Asynchronous Multiparty Computation: Theory and Implementation

Ivan Damgård, Martin Geisler, Mikkel Krøigaard, and Jesper Buus Nielsen*

Dept. of Computer Science, University of Aarhus, Denmark
{ivan,mg,mk,buus}@cs.au.dk

Abstract. We propose an asynchronous protocol for general multiparty computation with perfect security and communication complexity $\mathcal{O}(n^2|C|k)$ where n is the number of parties, $|C|$ is the size of the arithmetic circuit being computed, and k is the size of elements in the underlying field. The protocol guarantees termination if the adversary allows a preprocessing phase to terminate, in which no information is released. The communication complexity of this protocol is the same as that of a passively secure solution up to a constant factor. It is secure against an adaptive and active adversary corrupting less than $n/3$ players. We also present a software framework for implementation of asynchronous protocols called VIFF (Virtual Ideal Functionality Framework), which allows automatic parallelization of primitive operations such as secure multiplications, without having to resort to complicated multithreading. Benchmarking of a VIFF implementation of our protocol confirms that it is applicable to practical non-trivial secure computations.

* Supported by Ministry of Science, Technology and Innovation

Table of Contents

1 Introduction

A general multiparty computation protocol is an extremely powerful tool that allows n parties to compute any agreed function $f(x_1, ..., x_n)$, where each input x_i is privately held by the i'th player P_i, and where only the intended result becomes known to the players. The function is often represented by an arithmetic circuit C over some suitable finite field $\mathbb{F}$. It is required that privacy of the inputs and correctness of the result is ensured even in the presence of an adversary who may corrupt some number t of the players.

From the basic feasibility results of the late 80-ties [3, 6], it follows that any efficiently computable function may be computed securely in the model where players have secure point-to-point channels, if and only if $t < n/3$. In case the adversary is passive, i.e., corrupted players follow the protocol, the bound is $t < n/2$. Under computational assumptions, $t < n/2$ corruptions can be tolerated even if the adversary is active, i.e., corrupt players behave arbitrarily [9].

The solution from [3] with passive security can lead to quite practical solutions, when combined with later techniques for optimizing the efficiency of particular primitives, such as integer comparison – even to the point where large-scale practical applications can be handled [4].

On the other hand, this type of solution is not satisfactory in all cases. It is of course desirable to provide security against active cheating. However, this usually incurs a large cost in terms of efficiency. Techniques have been proposed to reduce this cost [10], but they – like most previous protocols – are designed for synchronous communication. Common ways to communicate such as the Internet are arguably better modeled as asynchronous networks, where there is no guarantee that a message is delivered before a certain time. Note that the way we model the network can have dramatic consequences for the practical efficiency of a protocol. Consider a network that usually delivers messages fast, but occasionally takes much longer time. If we run a synchronous protocol on top of such a network, we are forced to make every round last enough time so that we can be sure that all messages from honest players have been delivered. Otherwise, we may conclude that an honest player is corrupt because he did not send the message he was supposed to, and take action accordingly. Now, of course, the protocol is no longer secure. A synchronous protocol may therefore be much slower in practice than an asynchronous one, where every player may continue as soon as he has enough information to do so.

In the project reported on in this paper, our goal was therefore to develop and implement a practical general MPC protocol, with active security on an asynchronous network. Compared to the usual model for asynchronous MPC, we make two modifications, both of which we believe are well motivated:

- We allow our protocol to have one synchronization point. More precisely, the assumption is that we can set a certain time-out, and all messages sent by honest players before the deadline will also be delivered before the deadline.
- We do not guarantee that the protocol always terminates and gives output to all honest players. Instead we require the following: The preprocessing phase of the protocol, up to the synchronization point, never releases any new information to the adversary. The adversary may cause the preprocessing to fail, but if it terminates successfully, the entire protocol is guaranteed to terminate with output to all honest parties.

A discussion of this model: Without the first assumption, i.e., if the protocol is fully asynchronous, one cannot guarantee that all honest players will be able to contribute input

since the protocol cannot distinguish between t corrupt players that have not sent anything, and t honest players whose messages have been delayed. We believe that in most practical applications, this is not acceptable, and this is why we introduce the synchronization point, it is a minimal assumption allowing all honest players to provide input. We stress that the protocol is *asynchronous both before and after the synchronization point*. In other words, a protocol in this model is free to harvest the efficiency gain that follows from players being able to proceed as soon as possible. The only constraint we put is that honest players must reach the deadline on time, so we can have agreement on whether the preprocessing succeeded.

On the second point, although we do give the adversary extra power to stop the protocol, this is arguably of no use in practice: If the corrupted players stop the protocol at a point where no information is released, why play the game in the first place?

In this model, assuming secure point-to-point channels and that Byzantine agreement is available, we present a protocol that is perfectly secure against an adaptive and active adversary corrupting less than $n/3$ of the players. The communication and computational complexities (total communcation and work done) are $\mathcal{O}(n^2|C|k)$ where $|C|$ is the size of the arithmetic circuit being computed and k is the bit length of elements in the field used. It is noteworthy that a straightforward implementation with only passive security would have the same asymptotic complexity, all other things being equal.

As for any protocol in the point-to-point model, the exact security properties of an actual implementation of our protocol depend on how the point-to-point channels and – in our case – the Byzantine agreement are implemented. The choice of implementation does not, however, affect the complexities since the Byzantine agreement is only used once. In a typical implementation where one goes for efficiency – such as the one we present below – one would use standard encryption tools to implement the channels and do the Byzantine agreement based on public-key signatures. This gives a protocol with computational security against a static adversary (although it may also be secure against an adaptive adversary).

In recent concurrent work, Hirt *et al.* [11] construct an asynchronous protocol of similar asymptotic complexity as ours. This protocol is fully asynchronous, so it does not guarantee that all honest parties can provide inputs, it is computationally secure, against a static adversary. In another recent work Beerliová-Trubíniová *et al.* [2] present a protocol with a single synchronization point like we have. This protocol guarantees termination, has a better security threshold ($n/2$), but is only computationally secure against a static adversary, and has larger asymptotic complexity than our protocol. Both [2] and [11] make use of expensive public-key techniques throughout the protocol, making them less efficient in practice than our construction.

Thus, our result is incomparable to previous work, and we believe it provides a new tradeoff between security properties that is attractive in practice. We later give more exact numeric evidence of the efficiency.

Our protocol is based on Beaver's well known circuit randomization techniques, where one creates in a preprocessing phase shared random values a, b, c with $ab = c$. We show two techniques for generating these triples efficiently. One is a variant of the protocol from [1], the other is based on pseudorandom secret sharing [7], it is much faster for a small number of players, but only gives computational security. Both protocols are actually synchronous, but we handle this via a new technique that may be of independent interest, namely a general method by which – if one accepts that the protocol may abort – a synchronous protocol can be executed in an asynchronous fashion, using a single synchronization point to decide if the protocol succeeded.

A crucial observation we make is that if the protocol is based on Shamir secret sharing with threshold less than $n/3$, then the computation phase can be done asynchronously and still guarantee termination, if the preprocessing succeeded.

Another contribution of our paper is a software framework called VIFF, short for Virtual Ideal Functionality Framework. It provides a platform on which general MPC protocols can be implemented, and we use it later in the paper to benchmark our protocol. Protocols implemented in VIFF can be compositions of basic primitives like addition and multiplication of secret-shared values, or one can implement new primitives. VIFF is basically asynchronous and operates on the principle that players proceed whenever possible (but can handle synchronization points when asked to do so). This allows us to provide all protocol implementations with automatic parallel scheduling of the operations, i.e., the programmer does not have to explicitly use multithreading, for instance, or specify any explicit timing of operations.

When players distributed across a large network execute a large protocol, it is very important to be able to run as much as possible in parallel in order to lower the cost per operation of the network delays. Round-trip times across the Internet are typically in the order of 100–200 milliseconds, but when executing many multiplications in parallel we are able to obtain an average time of just 2 milliseconds per secure multiplication of 32-bit numbers, using a standard implementation based on Shamir secret-sharing, for 3 players and passive security.

Furthermore, the ability to program asynchronously is very important towards having simpler code: If the protocol to be implemented is synchronous, one has to implement waiting to make sure that all messages from the current round have arrived, and the actual waiting time has to be chosen correctly with respect to the network we use. This means that the software now depends on the underlying network which is clearly undesirable, as it creates an extra source of errors, insecurity, or both.

2 Preliminaries

For an element $x \in \mathbb{F}$ we let $[x]_d$ denote a set of Shamir shares [13] of x computed using threshold/degree d. We use the shorthand $[x]$ for sharings $[x]_t$ where t is the number of corrupted players, so that $t < n/3$. We use the notation $[x] + a[y]$ where a is a public constant to denote the set of shares obtained by locally adding the share of x to the share of y times a. Since Shamir sharing is linear, we have $[x] + a[y] = [x + ay]$.

When in the following, we say that x is *publicly reconstructed* from $[x]_t$, where at most $t < n/3$ players are actively corrupted, this simply means that each player sends his share to all other players. This allows all honest players to reconstruct x using standard decoding techniques since $t < n/3$. We may also *privately open* x to player P_i by sending shares only to him.

3 Overview and Security Model

The goal of the protocol is to securely compute $(y_1, \ldots, y_n) = f(x_1, \ldots, x_n)$. For notational convenience we assume that all inputs and outputs are single field elements. In addition each y_i can assume the value $y_i = \perp$, which indicates to P_i that the computation failed.

3.1 Overview of the Protocol

Our protocol consists of two phases, the *preprocess and input phase* and the *computation phase*.

Preprocessing and input phase. In the preprocessing phase, we can make use of any protocol that can generate a given number of multiplication triples, i.e., random secret-shared values $[a], [b], [c]$ where $ab = c$. In addition, for each player P_i, it should construct a secret sharing $[r_i]$ where r_i is random and reveal r_i privately to P_i. The protocol ends by outputting "success" or "failure" to all players, depending on whether the required values were successfully constructed or not. The purpose of $[r_i]$ is to allow P_i to contribute his input x_i securely by broadcasting $r_i + x_i$.

Instead of attempting to build such a protocol directly for the asynchronous model, it is much easier to design a protocol for the synchronous model with broadcast, we give two examples of this in Section 4. We then show below a special way to run any such protocol in an asynchronous way, i.e., we can avoid the use of timeouts after each communication round and we avoid having to implement broadcast. The price we pay for this is that the adversary can force the preprocessing to fail.

The basic idea is that in each round all parties just wait for messages from all other parties and progress to the next round immediately if and when they all arrived. Some extra technicalities are needed to make sure there is agreement at the end on whether the preprocessing succeeded, and to make sure that no information on the inputs is revealed prematurely.

To emulate a synchronous protocol with R rounds, each P_j proceeds as follows:

1. Wait for an input begin preprocess. Let $r = 1$ and for each P_i compute the message $m_{j,i,1}$ to be sent to P_i in the first round of the synchronous protocol. Also compute the message $m_{j,1}$ to be broadcast in the first round.
2. Send $(m_{j,i,1}, m_{j,1})$ to P_i.
3. While $r \leq R$:
   (a) Wait until a message $(m_{i,j,r}, m_{i,r})$ arrived from all P_i.
   (b) From the incoming messages $((m_{1,j,r}, m_{1,r}), \ldots, (m_{n,j,r}, m_{n,r}))$ compute the messages $(m_{j,1,r+1}, \ldots, m_{j,n,r+1})$ that the preprocessing protocol wants to send in the next round, and the message $m_{j,r+1}$ to be broadcast.
   (c) $r := r + 1$.
4. Let $g_j \in \{\text{preprocess success}, \text{preprocess failure}\}$ denote the output of the preprocessing protocol and let M_j consist of the broadcast messages $m_{i,r}$ for $i = 1, \ldots, n$ and $r = 1, \ldots, R$. Send (check, g_j, M_j) to all parties.
5. Wait until all $n-1$ other parties P_i send (check, g_i, M_i). If all P_i sent $g_i = \text{preprocess success}$ and $M_i = M_j$, then send $s_i = x_i + r_i$ to all parties.
6. Wait to receive s_j from all other parties, let $S_j = (s_1, \ldots, s_n)$ and send S_j to all parties.
7. If all $n - 1$ other parties P_i sent some S_i before the timeout and all $S_i = S_j$, then let $q_i = \text{success}$. Otherwise, let $q_i = \text{failure}$.
8. Run a Byzantine agreement (BA) on the q_i to agree on a common value $q \in \{\text{failure}, \text{success}\}$. Being a BA this protocol ensures that if $q_i = \text{success}$ for all honest parties, then $q = \text{success}$, and if $q_i = \text{failure}$ for all honest parties, then $q = \text{failure}$.

We assume that the preprocessing phase is started enough in advance of the time-out to guarantee that it will terminate successfully on time when there is no cheating. However, as mentioned in the introduction, the adversary can stop the preprocessing, in particular if a corrupted party does not send a message the preprocessing dead-locks.

Note that if just one honest party outputs $q_i = \text{success}$, then the preprocessing protocol terminated successfully before the timeout and all the values s_i were consistently distributed.

In particular, if $q = \mathsf{success}$, then $q_i = \mathsf{success}$ for at least one honest P_i, and therefore the preprocessing and inputting were successful.

As for security, if after each communication round in Step 3 the parties compared the messages $m_{i,r}$ and terminated if there was disagreement, then it is clear that a secure synchronous protocol[1] run asynchronously this way is again secure. The only loss is that the adversary can now deprive some parties of their input. The reason why it is secure to postpone the check of consistency of the broadcasted message until Step 5 is that the inputs x_i do not enter the computation until Step 6 and that there are no other secrets to be leaked, like secret keys. Sending inconsistent broadcast messages before Step 6 will therefore yield no information leakage. After Step 5 it is known that the preprocessing was an emulation of a consistent synchronous execution, at which point it becomes secure to use the result r_i to mask x_i.

This way to emulate a syncronous protocol in an asyncronous environment is generic and does not just apply to our protocols here.

Computation phase. If $q = \mathsf{failure}$, then all parties output $y_i = \perp$. If $q = \mathsf{success}$, then the parties compute $[x_i] = s_i - [r_i]$ for all P_i and run the asynchronous protocol described below which compute sharings $[y_i]$ of the outputs from the sharings $[x_i]$, making use of the multiplication triples from the preprocessing. Finally the shares of $[y_i]$ are sent privately to P_i which computes y_i.

We may assume that for each multiplication we have to do, a triple $[a], [b], [c]$ as described above is constructed in the preprocessing. To handle any arithmetic circuit describing the desired function, we then only need to describe how to deal with linear combinations and multiplications of shared values.

Linear Combinations: Shamir sharing is linear, and any linear function of shared values can therefore be computed locally by applying the same linear function to the shares.

Multiplication: Consider some multiplication gate in the circuit and let $[a], [b], [c]$ be the triple constructed for this gate. Assume we have computed sharings of the two input values $[x]$ and $[y]$, so we now wish to compute $[xy]$. Note that

$$\begin{aligned} xy &= ((x-a)+a)((y-b)+b) \\ &= de + db + ae + ab, \end{aligned}$$

where $d = x - a$ and $e = y - b$. We may now publicly reconstruct d and e, since they are just random values in $\mathbb{F}$. The product can then be computed locally as

$$[xy] = de + d[b] + [a]e + [c] \ .$$

The overall cost of this multiplication is the cost of two public reconstructions and a constant number of local arithmetic operations.

A crucial observation is that this protocol (assuming the triples are given) can be executed in a completely asynchronous fashion, and is guaranteed to terminate: At each multiplication gate, each player simply waits until he has received enough shares of d and e and then reconstructs them. More precisely, we need that at least $n - t$ shares of each value have arrived, and that at least $n - t$ of them are consistent with some polynomial. Since there are

[1] The synchronous security should be against a rushing adversary.

$n-t$ honest players, $n-t$ consistent shares will eventually arrive. Moreover, if $n-t$ shares are found to be consistent, since $t < n/3$, these must include at least $t+1$ shares from honest players, and so the correct value is always reconstructed. One can test if the conditions are satisfied using standard error correction techniques.

3.2 Security Model

The security of our protocol can be phrased in the UC framework [5]. For the protocol we assume the standard asynchronous communication model of the UC model, except that we let the timeout of P_i be called by the adversary by inputting timeout to that party, and that we assume secure point to point channels where the adversary can decide when a message sent is delivered. Our protocols are secure and terminate no matter when the timeouts are called. They provide outputs, $\neq \perp$, if all parties behave honestly in the preprocessing and the timeouts are called after the preprocessing succeeded at all honest parties. We formalize that by implementing an ideal functionality.

For a function $f\colon \mathbb{F}^n \to \mathbb{F}^n$, let $\mathcal{F}_{\text{FSFE}}^f$ be the following ideal functionality for fair secure function evaluation.

1. On input begin preprocess from P_i, output (P_i, begin preprocess) to the adversary.
2. On input x_i from P_i, output (P_i, gave input) to the adversary.
3. If the adversary inputs early timeout, then output $y_i = \perp$ to all P_i, and terminate.
4. If all P_i have input both begin preprocess and x_i and the adversary then inputs late timeout, then compute $(y_1, \ldots, y_n) = f(x_1, \ldots, x_n)$ and output y_i to all P_i, and terminate.

Note that the adversary can always make the evaluation fail, but must do so in a fair way: Either no party learns anything, or all parties learn a correct output. Our protocol securely implements this ideal functionality when $t < n/3$ parties are corrupted. If the BA is modeled as an ideal functionality, then our implementation is perfectly secure. We will not give the full simulation proofs below, as they follow more or less straightforwardly using known techniques.

On a high level, however, the simulation proceeds as follows: First the simulator simulates the first 4 steps by following the protocol. This is possible as the secret inputs x_i are not used.

If some honest P_j computed $g_j =$ preprocess failure, then the simulator inputs early timeout to $\mathcal{F}_{\text{FSFE}}^f$, which will make it output $y_i = \perp$ to all P_i. Clearly the same happens in the simulated execution since P_j sends $g_j =$ preprocess failure to all honest parties.

If all honest P_j compute $g_j =$ preprocess success, then the preprocessing was secure. This ensures that the sharings $[r_i]$ are consistent, and since the simulator knows the shares of all honest parties, it can compute all r_i. From the s_i broadcast by the corrupted parties in the simulation it computes $x_i = s_i - r_i$ and inputs these to $\mathcal{F}_{\text{FSFE}}^f$ on behalf of the corrupted parties. It broadcasts random s_i's on behalf of honest players.

Then the simulator finishes the execution of the preprocess and input phase. If during this the adversary cheats or calls the timeouts at a time which makes the BA terminate with $q =$ failure, then the simulator inputs early timeout to $\mathcal{F}_{\text{FSFE}}^f$, which will make it output $y_i = \perp$ to all P_i. Clearly the same happens in the simulated execution.

If $q =$ success in the simulation, then the simulator inputs late timeout to $\mathcal{F}_{\text{FSFE}}^f$, and learns the output for corrupted parties. It can now simulate the computation phase using standard techniques until all parties have computed their outputs[2]. Namely, since the computation

[2] In this process, the simulator may need to control the time at which results are delivered to honest parties, depending on when the adversary chooses to deliver the messages in the simulated execution.

phase is a sequence of public reconstructions, the simulator for each reconstruction selects the value to be opened, either a random value or a result y_i, as appropriate. It then computes shares to send on behalf of the honest players such that they are consistent with the opened value and the shares held by corrupted players.

4 Protocol for Preprocessing

In this section, we describe the techniques used in the preprocessing phase. One version of the preprocessing is obtained by simplifying in a straightforward way the protocols from Hirt and Beerliová-Trubíniová in [1], where *hyperinvertible* matrices are used to generate multiplication triples. Another version is based on pseudorandom secret-sharing [7].

4.1 Preprocessing based on Hyperinvertible Matrices

In this subsection we will show how the preprocessing and input phase works. This amounts to showing how to generate the multiplication triples.

The key element in the way we generate triples is that while in [1], a player elimination step is run whenever a fault occurs, we accept the possibility that our protocol will not terminate. Therefore we can simplify and speed up the protocols considerably by cutting away the player elimination and simply aborting if a fault is detected. For completeness and readability, we will describe the most important protocols here, but refer to [1] for security proofs and some of the tools.

In order for us to be able to generate multiplication triples, we first need to be able to generate double sharings of random element – that is, two Shamir sharings of the same random element, possibly with different thresholds. In other words we wish to generate for a random $r \in \mathbb{F}$ sharings $[r]_d$ and $[r]_{d'}$, where d and d' are the degrees or thresholds. A more compact notation for the double sharing is $[r]_{d,d'}$.

We will need some facts from [1] on reconstructing shared values, namely how to reconstruct a value robustly to one player using $\mathcal{O}(nk)$ bits of communication and how to reconstruct up to $T = n - 2t$ values publicly using $\mathcal{O}(n^2 k)$ bits, where k is the size of a field element.

The following is based on the concept of hyperinvertible matrices. "Hyperinvertible" is defined as in [1], where a straightforward way to construct such a matrix is also presented:

Definition 1. *An $m \times n$ matrix M is* hyperinvertible *if for any selection $R \subseteq \{1, \ldots, m\}$ of rows and $C \subseteq \{1, \ldots, n\}$ of columns such that $|R| = |C| > 0$, the square matrix M_C^R consisting of the intersections between rows in R and columns in C is invertible.*

The protocol for generating $T = n - 2t$ double sharings now works as follows (it assumes the existence of an publicly known $n \times n$ hyperinvertible matrix M):

1. Each player P_i Shamir shares a random value s_i to the others using both d and d' as degrees. Every P_i now knows shares of $[s_1]_{d,d'}, \ldots, [s_n]_{d,d'}$, but shares from corrupted players may be incorrect.
2. The players locally compute

$$([r_1]_{d,d'}, \ldots, [r_n]_{d,d'}) = M([s_1]_{d,d'}, \ldots, [s_n]_{d,d'}) \ .$$

   Note that there are actually two vectors here, and the matrix is applied to both, creating two new vectors.

3. All sharings $[s_i]_{d,d'}$ are verified for $i = T+1, \ldots, n$. They are verified by having each P_j send his share of $[s_i]_{d,d'}$ to P_i. Each P_i that is given shares must then check whether they are consistent and that both parts of the double sharing represent the same value. If not, P_i sets an *unhappy* flag to indicate the fault.
4. The double sharings $[r_1]_{d,d'}, \ldots, [r_T]_{d,d'}$ are the output.

The double sharing protocol is guaranteed to either output $T = n - 2t$ correct and random double sharings that are unknown to the adversary or cause at least one honest player to become unhappy. This is proved in [1], along with the fact that the communication complexity is $\mathcal{O}(n^2 k)$ bits. In our case, if an honest player becomes unhappy at any point, all other players are informed and the honest players will abort, as described in the Section 3. That is, we skip the player elimination used in [1].

If we only wanted to generate a set of T single Shamir sharings, it is easy to see that we can use the protocol above but considering only sharings using degree d for each step. The complexity of this is half that of creating double sharings. This is used for generating the sharings $[r_i]$ of a random r_i for each player P_i, that we promised in the Section 3.

Generating Multiplication Triples. Given sharings

$$[a_1]_t, \ldots, [a_T]_t, [b_1]_t, \ldots, [b_T]_t$$

and

$$[r_1]_{t,2t}, \ldots, [r_T]_{t,2t}$$

of random and independent numbers $a_i, b_i, r_i \in \mathbb{F}$, we can generate T multiplication triples as follows:

1. The players compute $[a_i]_t[b_i]_t - [r_i]_{2t} = [a_i b_i - r_i]_{2t}$ for $i = 1, \ldots, T$.[3] They then attempt to publicly reconstruct all of the $a_i b_i - r_i$. If the reconstruction of any of the values fails, an honest player becomes unhappy and we abort.
2. The players locally compute $[a_i b_i]_t = a_i b_i - r_i + [r_i]_t$. All honest players now own shares of the $[a_i b_i]_t$, the $[a_i]_t$ and the $[b_i]_t$ for $i = 1, \ldots, T$.

This protocol is clearly secure, assuming that the sharings we start from have been securely constructed. The simulator would choose random values s_i to play the role of $a_i b_i - r_i$, it would then expand the set of shares known by corrupt players of $[a_i b_i - r_i]$ to a complete set consistent with s_i and use these shares as those sent by honest players. For details, see [1].

The communication complexity is $\mathcal{O}(n^2 k)$ bits for the reconstructions and therefore a total of $\mathcal{O}(n^2 k)$ bits including the generation of the double sharings. That is, we can reconstruct $T = n - 2t = \Theta(n)$ shares with a communication complexity of $\mathcal{O}(n^2 k)$, where k is the bit length of the field elements.

4.2 Preprocessing based on Pseudorandom Secret-Sharing

We show here how to do the preprocessing based on pseudorandom secret-sharing. The techniques used are described in detail in [8], but we present here an overview for completeness.

[3] The notation $[a_i]_t[b_i]_t$ means that each player multiplies its shares of $[a_i]_t$ and $[b_i]_t$. This gives a $2t$ sharing of $a_i b_i$.

Pseudorandom Secret-Sharing. Let A be a set of players of size $n-t$. We can create a random, shared secret by defining for each set A a random value r_A and give it to all players in A. The secret is then given by

$$s = \sum_A r_A \ .$$

Now every maximal unqualified set $\{1,\ldots,n\} \setminus A$ misses exactly one value, namely r_A.

Keeping the above in mind, pseudorandom secret-sharing (PRSS) is then based on the observation that we can create many random shared secrets by distributing once and for all one set of r_A values.

The trick is to use a pseudorandom function ψ_{r_A} with r_A as its key. If the parties agree on some publicly known value a, they can generate the random values they need as $\psi_{r_A}(a)$. So the secret is now

$$s = \sum_A \psi_{r_A}(a) \ .$$

What we actually want, however, is a Shamir sharing. This can be fixed as follows. Define a degree at most t polynomial f_A by $f_A(0) = 1$ and $f_A(i) = 0 \quad \forall i \in \{1,\ldots,n\} \setminus A$. Now each player P_i computes its share

$$s_i = \sum_{\substack{A \subset \{1,\ldots,n\}: \\ |A|=n-t, i \in A}} \psi_{r_A}(a) f_A(i) \ .$$

This is in fact a Shamir sharing of s, since it defines the polynomial

$$f(\mathtt{x}) = \sum_{\substack{A \subset \{1,\ldots,n\}: \\ |A|=n-t}} \psi_{r_A}(a) f_A(\mathtt{x}) \ .$$

It is easy to see that this polynomial has degree at most t and that

$$f(0) = \sum_{\substack{A \subset \{1,\ldots,n\}: \\ |A|=n-t}} \psi_{r_A}(a) = s \ ,$$

which means that it shares the right secret. It is also clear that $s_i = f(i)$, which means that our sharing is a correct Shamir sharing.

Pseudorandom Zero-Sharing. We will need one more tool to be able to generate multiplication triples, and that is what is defined in [8] as pseudorandom zero-sharing (PRZS).

Like PRSS, it is meant to create a Shamir sharing using only local computations, but in this case it is a sharing of 0. We also make it a sharing of degree $2t$ because that is what we need in the following, but the approach works just as well with other thresholds. First define for a set A the set

$$G_A = \{g \in \mathbb{Z}_p[x] \mid \deg(g) \leq 2t \wedge g(0) = 0 \wedge (j \notin A \Rightarrow g(j) = 0)\} \ .$$

This is a subspace of the vector space of polynomials of degree at most $2t$. Because every polynomial in the set has $t+1$ zeros, the subspace must have dimension $2t+1-(t+1) = t$. The construction from [8] needs a basis for this subspace, but no explicit construction was given there. We suggest to use the following:

$$(g_A^1, \ldots, g_A^i, \ldots, g_A^t) = (xf_A, \ldots, x^i f_A, \ldots, x^t f_A),$$

where the f_A is defined as above. It is a basis because it has t elements of G_A which are all of different degrees and therefore linearly independent. Exactly as for PRSS, we assume that we have values r_A known (only) by players in A. Now we define the share at player j as

$$s_j = \sum_{\substack{A \subset \{1,\ldots,n\}: \\ |A|=n-t, j \in A}} \sum_{i=1}^{t} \psi_{r_A}(a,i) g_A^i(j) \ .$$

Note here that the inner sum is a pseudorandom choice of a polynomial from G_A, evaluated in the point j. Now we want to show that this leads to a Shamir sharing of 0, so we define the corresponding polynomial as

$$g_0(\mathtt{x}) = \sum_{\substack{A \subset \{1,\ldots,n\}: \\ |A|=n-t}} \sum_{i=1}^{t} \psi_{r_A}(a,i) g_A^i(\mathtt{x}).$$

The degree of g_0 is clearly at most $2t$, and it is also easy to see that it is consistent with the shares above and that $g_0(0) = 0$.

Making triples using PRSS and PRZS. In order to make multiplication triples, we already know that it is enough if we can build random sharings $[a]_t$, $[b]_t$, and a double sharing $[r]_{t,2t}$.

Using PRSS, it is easy to construct the random degree t sharings. A double sharing can be constructed as follows: Create using PRSS a random sharing $[r]_t$ and use PRZS to create a sharing of zero $[0]_{2t}$. Now

$$[r]_{2t} = [r]_t + [0]_{2t}$$

is clearly a sharing of r of degree $2t$. We can therefore use pseudorandom secret sharing and pseudorandom zero sharing to locally compute all the values needed to make multiplication triples. The only interaction needed is one public opening for each triple as described in Section 4.1.

This is faster than using hyperinvertible matrices for a small number of players, but does not scale well: Since $n - t = \Theta(n)$, the local computation is in exponential in n, as clearly seen from the benchmark results in Section 6. The break-even point between PRSS and hyperinvertible matrices depends both on local computing power and on the cost of communication.

Security of the PRSS approach. We claim that the overall protocol is secure against a computationally bounded and static adversary, when based on PRSS.

To argue this, consider some adversary who corrupts t players, and let A be the set of $n-t$ honest players. Now let π_{random} be the protocol that runs as described above, but where the function ψ_{r_A} is replaced with a truly random function.[4]

When we execute PRSS or PRZS in π_{random}, all secrets and sets of shares held by the honest players are uniformly random, with the only restriction that they are consistent with the shares held by corrupt players. We can therefore use the proof outlined in Section 3.2 to show that π_{random} implements $\mathcal{F}_{\text{FSFE}}^{f}$ (with perfect security).

[4] This can be formalized by assuming an ideal functionality that gives oracle access to the function for the honest players as soon as the adversary has corrupted a set of players initially.

For the rest of the argument, we refer to the protocol using the pseudorandom function as π_{pseudo}. We claim that real-world executions of π_{random} and π_{pseudo} are computationally indistinguishable. Assume for contradiction that there exists some computationally bounded environment $\mathcal{Z}$ that can distinguish between the two with a non-negligible advantage.

From $\mathcal{Z}$ we can now build a new machine $\mathcal{M}$, which gets oracle access to some function f and outputs its guess of whether the function is pseudorandom or truly random.

$\mathcal{M}$ simply runs the protocol with f inserted in the place of ψ_{r_A} (i.e., it runs either π_{random} or π_{pseudo}) for $\mathcal{Z}$. If $\mathcal{Z}$ outputs "π_{random}", M outputs "truly random", otherwise it outputs "pseudorandom". Clearly, M can distinguish between a pseudorandom function and a truly random function with a non-negligible advantage, breaking the security of our PRF.

Combining this with the fact that π_{random} securely realizes $\mathcal{F}$, we see that the same holds for π_{pseudo} (with computational security): The simulator that works for π_{random} also works for π_{pseudo}.

5 VIFF

The Virtual Ideal Functionality Framework, VIFF, is a library with building blocks for developing cryptographic protocols. The goal is to provide a solid basis on which practical applications using MPC can be built. It is also our hope that the framework offered by VIFF will help facilitate rapid prototyping of new protocols by researchers and so lead to more protocols with practical applications.

VIFF aims to be usable by parties connected by real world networks. Such networks are all asynchronous which means that no upper bound can be given on the message delivery time. A well-known example is the Internet where the communication from A to B must go through many hops, each of which introduces an unpredictable delay. Targeting networks with this kind of worst-case behavior from the beginning means that VIFF works well in all environments, including local area networks which behave in a more synchronous manner.

To deal with the asynchronous setting the VIFF runtime system tries to avoid waiting unless it is explicitly asked to do so. In a synchronous setting all parties wait for each other at the end of each round, but VIFF has no notion of "rounds". What determines the order of execution is solely the inherent dependencies in a given program. If two parts of a program have no mutual dependencies, then their relative ordering in the execution is unpredictable. This assumes that the calculations remain secure when executed out-of-order. Protocols written for asynchronous networks naturally enjoy this property since the adversary can delay packets arbitrarily, which makes the reordering done by VIFF a special case.

As an example, consider the simple program in Figure 1 for three players, $n = 3$. It starts by reading some input from the user (an integer) and then defines the field $\mathbb{Z}_{1031}$ where the toy-calculation will take place. All three players then take part in a Shamir sharing of their respective inputs, this results in three Share objects being defined. A fourth Share object is generated using pseudorandom secret sharing [7].

Here all variables represent secret-shared values – VIFF supports Shamir secret sharing for when $n \geq 3$ and additive secret shares for when $n = 2$. The execution of the above calculation is best understood as the evaluation of a tree, please see Figure 2. Arrows denote dependencies between the expressions that result in the calculation of the variable z.

The two variables x and y are independent, and so one cannot reliably say which will be calculated first. But more importantly: We may calculate x and y in *parallel*. It is in fact very important for efficiency reasons that we calculate x and y in parallel. The execution time of a multiparty computation is limited by the speed of the CPUs engaged in the local

```
# (Standard program setup not shown.)

input = int(raw_input("Your input: "))
Zp = GF(1031)
a, b, c = rt.shamir_share([1, 2, 3], Zp, input)
d = rt.prss_share_random(Zp)

x = a * b
y = c * d
z = x + y
```

Fig. 1. Program, rt is a Runtime object.

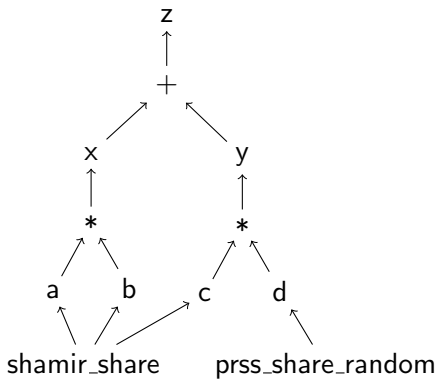

Fig. 2. Expression tree.

computations and by the delays in the network. Network latencies can reach several hundred milliseconds, and will typically dominate the running time. So when we say *parallel* we mean that when the calculation of x waits on network communication from other parties, then it is important that the calculation of y gets a chance to begin its network communication. This puts maximum load on both the CPU and the network.

5.1 Implementing VIFF

VIFF is written in Python, a modern object-oriented procedural language. Programs using VIFF are normal Python programs, and yet we have described how operations may be executed in a random order. This is possible by using a technique where we only work with *deferred* results and never with the results themselves. A deferred result is something that will eventually materialize, but we do not know when and the only available operation is to add *callbacks* to the result. Callbacks are simply function pointers, and each deferred result keeps a list of such pointers. The callbacks will be executed when the result is ready, typically when some share has arrived over the network. This programming style is well-known from graphical user interfaces where the programmer also attaches callbacks to certain events in the user interface. In VIFF this is implemented using the Twisted network library, specifically using the Deferred class provided by Twisted. An example of how Twisted works is this program which retrieves a page from the Internet and prints out the contents:

```
def print_contents(contents):
    print "The Deferred has called us with:"
    print contents

deferred = getPage('http://example.net/')
deferred.addCallback(print_contents)
```

The getPage function returns a Deferred which will eventually hold the HTML of the fetched page. When it does, it will call its callbacks in sequence. If we had retrieved several pages, and attached different callbacks to each, then the execution of those callbacks would depend on which page arrives first.

VIFF uses the Deferred class extensively. In Figure 2 the variables are Share objects, a subclass of Deferred. Using suitable operator overloading we are able to allow the programmer to do arithmetic with Share objects and so treat them like normal numbers. Key to the

implementation of VIFF is a function gather_shares which takes a list of Share objects as input and returns a new Share. This Share will call its callbacks when all Share objects in the list have called their callbacks. We use this to make Share objects that *wait* on other Share objects. Figure 4 in Appendix A shows the implementation of a standard multiplication protocol in VIFF, and uses gather_shares to make the product wait on the two operands to the multiplication.

The big advantage of this system is that it automatically runs the operations in parallel: The calculations implicitly create the tree shown in Figure 2, and this tree is broken down as fast as possible when operands become ready. There is no predefined execution order in the tree – it is deconstructed from the leaves inwards at the same speed as the needed operands arrive.

Also, by making this automatic scheduling implicit the parallelization is done on all levels: Application programmers might define new convenience functions that use primitives offered by VIFF, and these new functions will also be executed in parallel.

Executing things in this way changes the semantics of a program using VIFF from that of a normal Python program. Each statement is no longer executed when it is encountered, it is merely scheduled for execution and then executed later when the operands are available. The semantics of a program using VIFF is thus more like that of a declarative programming language where you declare your intentions but where the compiler takes care of scheduling the calculations in the optimal order.

6 Benchmark Results

In order to measure the efficiency of our implementation, we have run a number of tests using the techniques described above on a set of computers on a fast local area network. The computers had Intel Celeron CPUs with a clock speed of 2.40 GHz, 1 GiB of RAM and were running Red Hat Enterprise Linux 5.2, Python 2.4.3, and VIFF 0.7.

We ran benchmarks with $n = 4, 7, \ldots, 25$ corresponding to thresholds $t = 1, 2, \ldots, 8$, respectively. In each test we secret-shared 2,000 random 32-bit numbers and multiplied the 1,000 pairs in parallel. The results in Table 1 is the average online time spent per multiplication (columns 2, 3, and 5) and the average offline time spent per multiplication triple (columns 4 and 6).

Table 1. Benchmark results.

(n, t)	Passive	Active PRSS		Active Hyper		Ratio
$(4, 1)$	2 ms	4 ms	5 ms	4 ms	20 ms	2.6
$(7, 2)$	3 ms	6 ms	22 ms	6 ms	42 ms	2.2
$(10, 3)$	4 ms	8 ms	130 ms	8 ms	82 ms	2.0
$(13, 4)$	6 ms	10 ms	893 ms	10 ms	136 ms	1.7
$(16, 5)$	8 ms	—	—	12 ms	208 ms	1.6
$(19, 6)$	10 ms	—	—	14 ms	287 ms	1.5
$(22, 7)$	12 ms	—	—	17 ms	377 ms	1.3
$(25, 8)$	15 ms	—	—	19 ms	501 ms	1.3

Table 1 also includes a column giving the ratio between the online time for the multiplication protocol described here using multiplication triples, and the time for a standard multiplication protocol which is only secure against passive adversaries. The passively secure

multiplication protocol consists of a local multiplication followed by a resharing in which everybody communicates with everybody else. The actively secure multiplication, as described above, consists of local multiplications and two openings, which also involves quadratic communication.

The average online time per multiplication appears to grow linearly in the number of players, both in the case of passive and active adversaries. The total amount of network traffic is quadratic in the number of players (in both protocols), but the work done by each player grows only linearly. Our results therefore suggest that the players are CPU bound instead of being slowed down by the network. In the test setup all 25 machines were located on a fast LAN with ping times of about 0.1 ms, so this is to be expected. We hope to setup a better test environment with a controllable network delay in order to do more realistic testing in the future.

The average time per multiplication triple produced via hyperinvertible matrices grows quadratically, please see Figure 3. Fitting a curve $f(n) = an^2 + bn + c$ gives $a = 0.8$, $b = -1$, $c = 10$ as a best fit, and plotting this curve on top of our data points shows an exact match.

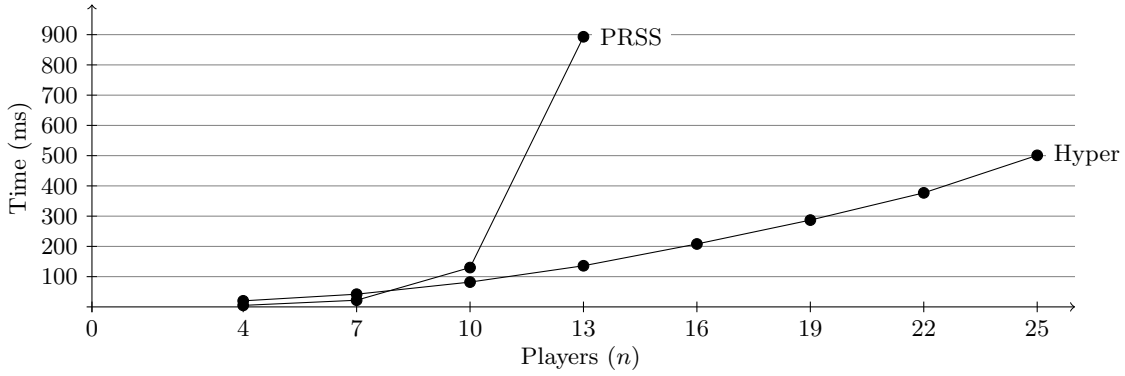

Fig. 3. Preprocessing time per multiplication triple as a function of the number of players.

As expected, the PRSS based preprocessing is faster for a small number of players but does not scale well, and we had to abandon it for $n > 13$. The amount of work per player depends on the number of subsets of size $n-t$ and with $\binom{n}{n-t}$ subsets this gives an exponential growth.

7 Conclusion

We have presented an efficient protocol for general multiparty computation secure against active and adaptive adversaries. The protocol provides a new tradeoff between guaranteeing termination and efficiency which we believe is relevant in practice. To demonstrate this we have implemented the protocol in a framework for secure multiparty computation called VIFF. This allowed us to show that achieving active security costs only about a factor of two in online time, if one is willing to accept that the preprocessing step might fail without revealing any private data. We believe this to be well-suited for practical applications where the parties typically have a much stronger incentive to participate in the computation than to halt it.

Even though the cost of preprocessing is larger than the online cost, it is certainly not prohibitive: For instance, for 4 players, 1000 multiplications can be prepared in 5 seconds.

Currently VIFF supports the fast arithmetic using standard Shamir shares for the case with three or more players, and has support for much slower arithmetic with additive shares in the two player case. Using the additively homomorphic Paillier public key cryptosystem [12], our benchmarks show an average time per multiplication of 300 ms for 32-bit numbers.[5] This is with a straightforward implementation of the cryptosystem in Python and we expect to gain some performance by reimplementing it as a C extension instead.

In the two player case we have $t = n - 1$, also known as *self trust* since every player only need to trust himself to be honest. We would like to develop protocols for $t = n - 1$, but for $n > 2$. Such a high threshold will again require public key cryptography, so we expect this to be expensive, but nevertheless interesting since there might be some situations where the parties are willing to wait longer in return for this level of security.

The VIFF source code is freely available at the VIFF homepage (no link provided due to anonymity) and it is hoped that others can verify our measurements and expand on it with other protocols.

[5] The implementation actually allows multiplication of much larger numbers, up to about 500 bits with a marginal performance penalty.

References

1. Zuzana Beerliová-Trubíniová and Martin Hirt. Perfectly-secure MPC with linear communication complexity. In Ran Canetti, editor, *TCC*, volume 4948 of *Lecture Notes in Computer Science*, pages 213–230. Springer, 2008.
2. Zuzana Beerliová-Trubíniová, Martin Hirt, and Jesper Buus Nielsen. Almost-asynchronous multi-party computation with faulty minority. Manuscript, 2008.
3. Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *STOC*, pages 1–10. ACM, 1988.
4. Peter Bogetoft, Dan Lund Christensen, Ivan Damgard, Martin Geisler, Thomas Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, Michael Schwartzbach, and Tomas Toft. Multiparty computation goes live. Cryptology ePrint Archive, Report 2008/068, 2008. `http://eprint.iacr.org/`.
5. Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, pages 136–145. IEEE, 2001.
6. David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty unconditionally secure protocols. In *STOC*, pages 11–19. ACM, 1988.
7. Ronald Cramer, Ivan Damgård, and Yuval Ishai. Share conversion, pseudorandom secret-sharing and applications to secure computation. In Joe Kilian, editor, *TCC*, volume 3378 of *Lecture Notes in Computer Science*, pages 342–362. Springer, 2005.
8. Ronald Cramer, Ivan Damgård, and Yuval Ishai. Share conversion, pseudorandom secret-sharing and applications to secure computation. In Joe Kilian, editor, *TCC*, volume 3378 of *Lecture Notes in Computer Science*, pages 342–362. Springer, 2005.
9. Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game – a completeness theorem for protocols with honest majority. In *STOC*, pages 218–229. ACM, 1987.
10. Martin Hirt and Ueli M. Maurer. Robustness for free in unconditional multi-party computation. In Joe Kilian, editor, *CRYPTO*, volume 2139 of *Lecture Notes in Computer Science*, pages 101–118. Springer, 2001.
11. Martin Hirt, Jesper Buus Nielsen, and Bartosz Przydatek. Asynchronous multi-party computation with quadratic communication. To appear at ICALP 2008, 2008.
12. Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In Jacques Stern, editor, *EUROCRYPT*, volume 1592 of *Lecture Notes in Computer Science*, pages 223–238. Springer, 1999.
13. Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, November 1979.

A Multiplication in VIFF

As an example of real VIFF code, we have included the implementation of the standard multiplication protocol which is secure against passive adversaries, please see Figure 4.

```
def mul(self, share_a, share_b):
    assert isinstance(share_a, Share) or isinstance(share_b, Share), \
        "Either share_a or share_b must be a Share."

    if not isinstance(share_a, Share):
        # Then share_b must be a Share => local multiplication. We
        # clone first to avoid changing share_b.
        result = share_b.clone()
        result.addCallback(lambda b: share_a * b)
        return result
    if not isinstance(share_b, Share):
        # Likewise when share_b is a constant.
        result = share_a.clone()
        result.addCallback(lambda a: a * share_b)
        return result

    # At this point both share_a and share_b must be Share objects. We
    # wait on them, multiply, reshare, and recombine.
    result = gather_shares([share_a, share_b])
    result.addCallback(lambda (a, b): a * b)
    self.schedule_callback(result, self._shamir_share)
    self.schedule_callback(result, self._recombine, threshold=2*self.threshold)
    return result
```

Fig. 4. The standard multiplication protocol for passive adversaries.

The code handles both local multiplication and multiplication involving network traffic. First, if either share_a or share_b is a not a Share object, i.e., one of them is a constant integer or a FieldElement, then we do a quick local multiplication. Assume that share_a is the constant and share_b is the Share (lines 5–10). We cannot simply multiply share_a and share_b since share_b is a Deferred and might not have a value yet. The solution is to clone share_b and add a callback to it. This callback is simply a lambda expression (an anonymous function) that takes care of the correct multiplication when share_b eventually gets a value (line 9). The opposite case is handled in the same way (lines 11–15).

If it is established that both share_a and share_b are Share objects we create a new Share which waits on both of them (line 19). We then add several callbacks: First we multiply, then we reshare, and finally we recombine. These three operations will be executed in sequence when both share_a and share_b have received their values due to incoming network traffic. The last two callbacks involve network traffic, and must be added using a more expensive mechanism which ensures that everybody agree on the labels put on the data as it is sent over the network.

In all three cases the mul method returns result to the caller (lines 10, 15, or 23). Note that result probably does not have a value at this point, but result is a Share that we have prepared in such a way that it *will* receive the correct value at some point in the future. All VIFF methods work like this: They return Share objects which will eventually get the correct value when other Share objects arrive over the network.