2, whereas CP_2 finishes with sending the last message. It would be possible to achieve about 2 times speed-up by using keys for both parties and making the protocol symmetric.

The online multiplication is in general faster than the three-party multiplication, but shows confusing trends as the lengths increase. This protocol also puts slightly more workload on CP_1, who has to compute an extra local multiplication and addition compared to CP_2. This difference well explains the case of length 100000 running times, however, the gaps between running times of other cases seem too big to be caused by just some local operations. Interestingly, even if we multiply only one value we currently get 0.02 ms for CP_2, but still 19.8 ms for CP_1. This phenomena could be caused by the timing of processor threads, especially because some of the measurements showed equal times for the miners. However, there is no good explanation at the moment.

The different l values result in 376.0 seconds on average for $l = 2k + 1 + \sigma$ and 375.5 seconds on average for $l = 2k + 2 + \sigma$ for 10000 elements multiplication. Therefore, there is no significant difference at the moment and this question should be reconsidered once the protocol in general becomes faster. As long as there is no difference, we could use $l = 2k + 1 + \sigma$ which will allow to compute the randomness faster as described in Section 4.2.1.

All previous measurements were performed with the multiplication fragment size 500, choosing 2000 results in an average of 377.5 seconds, choosing 1000 gives an average of 375.8 seconds, and choosing 250 gives an average of 376.1 seconds for length 10000. These results do not distinguish any of the choices, but they may give different results in real networks where a suitable parameter should be chosen for each setup.

Precomputing the exponentiation of the randomness can decrease the running time of encryption operation about 100 times and should also considerably affect the time requirements of the multiplication protocol that uses homomorphic encryption. However, it is questionable if it would give any noticeable speed-up in the final setting, where we only need encryption as part of the precomputation phase.

6 Conclusion

This report introduced the basic protocols for secure two-party computation based on additive secret sharing. Most of the protocols were directly ported from the tree-party protocols of Sharemind, the only exception being the multiplication protocol. The proposed multiplication protocol uses the homomorphic properties of Paillier's cryptosystem to perform multiplication by combining multiplication with a public value and addition under encryption. In addition it is possible to pack several values into one ciphertext to reduce the cost of communication. The time requirement for the Paillier encryption is currently the main difficulty of the protocol. This problem could be reduced by precomputing the randomness for encryption, but the current implementation does not have a mechanism for precomputation.

The second possibility is to use the multiplication protocol based on additively homomorphic encryption as the precomputation phase. This protocol can be used to produce Beaver's triples and then use a simple multiplication protocol which requires these triples. The experiments show that the triple multiplication protocol is usable for practical purposes, whereas the initial idea was time-consuming.

Further work could consider different ways to more efficiently pack the values into a ciphertext to reduce the amount of network traffic and separate encryptions. It would be beneficial to precompute the randomness of encryption and possibly check the efficiency of NTL functions. In addition, it would be interesting to develop other protocols using precomputation ideas similar to the final multiplication. Furthermore, it would be important to establish a framework for precomputing protocols and storage of the precomputed values. Finally, it could be beneficial to add a protocol to alter between the two protection domains.

7 Acknowledgments

The authors would like to thank Sven Laur for helpful discussions and suggestions.

References

[BDOZ10] Rikke Bendlin, Ivan Damgrd, Claudio Orlandi, and Sarah Zakarias. Semi-homomorphic encryption and multiparty computation. Cryptology ePrint Archive, Report 2010/514, 2010.

[Bea91] Donald Beaver. Efficient multiparty protocols using circuit randomization. In Joan Feigenbaum, editor, *Proceedings of the 11th Annual International Cryptology Conference. CRYPTO '91*, volume 576 of *Lecture Notes in Computer Science*, pages 420–432. Springer, 1991.

[BLW08] Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: a framework for fast privacy-preserving computations. Cryptology ePrint Archive, Report 2008/289, 2008. `http://eprint.iacr.org/`.

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

[DJ01] Ivan Damgård and Mats Jurik. A generalisation, a simplification and some applications of paillier's probabilistic public-key system. In *Proceedings of the 4th International Workshop on Practice and Theory in Public Key Cryptography: Public Key Cryptography*, PKC '01, pages 119–136, London, UK, UK, 2001. Springer-Verlag.

[Gir] Damien Giry. BlueKrypt - cryptographic key length recommendation. `http://www.keylength.com`.

[Gra] Torbjörn Granlund. The GNU multiple precision arithmetic library. `http://gmplib.org/`.

[Pai99] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Proceedings of the 17th international conference on Theory and application of cryptographic techniques*, EUROCRYPT'99, pages 223–238, Berlin, Heidelberg, 1999. Springer-Verlag.

[Sho] Victor Shoup. NTL: A library for doing number theory. `http://www.shoup.net/ntl/`.

[YH] Eric A. Young and Tim J. Hudson. OpenSSL: Cryptography and SSL/TLS toolkit. `http://www.openssl.org/`.