Implementing Asynchronous Multi-Party Computation

PhD Progress Report

Martin Geisler

January 2008

Department of Computer Science
University of Aarhus
Denmark

Contents

1 Introduction

More than 25 years ago Yao [40] defined the problem of secure two-party computation. In 1987 Goldreich et al. [19] extended the problem to multi-party computation and showed that all functions can be evaluated securely and correctly by a group with honest majority in the presence of a passive adversary. Today the area of secure multi-party computation is a very active research area.

This PhD progress report will look at both the theory behind the protocols and at the issues that arise when one wants to implement the protocols efficiently on real life communication networks.

1.1 Overview

We begin in Section 2 by looking at an efficient secure comparison protocol. This material is from the paper "Homomorphic Encryption and Secure Comparison" [12] which is joint work with Ivan Damgård and Mikkel Krøigaard. The protocol is motivated by the increasingly important scenario of on-line auctions in which a number of bidders compete against each other. In order to avoid being on-line during the entire auction (which may run for several days) the bidders submit an encrypted maximum bid to the auction. These encrypted bids are compared with a public current price, which is incremented in rounds until only one winning bid is larger.

The comparison protocol is based on a new efficient homomorphic encryption scheme which can also used as the basis of efficient and general secure multiparty computation. We show how our comparison protocol can be used to improve security of on-line auctions, and demonstrate that it is efficient enough to be used in practice. For comparison of 16 bit numbers with security based on 1024 bit RSA (executed by two parties), our implementation takes 0.28 seconds including all computation and communication.

Section 3 presents material not previously published. It describes the implementation and security properties of a system called the Virtual Ideal Functionality Framework (VIFF). Using VIFF, one can easily program general secure multi-party computations in an asynchronous setting. The protocol implemented by VIFF is proven secure in the Universally Composable (UC) security framework by Canetti [7].

While there has been a large amount of theoretical work done on cryptographic protocols and secure multi-party computation, little effort has been spent trying to implement the protocols. The protocols were viewed as impractical for real tasks. But as more and more efficient protocols are discovered and faster processors and networks are developed, it becomes feasible to implement multi-party computations for problems other than simple toy examples. The double auction run by the SIMAP[1] project is the first example of a multi-party computation used in real life.

Section 4 describes the plans for future work both on cryptographic protocols in general and on VIFF in particular. VIFF is still in its initial development iteration and there are many planned improvements which could be implemented given sufficient time. The report concludes with Section 5, which sums up the contributions presented.

[1] SIMAP is described in Section 3.1.2.

1.2 Acknowledgments

The work presented here was not done alone. Section 2 on secure comparison is joint work with Ivan Damgård and Mikkel Krøigaard. Tomas Toft, Rune Thorbek, and Thomas Mølhave helped with good suggestions and comments.

Ivan Damgård is thanked for good guidance but especially for letting me try out my ideas, which resulted in VIFF. Tomas Toft is thanked for many fruitful discussions on the architecture and help with the implementation. Tord Reistad provided a computer for testing VIFF. Mikkel Krøigård and Rune Thorbek are also thanked for proof reading and for many discussions on the UC framework together with Gert Mikkelsen.

Finally, I would like to thank my wife Stephanie for always providing me with encouragement and support for my work.

2 Homomorphic Encryption and Secure Comparison

The results in this section were presented in [12], which itself is an extended version of a paper presented at the ACISP 2007 conference [11].

2.1 Introduction

We define secure comparison of integers as the following problem: Two or more players are given integers n_A, n_B, where one or both are private, i.e., not known to all players. We then want to decide whether $n_A \geq n_B$, while making sure that the comparison result is the only new information that becomes known. Many variants of this problem exist, depending on whether n_A, n_B are known to particular players, or unknown to everyone. It may even be the case that the comparison result is not supposed to be public, but is to be produced in encrypted form, for instance.

Secure comparison is a special case of general secure computation where all players hold private inputs and want to compute some agreed function of these inputs. Comparison protocols are very important ingredients in many potential applications of secure computation. Examples of this include auctions, benchmarking, and secure extraction of statistical data from databases.

As a concrete example to illustrate the application of our results, we take a closer look at on-line auctions: Many on-line auction systems offer as a service to their customers that one can submit a maximum bid to the system. It is then not necessary to be on-line permanently, the system will automatically bid for you, until you win the auction or your specified maximum is exceeded. We assume in the following what we believe is a realistic scenario, namely that the auction system needs to handle bidders that bid on-line manually, as well as others that use the option of submitting a maximum bid.

Clearly, such a maximum bid is confidential information: Both the auction company and other participants in the auction have an interest in knowing such maximum bids in advance, and could exploit such knowledge to their advantage: The auction company could force higher prices (what is known as "shill bidding") and thereby increase its income and other bidders might learn how valuable a given item is to others and change their strategy accordingly.

In a situation where anyone can place a bid by just connecting to a web site, the security one can obtain by storing the maximum bids with a single trusted party is questionable, in particular if that trusted party is the auction company. Indeed, there are cases known from real auctions where an auction company has been accused of misusing its knowledge of maximum bids [34].

An obvious solution is to share the responsibility of storing the critical data among several parties, and do the required operations via secure computation. One can then make sure that, unless all parties are corrupted, every time the bid goes up, it will become known whether a given player is still in the game because his maximum is larger then the current price, but the actual value of the maximum will remain secret. To keep the communication pattern simple and to minimize problems with maintenance and other logistical problems, it seems better to keep the number of involved players small. We therefore consider the following model:

An input client C supplies an ℓ bit integer m as private input to the computation, which is done by players A and B. Because of our motivating scenario, we require that the input is supplied by sending one message to A, respectively to B, and no further interaction with C is necessary. One may, for instance, think of A as the auction house and B as an accounting company. We will also refer to these as the *server* and *assisting server*.

An integer x (which we think of as the currently highest bid) is public input. As public output, we want to compute one bit that is 1 if $m > x$ and 0 otherwise, i.e., the output tells us if C is still in the game and wants to raise the bid, say by some fixed amount agreed in advance. Of course, we want to do the computation securely so that neither A nor B learns any information on m other than the comparison result.

We will assume that players are honest but curious. We believe this is quite a reasonable assumption in our scenario: C may submit incorrectly formed input, but since the protocol handles even malformed input deterministically, he cannot gain anything from this: any malformed bid will determine a number x_0 such that when the current price reaches x_0, the protocol output will cause C to leave the game. So this is equivalent to submitting x_0 in correct format. Moreover, the actions of A and B can be checked after the auction is over – if C notices that incorrect decisions were taken, he can prove that his bid was not correctly handled. Such "public disgrace" is likely to be enough to discourage cheating in our scenario. In the original paper [12] we sketch how to obtain active security at moderate extra cost.

2.1.1 Our Contribution

We first propose a new homomorphic cryptosystem that is well suited for our application, this is the topic of Section 2.2. The cryptosystem is much more efficient than, e.g., the encryption scheme by Paillier [29] in terms of en- and decryption time. The efficiency is obtained partly by using a variant of Groth's idea of exploiting subgroups of $\mathbb{Z}_n^*$ for an RSA modulus n [20], and partly by aiming for a rather small plaintext space, of size $\Theta(\ell)$.

In Section 2.3 we propose a comparison protocol in our model described above, based on additive secret sharing and homomorphic encryption. The protocol is a new variant of an idea originating in a paper by Blake and Kolesnikov [3]. Their original idea was also based on homomorphic encryption but required

a plaintext space of size exponential in ℓ. Here, we present a new technique allowing us to make do with a smaller plaintext space. This means that the exponentiations we do will be with smaller exponents and this improves efficiency. Also, we save computing time by using additive secret sharing as much as possible instead of homomorphic encryption.

As mentioned, our encryption is based on a k bit RSA modulus. In addition there is an "information theoretic" security parameter t involved which is approximately the logarithm of the size of the subgroup of $\mathbb{Z}_n^*$ we use. Here, t needs to be large enough so that exhaustive search for the order of the subgroup and other generic attacks are not feasible. Section 2.4 contains more information about the security of the protocol.

In the protocol, C sends a single message to A and another to B, both of size $\mathcal{O}(\ell \log \ell + k)$ bits. To do the comparison, there is one message from A to B and one from B to A. The size of each of these messages is $\mathcal{O}(\ell k)$ bits. As for computational complexity, both A and B need to do $\mathcal{O}(\ell(t + \log \ell))$ multiplications mod n. Realistic values of the parameters might be $k = 1024$, $t = 160$, and $\ell = 16$. In this case, counting the actual number of multiplications works out to roughly 7 full scale exponentiations mod n, and takes 0.28 seconds in our implementation, including all computation and communication time. Moreover, most of the work can be done as preprocessing. Using this possibility in the concrete case above, the on-line work for B is about 0.6 exponentiations for A and 0.06 for B, so that we can expect to save a factor of at least 10 compared to the basic version. It is clear that the on-line performance of such a protocol is extremely important: Auctions often run up a certain deadline, and bidders in practice sometimes play a strategy where they suddenly submit a much larger bid just before the deadline in the hope of taking other bidders by surprise. In such a scenario, one cannot wait a long time for a comparison protocol to finish.

We emphasize that, while it may seem easier to do secure comparison when one of the input numbers is public, we do this variant only because it comes up naturally in our example scenario. In fact, it is straightforward to modify our protocol to fit related scenarios. For instance, the case where A has a private integer a, B has a private integer b and we want to compare a and b, can be handled with essentially the same cost as in our model. Moreover, at the expense of a factor about 2 in the round, communication and computational complexities, our protocol generalizes to handle comparison of two integers that are shared between A and B, i.e., are unknown to both of them. It is also possible to keep the comparison result secret, i.e., produce it in encrypted form. The details on that and other extensions can be found in the full version [12].

Finally, in Section 2.5 we describe our implementation and the results of a benchmark between our proposed protocol and the one from [16].

2.1.2 Related Work

There is a very large amount of work on secure auctions, which we do not attempt to survey here, as our main concern is secure protocols for comparison, and the on-line auction is mainly a motivating scenario. One may of course do secure comparison of integers using generic multiparty computation techniques. For the two-party case, the most efficient generic solution is based on Yao-garbled circuits, which were proposed for use in auctions by Naor et al. [26]. Such methods are typically less efficient than ad hoc methods for comparison –

although the difference is not very large when considering passive security. For instance, the Yao garbled circuit method requires – in addition to garbling the circuit – that we do an oblivious transfer of a secret key for every bit position of the numbers to compare. This last part is already comparable to the cost of the best known ad hoc methods.

There are several existing ad hoc techniques for comparison, we already mentioned the one from [3] above, a later variant appeared in [4], allowing comparison of two numbers that are unknown to the parties. A completely different technique was proposed earlier by Fischlin [16].

It should be noted that previous protocols typically are for the model where A and B want to compare two private numbers. Our model is a bit different, as we have one public number that is to be compared to a number that should be known to neither party, and so has to be shared between them. However, the distinction is not very important: Previous protocols can quite easily be transformed to our model, and as mentioned above, our protocol can also handle the other models at marginal extra cost. Therefore the comparison of our solution to previous work can safely ignore the choice of model.

Fischlin's protocol is based on the well-known idea of encrypting bits as quadratic residues and non-residues modulo an RSA modulus, and essentially simulates a Boolean formula that computes the result of the comparison. Compared to [3, 4], this saves computing time, since creating such an encryption is much faster than creating a Paillier encryption. However, in order to handle the non-linear operations required in the formula, Fischlin extends the encryption of each bit into a sequence of λ numbers, where λ is a parameter controlling the probability that the protocol returns an incorrect answer. Since these encryptions have to be communicated, we get a communication complexity of $\Omega(\lambda\ell k)$ bits. Choosing λ such that $5\ell \cdot 2^{-\lambda}$ is an acceptable (small enough) error probability makes the communication complexity significantly larger than the $\mathcal{O}(\ell k)$ bits one gets in our protocol and the one from [4].

The computational complexity for Fischlin's protocol is $\mathcal{O}(\ell\lambda)$ modular multiplications, which for typical parameter values is much smaller than that of [3, 4], namely $\mathcal{O}(\ell k)$ multiplications.[2] Fischlin's result is not directly comparable to ours, since our parameter t is of a different nature than Fischlin's λ: t controls the probability that the best known generic attack breaks our encryption scheme, while λ controls the probability that the protocol gives incorrect results. When the parameters are chosen to make the two probabilities be roughly equal, then the two computational complexities are asymptotically the same.

Thus, in a nutshell, [3, 4] has small communication and large computational complexity while [16] is the other way around. In comparison, our contribution allows us to get "the best of both worlds". In Section 2.5.3 we give results of a comparison between implementations of our own and Fischlin's protocols. Finally, note that our protocol always computes the correct result, whereas Fischlin's has a small error probability.

In concurrent independent work, Garay et al. [17] propose protocols for comparison based on homomorphic encryption that are somewhat related to ours, although they focus on the model where the comparison result is to remain secret. They present a logarithmic round protocol based on emulating a new

[2] In [3, 4] the emphasis is on using the comparison to transfer a piece of data, conditioned on the result of the comparison. For this application, their solution has advantages over Fischlin's, even though the comparison itself is slower.

Boolean circuit for comparison, and they also have a constant round solution. In comparison, we do not consider the possibility of saving computation and communication in return for a larger number of rounds. On the other hand, their constant round solution is based directly on Blake and Kolesnikov's method, i.e., they do not have our optimization that allows us to make do with a smaller plaintext space for the encryption scheme, which means that our constant round protocol is more efficient.

2.2 Homomorphic Encryption

For our protocol we need a semantically secure and additively homomorphic cryptosystem which we will now describe.

To generate keys, we take as input parameters k, t, and ℓ, where $k > t > \ell$. We first generate a k bit RSA modulus $n = pq$ for primes p, q. This should be done in such a way that there exists another pair of primes u, v, both of which should divide $p-1$ and $q-1$. We will later be doing additions of small numbers in $\mathbb{Z}_u$ where we want to avoid reductions modulo u, but for efficiency we want u to be as small as possible. For these reasons we choose u as the minimal prime greater than $\ell + 2$. The only condition on v is that it is a random t bit prime.

Finally, we choose random elements $g, h \in \mathbb{Z}_n^*$ such that the multiplicative order of h is v modulo p and q, and g has order uv. The public key is now $pk = (n, g, h, u)$ and the secret key is $sk = (p, q, v)$. The plaintext space is $\mathbb{Z}_u$, while the ciphertext space is $\mathbb{Z}_n^*$. To encrypt $m \in \mathbb{Z}_u$, we choose r as a random $2t$ bit integer, and let the ciphertext be

$$E_{pk}(m, r) = g^m h^r \bmod n.$$

We note that by choosing r as a much larger number than v, we make sure that h^r will be statistically indistinguishable from a uniformly random element in the group generated by h. The owner of the secret key (who knows v) can do it more efficiently by using a random $r \in \mathbb{Z}_v$.

For decryption of a ciphertext c, it turns out that for our main protocol, we will only need to decide whether c encrypts 0 or not. This is easy, since $c^v \bmod n = 1$ if and only if c encrypts 0. This follows from the fact that v is the order of h, uv is the order of g, and $m < u$. If the party doing the decryption has also stored the factors of n, one can optimize this by instead checking whether $c^v \bmod p = 1$, which will save a factor of 3–4 in practice.

It is also possible to do a "real" decryption by noting that

$$E_{pk}(m, r)^v = (g^v)^m \bmod n.$$

Clearly, g^v has order u, so there is a 1–1 correspondence between values of m and values of $(g^v)^m \bmod n$. Since u is very small, one can simply build a table containing values of $(g^v)^m \bmod n$ and corresponding values of m.

To evaluate the security, there are various attacks to consider: factoring n will be sufficient to break the scheme, so we must assume factoring is hard. Also note that it does not seem easy to compute elements with orders such as g, h unless you know the factors of n, so we implicitly assume here that knowledge of g, h does not help to factor. Note that it is very important that g, h both have the same order modulo both p and q. If g had order uv modulo p but was 1 modulo q, then g would have the correct order modulo n, but $\gcd(g-1, n)$ would

immediately give a factor of n. One may also search for the secret key v, and so t needs to be large enough so that exhaustive search for v is not feasible. A more efficient generic attack (which is the best we know of) is to compute $h^R \bmod n$ for many large and random values of R. By the "birthday paradox", we are likely to find values R, R' where $h^R = h^{R'} \bmod n$ after about $2^{t/2}$ attempts. In this case v divides $R - R'$, so generating a few of these differences and computing the greatest common divisor will produce v. Thus, we need to choose t such that $2^{t/2}$ exponentiations is infeasible.

To say something more precise about the required assumption, let G be the group generated by g, and H the group generated by h. We have $H \leq G$ and that a random encryption is indistinguishable from a uniformly random element in G. The assumption underlying security is now

Conjecture 2.1 *For any constant ℓ and for appropriate choice of t as a function of the security parameter k, the tuple (n, g, h, u, x) is computationally indistinguishable from (n, g, h, u, y), where n, g, h, u are generated by the key generation algorithm sketched above, x is uniform in G and y is uniform in H.*

Proposition 2.2 *Under the above conjecture, the cryptosystem is semantically secure.*

Proof: Consider any polynomial time adversary who after seeing the public key, chooses a message m and gets an encryption of m, which is of the form $g^m h^r \bmod n$, where g has order uv and h has order v modulo p and q. The conjecture now states that even given the public key, the adversary cannot distinguish between a uniformly random element from H and one from G. But h^r was already statistically indistinguishable from a random element in H, and so it must also be computationally indistinguishable from a random element in G. But this means that the adversary cannot distinguish the entire encryption from a random element of G, and this is equivalent to semantic security – recall that one of the equivalent definitions of semantic security requires that encryptions of m be computationally indistinguishable from random encryptions.

The only reason we set t to be a function of k is that the standard definition of semantic security talks about what happens asymptotically when a *single* security parameter goes to infinity. From the known attacks sketched above, we can choose t much smaller than k. Realistic values might be $k = 1024, t = 160$.

A central property of the encryption scheme is that it is homomorphic over u:

$$E_{pk}(m, r) \cdot E_{pk}(m', r') \bmod n = E_{pk}(m + m' \bmod u, r + r').$$

The cryptosystem is related to that of Groth [20], in fact ciphertexts in his system also have the form $g^m h^r \bmod n$. The difference lies in the way n, g and h are chosen. In particular, our idea of letting h, g have the same order modulo p and q allows us to improve efficiency by using subgroups of $\mathbb{Z}_n^*$ that are even smaller than those from [20].

2.3 Comparison Protocol

For the protocol, we assume that A has generated a key pair $sk = (p, q, v)$ and $pk = (n, u, g, h)$ for the homomorphic cryptosystem we described previously. The protocol proceeds in two phases: An input sharing phase in which the client

must be on-line, and a computation phase where the server and assisting server determine the result while the client is offline. See Figure 2.1 for an overview.

In the input sharing phase C secret shares his input m between A and B:

- Let the binary representation of m be $m_\ell \dots m_1$, where m_1 is the least significant bit. C chooses, for $i = 1, \dots, \ell$, random pairs $a_i, b_i \in \mathbb{Z}_u$ subject to $m_i = a_i + b_i \bmod u$.
- C sends privately $a_1, \dots, a_\ell$ to A and $b_1, \dots, b_\ell$ to B. This can be done using any secure public-key cryptosystem with security parameter k, and requires communicating $\mathcal{O}(\ell \log u + k)$ bits.[3] In practice, a standard TLS[4] connection would probably be used.

In the second phase we wish to determine the result $m > x$ where x is the current public price (with binary representation $x_\ell \dots x_1$).

Assuming a value $y \in \mathbb{Z}_u$ has been shared additively between A and B, as C did it in the first phase, we write $[y]$ for the pairs of shares involved, so $[y]$ stands for "a sharing of" y. Since the secret sharing scheme is linear over $\mathbb{Z}_u$, A and B can compute from $[y]$, $[w]$ and a publically known value α a sharing $[y + \alpha w \bmod u]$. Note that this does not require interaction but merely local computation. The protocol proceeds as follows:

- A and B compute, for $i = 1, \dots, \ell$ sharings $[w_i]$ where

$$w_i = m_i + x_i - 2x_i m_i = m_i \oplus x_i.$$

- A and B now compute, for $i = 1, \dots, \ell$ sharings $[c_i]$ where

$$c_i = x_i - m_i + 1 + \sum_{j=i+1}^{\ell} w_j.$$

  Note that if $m > x$, then there is exactly one position i where $c_i = 0$, otherwise no such position exists. Note also, that by the choice of u, it can be seen that no reductions modulo u take place in the above computations.

- Let α_i and β_i be the shares of c_i that A and B have now locally computed. A computes encryptions $E_{pk}(\alpha_i, r_i)$ and sends them all to B.
- B chooses at random $s_i \in \mathbb{Z}_u^*$ and s_i' as a $2t$ bit integer and computes a random encryption of the form

$$\gamma_i = \left(E_{pk}(\alpha_i, r_i) \cdot g^{\beta_i}\right)^{s_i} \cdot h^{s_i'} \bmod n.$$

  Note that, if $c_i = 0$, this will be an essentially random encryption of 0, otherwise it is an essentially random encryption of a random nonzero value. B sends these encryptions to A in randomly permuted order.

- A uses his secret key to decide, as described in the previous section, whether any of the received encryptions contain 0. If this is the case, he outputs "$m > x$", otherwise he outputs "$m \leq x$".

[3] We need to send $\ell \log u$ bits, and public-key systems typically have $\Theta(k)$-bit plaintexts and ciphertexts.

[4] TLS (Transport Layer Security [14]) is the successor to SSL (Secure Sockets Layer).

A note on preprocessing: One can observe that the protocol frequently instructs players to compute a number of form $h^r \bmod n$ where r is randomly chosen in some range, typically $[0 \dots 2^{2t}[$. Since these numbers do not depend on the input, they can be precomputed and stored. As mentioned in the Introduction, this has a major effect on performance because all other exponentiations are done with very small exponents.

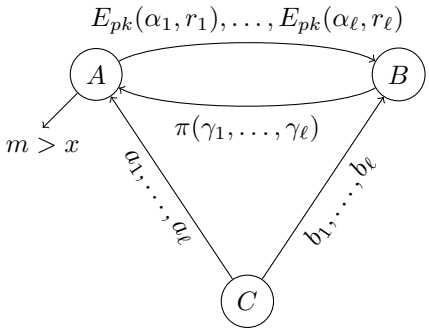


Figure 2.1: Our proposed protocol with both phases illustrated. In the first phase C sends shares to A and B. The second phase consists of a message from A to B and a reply, which A can decrypt to learn the result of the computation.

2.4 Security

In this section the protocol is proven secure against an honest but curious adversary corrupting a single player at the start of the protocol.

The client C has as input its maximum bid m and all players have as input the public bid x. The output given to A is the evaluation of $m > x$, and B and C get no output.

In the following we argue correctness and we argue privacy using a simulation argument. This immediately implies that our protocol is secure in Canetti's model for secure function evaluation [6] against a static and passive adversary.

2.4.1 Correctness

The protocol must terminate with the correct result: $m > x \iff \exists i : c_i = 0$. This follows easily by noting that both $x_i - m_i + 1$ and w_i is nonnegative so

$$c_i = 0 \iff x_i - m_i + 1 + \sum_{j=i+1}^{\ell} w_j = 0$$

$$\iff x_i - m_i + 1 = 0 \wedge \sum_{j=i+1}^{\ell} w_j = 0.$$

We can now conclude correctness of the protocol since $x_i - m_i + 1 = 0 \iff m_i > x_i$ and $\sum_{j=i+1}^{\ell} w_j = 0 \iff \forall j > i : m_j = x_j$, which together imply $m > x$. Note that since the sum of the w_j is positive after the first position in which $x_i \neq m_i$, there can be at most one zero among the c_i.

2.4.2 Privacy

Privacy in our setting means that A learns only the result of the comparison, and B learns nothing new. We can ignore the client as it has the only secret input and already knows the result based on its input.

First assume that A is corrupt, i.e, that A tries to deduce information about the maximum bid based on the messages it sees. From the client, A sees both his own shares $a_1, \ldots, a_\ell$, and the ones for B encrypted under some semantically secure cryptosystem, e.g., TLS. From B, A sees the message:

$$\big(E_{pk}(\alpha_i, r_i) \cdot g^{\beta_i}\big)^{s_i} \cdot h^{s'_i} \bmod n.$$

By the homomorphic properties of our cryptosystem this can be rewritten as

$$E_{pk}(s_i \cdot \alpha_i, s_i \cdot r_i) \cdot E_{pk}(s_i \cdot \beta_i, s'_i) = E_{pk}\big(s_i(\alpha_i + \beta_i), s_i \cdot r_i + s'_i\big).$$

In order to prove that A learns no additional information, we can show that A could – given knowledge of the result, the publically known number and nothing else – simulate the messages it would receive in a real run of the protocol.

The message received and seen from the client can trivially be simulated as it consists simply of ℓ random numbers modulo u and ℓ encrypted shares. The cryptosystem used for these messages is semantically secure, so the encrypted shares for B can be simulated with encryptions of random numbers.

To simulate the messages received from B, we use our knowledge of the result of the comparison. If the result is "$m > x$", we can construct the second message as $\ell - 1$ encryptions of a nonzero element of $\mathbb{Z}_u^*$ and one encryption of zero in a random place in the sequence. If the result is "$m \leq x$", we instead construct ℓ encryptions of nonzero elements in $\mathbb{Z}_u^*$.

If we look at the encryptions that B would send in a real run of the protocol, we see that the plaintexts are of form $(\alpha_i + \beta_i)s_i \bmod u$. Since s_i is uniformly chosen, these values are random in Z_u if $\alpha_i + \beta_i \neq 0$ and 0 otherwise. Thus the plaintexts are distributed identically to what was simulated above. Furthermore, the ciphertexts are formed by multiplying $g^{(\alpha_i+\beta_i)s_i}$ by

$$h^{s_i r_i + s'_i} = h^{s_i r_i} h^{s'_i}.$$

But h has order v which is t bits long, and therefore taking h to the power of the random $2t$ bit number s'_i will produce something which is statistically indistinguishable from the uniform distribution on the subgroup generated by h. But since $h^{s_i r_i} \in \langle h \rangle$, the product will indistinguishable from the uniform distribution on $\langle h \rangle$. So the s'_i effectively mask out $s_i r_i$ and makes the distribution of the encryption statistically indistinguishable from a random encryption of $(\alpha_i + \beta_i)s_i$. Therefore, the simulation is statistically indistinguishable from the real protocol messages.

The analysis for the case where B is corrupt is similar. Again we will prove that we can simulate the messages of the protocol. The shares received from the client and the encryptions seen are again simply ℓ random numbers modulo u and ℓ random encryptions and are therefore easy to simulate. Also, B receives the following from A:

$$E_{pk}(\alpha_i, r_i).$$

But since the cryptosystem is semantically secure, we can make our own random encryptions instead and their distribution will be computationally indistinguishable from the one we would get by running the protocol normally.

2.5 Complexity and Performance

In this section we measure the performance of our solution through practical tests. The protocol by Fischlin [16] provides a general solution to comparing two secret integers using fewer multiplications than the other known general solutions. We show that in the special case where one integer is publically known and the other is additively shared between two parties, our solution provides for faster comparisons than our adaptation of [16].

2.5.1 Setup and Parameters

As described above, our special case consists of a server, an assisting server and a client. The client must be able to send his value and go offline, whereafter the other two parties should be able to do the computations together. In our protocol the client simply sends additive shares to each of the servers and goes offline. However, the protocol by Fischlin needs to be adapted to this scenario before we can make any reasonable comparisons. A very simple way to mimic the additive sharing is for the client to simply send his secret key used for the encoding of his value to the server while sending the actual encoding to the assisting server. Clearly the computations can now be done by the server and assisting server alone, where the server plays the role of the client. The modified protocol is shown in Figure 2.2.

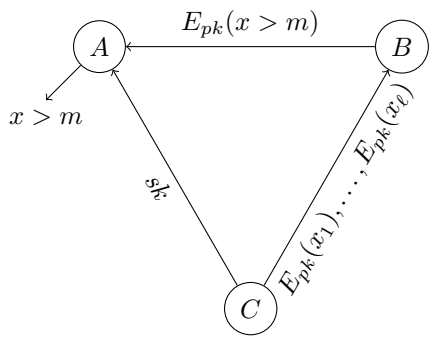


Figure 2.2: The modified Fischlin protocol. The client C can go offline after having sent the key to A and the encryptions to B. From that point the protocol proceeds as in [16].

Together, the key and encoding determine the client's secret value, but the key or the encoding alone do not. The key of course reveals no information about the value. Because of semantic security, the encryption alone does not reveal the secret to a computationally bounded adversary.

Another issue is to how to compare the two protocols in a fair way. Naturally, we want to choose the parameters such that the two protocols offer the same security level, but it is not clear what this should mean – some of the parameters in the protocols control events of very different nature. Below, we describe the choices we have made and the consequences of making different choices.

Both protocols use an RSA modulus for their encryption schemes, and it is certainly reasonable to use the same bit length of the modulus in both cases, say 1024 bits. Our encryption scheme also needs a parameter t which we propose to choose as $t = 160$. This is because the best known attack tries to have random results of exponentiations collide in the subgroup with about 2^{160} elements. Assuming the adversary cannot do much more than 2^{40} exponentiations, the collision probability is roughly $2^{2 \cdot 40}/2^{160} = 2^{-80}$.

We do not have this kind of attack against Fischlin, but we do have an error probability of $5\ell \cdot 2^{-\lambda}$ per comparison. If we choose the rationale that the probability of "something going wrong" should be the same in both protocols, we should choose λ such that Fischlin's protocol has an error probability of 2^{-80}. An easy computation shows that for $\ell = 16$, $\lambda = 86$ gives us the desired error probability, and it follows that $\lambda = 87$ works for $\ell = 32$.

We have chosen the parameter values as described above for our implementation, but it is also possible to argue for different choices. One could argue, for instance, that breaking our scheme should be as hard as factoring the (1024 bit) modulus using the best known algorithm, even when the generic attack is used. Based on this, t should probably be around 200. One could also argue that having one comparison fail is not as devastating as having the cryptosystem broken, so that one could perhaps live with a smaller value of λ than what we chose. Fischlin mentions an error probability of 2^{-40} as being acceptable. These questions are very subjective, but fortunately, the complexities of the protocols are linear in t and λ, so it is easy to predict how modified values would affect the performance data we give below. Since we find that our protocol is about 10 times faster, it remains competitive even with $t = 200, \lambda = 40$.

2.5.2 Implementation

To evaluate the performance of our proposed protocol we implemented it along with the modified version of the protocol by Fischlin [16] described above. The implementation was done in Java 1.5 using the standard BigInteger class for the algebraic calculations and Socket and ServerSocket classes for TCP communication. The result is two sets of programs, each containing a server, an assisting server, and a client. Both implementations weigh in at about 1,300 lines of code. We have naturally tried our best to give equal attention to optimizations in the two implementations.

We tested the implementations using keys of different sizes (k in the range of 512–2048 bits) and different parameters for the plaintext space ($\ell = 16$ and $\ell = 32$). We fixed the security parameters to $t = 160$ and $\lambda = 86$ which, as noted above, should give a comparable level of security.

The tests were conducted on six otherwise idle machines, each equipped with two 3 GHz Intel Xeon CPUs and 1 GiB of RAM. The machines were connected by a high-speed LAN. In a real application the parties would not be located on the same LAN – for credibility the server and assisting server would have to be placed in different locations and under the control of different organizations (e.g., the auction house and the accountant), and the client would connect via a typical Internet connection with a limited upstream bandwidth. Since the client is only involved in the initial sharing of his input, this should not pose a big problem – the majority of network traffic and computations are done between the server and assisting server, who, presumably, have better Internet connections and considerable computing power.

The time complexity is linear in ℓ, so using 16 bit numbers instead of 32 bit numbers cuts the times in half. In many scenarios one will find 16 bit to be enough, considering that most auctions have a minimum required increment for each bid, meaning that the entire range is never used. As an example, eBay require a minimum increment which grows with the size of the maximum bid meaning that there can only be about 450 different bids on items selling for

less than $5,000 [15]. The eBay system solves ties by extra small increments, but even when one accounts for them one sees that the 65,536 different prices offered by a 16 bit integer would be enough for the vast majority of cases.

2.5.3 Benchmark Results

The results of the benchmarks can be found in Table 2.1 with a graph in Figure 2.3. From the table and the graph it is clear to see that our protocol has performed significantly faster in the tests than the modified Fischlin protocol. The results also substantiate our claim that the time taken by an operation is proportional to the size of ℓ and that we do indeed roughly halve the time taken by reducing the size of ℓ from 32 to 16 bits.

k	DGK_{16}	F_{16}	DGK_{32}	F_{32}
512	82	844	193	1,743
768	168	1,563	331	3,113
1024	280	2,535	544	5,032
1536	564	4,978	1,134	10,135
2048	969	8,238	1,977	16,500

Table 2.1: Benchmark results. The first column denotes the key size k, the following columns have the average time to a comparison. The average was taken over 500 rounds, after an initial warm-up phase of 10 rounds. All times are in milliseconds. The abbreviation "DGK" refers to our protocol and "F" refers to the modified Fischlin protocol. The subscripts refer to the ℓ parameter used in the timings.

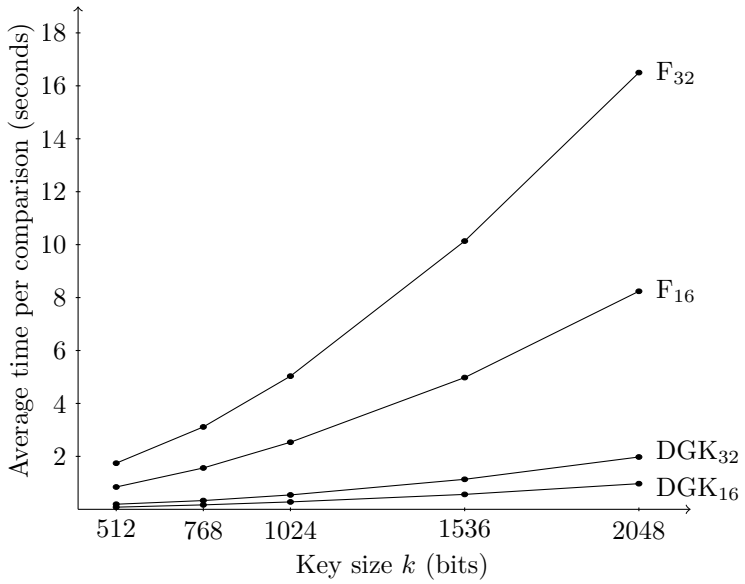


Figure 2.3: Graph of the data from Table 2.1.

We should note that these results are from a fairly straight-forward implementation of both protocols and that further optimizations might be possible.

3 Efficient Asynchronous Multi-Party Computation

The benchmarks described in the previous section were made with a special purpose one-off implementation. We will now describe a framework called VIFF [18] which allows general multi-party computations. VIFF is short for *Virtual Ideal Functionality Framework* and the goal is to create a library of building blocks that make implementing secure multi-party computations easy. In the UC framework [7] a multi-party computation will implement an ideal functionality, and one can consider the VIFF implementation to be a *virtual* ideal functionality. The VIFF homepage is `http://viff.dk/`, from where the VIFF source code can be downloaded.

3.1 Introduction

VIFF was created in the spring of 2007 as a test-bed for alternative MPC implementations and to show that it is possible to make a simple and light-weight framework for MPC. The ideas in VIFF has since shown themselves to be solid and VIFF has grown from being a toy to a full and efficient MPC framework.

VIFF is implemented in the Python programming language developed by van Rossum et al. [39] in the early 1990s. Python is a general purpose programming language with support for object-oriented programming. Anonymous and higher-order functions make some functional programming possible too.

The choice of Python was largely driven by the desire to have a flexible language for rapid prototyping and by the need for a good library for asynchronous network communication. Like many languages, Python comes with a standard library that gives access to sockets for doing network communication. Twisted [23] is a Python framework that abstracts the low-level socket communication away and allows the programmer to easily build efficient network applications with asynchronous communication.

3.1.1 Our Contribution

The main contribution of VIFF is an automatic parallel scheduling of operations. Operations are executed as soon as their operands are ready *without* waiting for the other parties. Intuitively this is secure since although an adversary might see things in a different order than normal, he still sees exactly the same shares as if we had waited for the other parties. We formalize and prove this in the UC framework.

Choosing this model of execution brings two important benefits:

- Efficient use of network resources. Network latency is the dominant factor in our benchmarks and by scheduling many operations in parallel the average cost can be greatly reduced.
- Modular design. Each operation in VIFF is independent of the others which makes it easy to implement new and faster operations.

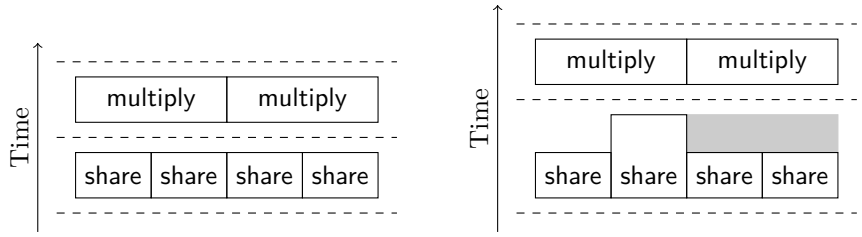
VIFF is still a young project with a code base of only 2,500 lines. The design is simple on purpose to keep it transparent with only the needed abstractions. This should make VIFF readily adaptable to new requirements.

3.1.2 Related Work

The MPC operations in VIFF are inspired by the commands of the $\mathcal{F}_{\mathbf{MPC}}$ ideal functionality by Toft [37]. The greater_than comparison protocol is from [36].

VIFF is a spin-off project from the Secure Information Management and Processing (SIMAP) project, which in turn is a successor to the Secure Computing Economy and Trust (SCET) project, both at the University of Aarhus. The SCET project set out to implement a platform for multi-party computations with a focus on economic applications such as auctions and benchmarking. The project implemented a prototype of a secure double auction [5]. The SIMAP project is more general and will be designing a dedicated programming language in which high-level protocol descriptions can be specified [27].

The biggest difference between VIFF and SIMAP is that VIFF automatically makes operations run in parallel, whereas this must be done explicitly with the SIMAP runtime. This makes VIFF much simpler since one does not need to specify how a new primitive interacts with every other primitive. Also, the automatic parallelism can potentially yield a faster execution since it will adapt better to changing network conditions: With a static schedule based on rounds, the execution stalls if a round takes longer than expected. VIFF would begin executing the next available operation immediately, see Figure 3.1.



(a) An optimal execution in which all four sharings proceed quickly and both rounds are fast.

(b) A slow execution where time is wasted (gray) when the second multiplication waits for the unrelated sharing.

Figure 3.1: An execution where fixed communication rounds leads to wasted time. The dashed lines represent the synchronization done between each round and multiple operations in the same line represent parallel execution.

Another example of a cryptographically aware compiler is CAO by Page [28]. This project is more low-level than VIFF and SIMAP and is focusing on aspects such as how to efficiently implement cryptographic primitives without being prone to timing attacks at the CPU level. No thought has been put into these issues in VIFF. It might be possible to use CAO as a back-end for VIFF, providing both more speed and better security than plain Python.

VIFF currently only works for computations where an honest majority can be found – that means the smallest number of parties is three. The Fairplay system by Malkhi et al. [24] is an example of a system for only two parties. In VIFF you specify the desired computation as a normal Python program, but in Fairplay you write the function to be evaluated in a special purpose language. A compiler transforms the function into an optimized Boolean circuit. The circuit is evaluated using the technique described by Yao [41]. Currently Fairplay does secure function evaluation only whereas VIFF allows reactive programs with multiple rounds of interaction between the parties.

3.2 The Universally Composable Security Framework

Proving that a cryptographic protocol is secure is a hard problem. First we must define rigorously what "secure" means, and then we must prove that the protocol lives up to the definition. The Universally Composable (UC) security framework by Canetti [7] helps solve these problems. We will now give a brief overview of the UC framework and describe how it relates to the work on VIFF.

3.2.1 Security Definition

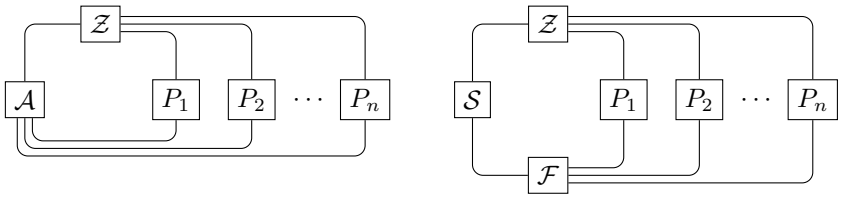
We begin with an informal statement of what it means to be "secure":

Definition 3.1 A protocol is *secure* if an outside observer cannot distinguish between an execution of the real protocol and a secure replacement protocol.

Even though this definition is very short, it turns out to be useful. It gives a testable condition of when a protocol is secure: You take a protocol which is known to be secure and prove that the new protocol is indistinguishable from the secure protocol. To bootstrap this process we use a trick and compare our protocol with a protocol which we *define* to be secure. This ideal protocol uses an *ideal functionality* to do the computation. This is an interactive Turing machine, $\mathcal{F}$, which cannot be corrupted and always calculates the correct result. The ideal protocol is simple: Everybody starts by handing their inputs over to $\mathcal{F}$, which spends some time calculating and finally gives everybody their correct result. Clearly no information is leaked and it makes good sense to define this as "secure". Since the parties do nothing in the ideal protocol we will call them *dummy parties*. To provide maximum generality, we will allow the observer to specify the inputs to the dummy parties. We will call the observer the *environment* and denote it by $\mathcal{Z}$ from now on to match the standard UC terminology.

The real protocol is not so simple and contains actual parties which follow some protocol π without having an ideal functionality to help them. An additional entity is the *adversary*, $\mathcal{A}$. By default the adversary listens to all communication between the parties and observes the internal state of corrupted players (a passive adversary) but may also be allowed to take full control over a party and change messages arbitrarily (an active adversary). The adversary can talk with the environment both to provide details of what it sees in the protocol, but also to receive instructions on what to do next. Figure 3.2a shows this situation with no corruptions. Note that the parties cannot communicate directly with each other – all communication between parties must pass through the adversary. As a worst case assumption, $\mathcal{A}$ is allowed to delay and reorder the delivery of the messages sent between the parties in the protocol arbitrarily, even when passive. We only require that messages are eventually delivered – this models an asynchronously network with a reliable transport layer.

The goal of $\mathcal{Z}$ is to distinguish the situation in Figure 3.2a from the ideal protocol execution. Since $\mathcal{Z}$ expects to talk to the adversary, we include an extra party called a simulator, $\mathcal{S}$, in the ideal protocol experiment shown in Figure 3.2b. The job of the simulator is to pretend to be the adversary towards $\mathcal{Z}$. To do this the simulator gets help from $\mathcal{F}$ – the precise amount of help allowed is part of the description of $\mathcal{F}$. Since the behavior of $\mathcal{F}$ is our definition of security, we do not want this help to leak private information, so $\mathcal{F}$ will typically send messages of the form $\langle x := ? \rangle$ which only tells $\mathcal{S}$ that x now has a value, but does not tell which value.



(a) Real world. **(b)** Ideal world.

Figure 3.2: The two central protocol experiments in the UC framework. Note how the two figures looks exactly the same towards $\mathcal{Z}$ who, in both cases, interacts with $n + 1$ parties over a network.

As mentioned above, the adversary has the power to corrupt some of the parties. We will allow up to t corruptions, where $n = 2t + 1$. We restrict $\mathcal{A}$ to be static and passive. Static means that the adversary chooses a fixed subset of the parties to corrupt at the beginning of the protocol. The corrupted parties do not take part in the protocol, instead $\mathcal{A}$ sends messages on their behalf. Passive means that $\mathcal{A}$ still follows the protocol – it is honest-but-curious.

Having described the two protocol experiments and their participants, we now define a random variable $\text{EXEC}_{\mathcal{F},\mathcal{S},\mathcal{Z}}$ to be the output of $\mathcal{Z}$ in the ideal world execution with the ideal functionality $\mathcal{F}$ and simulator $\mathcal{S}$. Define $\text{EXEC}_{\pi,\mathcal{A},\mathcal{Z}}$ to be the output of $\mathcal{Z}$ in the real world execution of π under attack by $\mathcal{A}$. Let k be the *security parameter*. We can then refine Definition 3.1 slightly as follows:

Definition 3.2 A protocol π is *secure* with regard to an ideal functionality $\mathcal{F}$ if for any adversary $\mathcal{A}$ there exists a simulator $\mathcal{S}$ such that for any environment $\mathcal{Z}$ the statistical difference between $\text{EXEC}_{\mathcal{F},\mathcal{S},\mathcal{Z}}$ and $\text{EXEC}_{\pi,\mathcal{A},\mathcal{Z}}$ is negligible in k.

A more formal definition can be found in the full UC paper [7], but this definition should capture the gist of what it means to be UC-secure. Let us just here remark that if the environment has unbounded computational power, one talks about *perfect* security when the two distributions are equal, and *statistical* security otherwise. If $\mathcal{Z}$ is limited to polynomial computations one obtains *computational* security. In this report we will only deal with polynomial time interactive Turing machines and thus aim for computational security.

3.2.2 Simulation

To prove that a protocol is secure, we must describe how the simulator fools the environment. It must produce output distributed in the same way as $\mathcal{A}$ would in the real world, otherwise $\mathcal{Z}$ would notice and the protocol would not be secure. Figure 3.3 shows how the simulator for VIFF does this: It runs a copy of $\mathcal{A}$ inside and feeds this copy with inputs as it gets messages from $\mathcal{F}$.

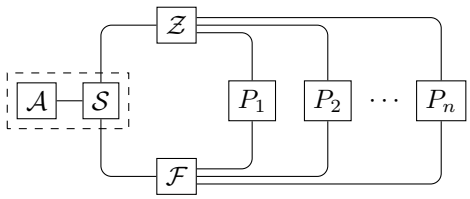


Figure 3.3: The ideal world with the adversary run by the simulator. Giving $\mathcal{A}$ inputs distributed as in the real world ensures that the output of $\mathcal{A}$ is correctly distributed in the ideal world.

3.2.3 Composability

An important benefit of the UC framework is that UC-secure protocols can be composed in arbitrary ways to build larger protocols, which remain UC-secure. Previously one had to prove the security of new protocols from first principles, but with the UC framework one can easily build on top of the work of others. UC functionalities have been defined for standard tasks such as secure message transmission, public-key cryptography, secret sharing, etc.

3.3 Network Assumptions

The classic results on secure multi-party computation are made under the assumption of a synchronous setting where the communication takes place on a network with known packet delays (latency) and where the parties are equipped with clocks with a known maximum drift rate. In this setting it is easy to divide the protocols into logical units called *rounds*. A round begins with the delivery of all messages sent in the previous round. Each party is then asked to specify a number of new messages which will be delivered at the beginning of the next round. It is natural to take the number of rounds needed as a measure of the protocol execution time. The total number of bits transmitted (communication complexity) is often counted as well. The time used for local computation by the parties is assumed to be negligible in comparison to the network delays and is typically not counted.

3.3.1 Modern Asynchronous Communication Networks

The synchronous model does not match communication networks or computers as we know them today. Modern networks are *asynchronous* and computers do normally not have access to precise clocks.

When sending packets over the Internet, the Internet Protocol (IP) [31] has the responsibility of getting the packets to the correct destination. But the IP gives very few guarantees: Intermediate routers might drop packets at any time (due problems like congestion and transmission errors) and packets may be reordered or duplicated. In particular, the IP gives no guarantees about delivery time (if the packet even reaches the destination).

The Transmission Control Protocol (TCP) [32] is normally used to create a virtual connection on top of the connection-less IP network. Because packets can be lost on the IP level, TCP must be prepared to ask for retransmission of data. This means that the delivery can be delayed further. A sender and receiver communicating over TCP are reading and writing a *stream* of bytes – there are no messages at the TCP level. As bytes are written to the stream, TCP will take care of buffering and will send out IP packets as they are filled or when it has been too long since the last packet was sent. Such buffering introduces further unpredictable delays in the protocol.

Normal computers also have no access to a globally correct clock. Computers are typically built with an on-board oscillator used to keep track of the time. Even if initially synchronized, clocks will drift away from each other since frequencies of oscillators vary with temperature. The Network Time Protocol (NTP) is widely used to keep computers synchronized to a standard time [25]. Roughly speaking, this is done by exchanging packets containing timestamps,

from which the network delay can be estimated and the local clock adjusted accordingly. But the NTP server is a trusted third party and we would rather design our protocols without relying on such a service.

3.3.2 Implementing Protocols on Asynchronous Networks

Because of the differences between the synchronous model and what is available for implementation, it is clear that we cannot simply implement the standard protocols in a straightforward way. We have to adapt the situation. There are at first glance two possible approaches

1. Use the synchronous protocol unchanged and build an emulation layer on top of the network to simulate a synchronous network. Awerbuch [1] describes such a simulation where n parties can synchronize by paying an overhead of $\mathcal{O}(n)$ messages per round in the synchronous network. In addition to the communication overhead of synchronization, some parties will sit idle waiting for the other parties when they actually have the data required to start the next round of computation as shown in Figure 3.1.

   A more fundamental problem is the possibility of errors in the form of faulty parties when considering active adversaries. The problem of synchronizing several parties is then equivalent to the Byzantine agreement problem, which is much harder [33, 38].

2. Develop new asynchronous protocols, but keep the network model unchanged. This is the course taken by Ben-Or et al. [2] and recently by Hirt et al. [21] for the case with active adversaries. This is the more difficult approach, but it also has the chance of being the most efficient since the protocols are executed in their native environment.

A third possibility is to "cheat" and run the synchronous protocol on an asynchronous network, but prove that the resulting functionality is secure. This is the approach taken by VIFF. Section 3.5 will describe a real world model that captures this behavior: It is asynchronous and allow data to be delayed in arbitrary (even malicious) ways. The real world model assumes that data eventually arrives and that it arrives intact (the guarantees provided by TCP). Also like TCP, the model does not provide authenticity and confidentiality. Instead standard techniques such as TLS can be used by higher layers.

3.4 Ideal World Execution

We will now describe an ideal functionality that can be implemented on modern communication networks as described in the previous section. We will do this by defining a number of primitive operations necessary to do efficient MPC.

3.4.1 Primitives for Multi-Party Computation

All computations can in principle be reduced to Boolean operations, and for those the NAND and NOR gates are universal. But even though all other logical gates can be constructed from two or more of these universal gates, we seldom want to limit ourselves to just Boolean operations. Instead we consider operations such as addition, multiplication, and comparison primitive since they

can be executed in a single clock cycle on a modern CPU. In fact many such operations can normally be executed at once due to the use of several parallel pipelines in the CPU.

When doing MPC we consider addition and multiplication of values from a finite field as primitive operations. Implementing NAND or NOR on bit-values would in principle have been enough, but evaluating things at such a fine granularity is very expensive, just as it would be expensive if a CPU only did a few NAND operations per clock cycle instead of adding or multiplying whole 32- or 64-bit integers. Input to the computation in the form of sharing and output in the form of opening of sharings are considered primitive too.

We want the functionality to allow several primitive operations to be started at once – not due to CPU pipelines, but due to the inherit delays of network traffic which makes it possible to send out several packets before getting a reply to the first. The programming language defined by the available commands resemble a simple imperative programming language of the same flavor as Pascal and C. The language is *straight-line*, meaning that there are no looping or branching constructs. The meta-program run by the environment can still include branching and looping.

3.4.2 Ideal Protocol

The ideal functionality reacts on input from $\mathcal{Z}$ (sent through the dummy parties) and processes one command at a time – if $\mathcal{Z}$ sends commands too fast they are buffered. Commands are discarded if they are not *valid*, meaning that they must consist of a recognized instruction as well as a *program counter*. This is an opaque tag that must be unique for a given protocol run. The program counters are used by $\mathcal{F}$ to associate each result with a specific command – this is necessary because new commands may be started while waiting for the results of earlier commands.

Each valid command may have some conditions attached to it. If the conditions are not fulfilled, $\mathcal{F}$ buffers the command. If more than one command is eligible for execution, $\mathcal{F}$ will choose the next command to execute at random.

The functionality informs the simulator of all inputs it receives from the dummy parties, including the sending party. Private inputs in the commands are blanked (replaced by ?). This models that values used in computation are secret but the computation trace itself is public. As each command is executed, $\mathcal{Z}$ will expect to see acknowledgments with the correct program counters. We let $\mathcal{S}$ send those through $\mathcal{F}$, who will simply pass on any input it get from $\mathcal{S}$.

The protocol execution terminates when $\mathcal{Z}$ outputs a bit. It is up to $\mathcal{Z}$ to decide (depending on the output it receives from the parties and $\mathcal{A}$) when it terminates the execution.

The available commands and their semantics are:

Assignment Variables are defined by their first assignment. To assign the value v to the variable x the environment must select a single party P_i to be the one who learns the value and send

$$\langle x := v, pc \rangle,$$

to that party. The functionality stores $x \longmapsto v$ in its memory M.

Output Variables can be output to reveal their value to a particular party. When x is defined and $\mathcal{F}$ receives

$$\langle \text{output}, x, P_i, pc \rangle,$$

from all parties it sends $\langle M(x), pc \rangle$ to $\mathcal{S}$.

Linear combination To store a linear combination of previously defined variables $x_1, \ldots, x_j$ with constants $c_1, \ldots, c_j$ in x, $\mathcal{Z}$ sends the command

$$\langle x := c_1 \cdot x_1 + c_2 \cdot x_2 + \cdots + c_j \cdot x_j, pc \rangle$$

to all P_i. This makes $\mathcal{F}$ store the assignment $x \longmapsto \sum_{i=1}^{j} c_i \cdot M(x_i)$ in its memory. Please note that this command covers simple addition of variables when all $c_i = 1$.

Multiplication When $\mathcal{F}$ has received

$$\langle x := y \cdot z, pc \rangle$$

from all P_i where y and z are already defined, $\mathcal{F}$ stores the assignment $x \longmapsto M(y) \cdot M(z)$ in its memory.

Synchronization By sending

$$\langle \text{synchronize}, pc \rangle$$

to all P_i, the environment ask the parties to synchronize. Synchronization is a tool for $\mathcal{Z}$ to structure its meta-program, and $\mathcal{F}$ needs to do nothing here – it is handled entirely by $\mathcal{S}$.

3.5 Real World Execution

We will now describe the concrete protocols which will realize the ideal world protocol just described. This is a mathematical model of the protocols implemented in VIFF.

We assume that the parties communicate using a semantically secure public-key cryptosystem and that the public keys have been distributed securely in advance. In practice, each party could obtain a TLS certificate from a known certificate authority (CA) and announce this certificate to the others before the computation starts. When keys have been distributed the different commands are implemented as follows:

Assignment When $\mathcal{Z}$ sends

$$\langle x := v, pc \rangle,$$

to P_i it secret shares the value v into shares $v_1, \ldots, v_n$ using a Shamir secret sharing [35] with threshold t. The shares are sent securely to the other parties, i.e., P_i sends $\langle E_{pk_j}(v_j), pc \rangle$ to each P_j where $j \neq i$. When all shares has been sent, P_i stores $x \longmapsto v_i$ in M_i and outputs $\langle \text{ok}, pc \rangle$ to $\mathcal{Z}$.

The other parties P_j store $x \longmapsto v_j$ in M_j and output $\langle \text{ok}, pc \rangle$ to $\mathcal{Z}$ when they have received the share from P_i.

Output To open x to P_i the environment sends

$$\langle \text{output}, x, P_i, pc \rangle,$$

to all parties. Party P_j sends its share of x securely to P_i and output $\langle \text{ok}, pc \rangle$ to $\mathcal{Z}$. When receiving $t+1$ shares, P_i will reconstruct the value v stored in x and output $\langle \text{ok}, v, pc \rangle$ to $\mathcal{Z}$.

Linear combination Receiving

$$\langle x := c_1 \cdot x_1 + c_2 \cdot x_2 + \cdots + c_j \cdot x_j, pc \rangle$$

where all x_i are defined will make P_i store $x \longmapsto \sum_{i=1}^{j} c_i \cdot M_i(x_i)$ in its memory and output $\langle \text{ok}, pc \rangle$ to $\mathcal{Z}$.

Multiplication The environment sends

$$\langle x := y \cdot z, pc \rangle,$$

to all parties. When receiving the command and when both shares are defined, each party reacts by multiplying the two shares to get a temporary share x', which will correspond to a Shamir sharing of $y \cdot z$ with threshold $2t$.

This share is reshared by having each P_i run the same commands as if it had received the command $\langle x_i := x', pc_i \rangle$ where pc_i is a fresh program counter derived from pc and i. Each party P_i recombines its shares $x_1, \ldots, x_n$ into its share of x using a threshold of $2t = n-1$. When a party has reconstructed its share of x, it outputs $\langle \text{ok}, pc \rangle$ to $\mathcal{Z}$.

Synchronization When receiving

$$\langle \text{synchronize}, pc \rangle,$$

P_i will send $\langle \text{ready}, pc \rangle$ to all other parties. When P_i has received such ready messages from all parties, it outputs $\langle \text{ok}, pc \rangle$ to $\mathcal{Z}$.

3.6 From Ideal to Real World

We will now on a case by case basis describe how the simulator reacts to messages from the ideal functionality in order to pose as the adversary towards the environment. In each case $\mathcal{S}$ will simulate the messages seen by $\mathcal{A}$ in the real world execution, and simply pass on the output (if any) to $\mathcal{Z}$.

Assignment When $\mathcal{S}$ receives $\langle x := ?, pc \rangle$ from P_i it picks random values $r_1, \ldots, r_n$ and encrypts each value $c_j = E_{pk_j}(r_j)$. The simulator gives these ciphertexts to $\mathcal{A}$ with P_i as the sender and P_j as the receiver. If $\mathcal{A}$ produces an output, this is sent to $\mathcal{Z}$. The simulator sends $\langle \text{ok}, pc \rangle$ back to P_i (through $\mathcal{F}$).

When $\mathcal{A}$ delivers c_j to P_j, $\mathcal{S}$ sends $\langle \text{ok}, pc \rangle$ to P_j (again through $\mathcal{F}$).

Assuming a semantically secure cryptosystem, the output produced by $\mathcal{A}$ in the ideal world must match the output produced in the real world and $\mathcal{Z}$ is thus unable to distinguish between the two cases.

Output We will start by considering the case where x is opened to an honest party P_i. The simulator sees when each P_j receives the $\langle \text{output}, x, P_i, pc \rangle$ message. The simulator invents a share r_j at random and encrypts it to get $c_j = E_{pk_j}(r_j)$. The simulator gives c_j to $\mathcal{A}$ on behalf of P_j for delivery to P_i. Any output by $\mathcal{A}$ is sent to $\mathcal{Z}$.

The encryptions received by $\mathcal{A}$ in the real world contain Shamir shares, which (when looking at up to t shares) are uniformly random numbers. The simulator can therefore simulate this perfectly towards the adversary by encrypting random numbers. When $\mathcal{S}$ sees that $\mathcal{A}$ has delivered all shares to P_i, it sends $\langle \text{ok}, v, pc \rangle$ to P_i ($\mathcal{S}$ was told the correct value v by $\mathcal{F}$).

If the party P_i is corrupt, $\mathcal{S}$ must work a little harder. In that case $\mathcal{A}$ knows (by previous simulation) the share v_i belonging to P_i. This share was chosen at random by $\mathcal{S}$ without knowing v. But because $\mathcal{A}$ can only corrupt up to t players, there will be at least one share which is unknown to $\mathcal{A}$ but needed for reconstructing v. With the knowledge of v and the shares possessed by $\mathcal{A}$, $\mathcal{S}$ chooses consistent shares for the honest parties and sends them securely to $\mathcal{A}$. By sending the shares as the $\langle \text{output}, x, P_i, pc \rangle$ arrive, the simulator ensures that $\mathcal{A}$ sees the same arrival order as in the real world and which makes the view of the adversary indistinguishable from the real world view.

Linear combination In the real world no communication is done, and the adversary sees no messages from the parties. The simulator should therefore do nothing in the ideal world.

Multiplication In the real world $\mathcal{A}$ sees the communication produced by the resharing. Those messages are produced by the result of the parties executing the same steps as if they had received $\langle x_i := x', pc_i \rangle$ and it can be simulated in the same way as a normal assignment.

Synchronization When synchronizing, the only communication produced is the $\langle \text{ready}, pc \rangle$ messages sent by each to the other parties and the final $\langle \text{ok}, pc \rangle$ message sent to $\mathcal{Z}$ when a party hears that all other parties are ready.

To simulate the messages sent in response to a $\langle \text{synchronize}, pc \rangle$, the simulator sends a $\langle \text{ready}, pc \rangle$ message to $\mathcal{A}$ from P_i when it learns that P_i has received $\langle \text{synchronize}, pc \rangle$ from $\mathcal{Z}$.

When $\mathcal{A}$ has delivered $\langle \text{ready}, pc \rangle$ message from all other parties to P_i, then $\mathcal{S}$ sends $\langle \text{ok}, pc \rangle$ to P_i. This is a perfect simulation of the real world.

The correctness of this simulation was argued above and it is also clear that the simulator is efficient. The simulator uses $\mathcal{A}$ as a black-box and thus works for any $\mathcal{A}$. It produces a view for $\mathcal{Z}$ that is identical in the two worlds, except for the possibility that the semantically secure cryptosystem breaks. We conclude that the real world protocol is secure with regard to $\mathcal{F}$.

Having a mathematical model for a secure cryptographic protocol that can be used for multi-party computations is only the first step towards making useful computations. Implementing the model is the next step and will be described in the following.

3.7 From VIFF to the Real World Model

VIFF offers a set of commands closely resembling the MPC commands described in Section 3.5. For the description of the VIFF commands, let rt be an instance of the Runtime class and that x, y, and z are instances of the Share class. We will describe the commands with three parties for concreteness, but this is not a limit of the VIFF runtime.

Shamir sharing The rt.shamir_share method is used for Shamir sharing values over $\mathbb{Z}_p$. The command is symmetric in the sense that all parties execute it, but with different input values. The result is therefore not just one share, but a tuple with one share for each party.

Consider three parties P_1, P_2, and P_3 who all execute the line

```
x, y, z = rt.shamir_share(v_i)
```

each with its own input value in place of v_i. This corresponds directly to the environment sending

$$\langle x := v_1, pc_1 \rangle, \qquad \langle y := v_2, pc_2 \rangle, \qquad \langle z := v_3, pc_3 \rangle$$

to P_1, P_2, and P_3, respectively.

Opening Executing the command

```
open_x = rt.open(x)
```

will open x to all parties. This corresponds to the environment sending

$$\langle \text{output}, x, P_i, pc_i \rangle$$

to all parties $P_1, \ldots, P_n$ and for all i – all parties broadcast their share to all other parties. Currently, the rt.open command is symmetric and values are always opened to all parties, but this will be changed in the future.

Linear combination Forming a linear combination of shares x, y, and z using coefficients a, b, and c can be done by executing

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

and this maps directly to $\mathcal{Z}$ sending the command

$$\langle w := a \cdot x + b \cdot y + c \cdot z, pc \rangle$$

to all parties. This involves no communication between the parties.

Multiplication Shares can be multiplied by executing

```
z = x * y
```

and this maps directly to $\mathcal{Z}$ sending the command

$$\langle z := x \cdot y, pc \rangle$$

to all parties. As in the real world model, this involves a resharing.

Synchronization Executing a function f after synchronizing is done by

```
sync = rt.synchronize()
sync.addCallback(f)
```

This corresponds to the environment sending

$$\langle \text{synchronize}, pc \rangle$$

to all parties and then executing f when all parties are ready.

In addition to these primitive commands VIFF provides a number of higher-level commands. They use the primitives described above and are thus secure since the primitives themselves are secure. The commands are:

Exclusive-or If x and y are bit values, the exclusive-or can be calculated by

```
z = rt.xor_int(x, y)
```

This simply calculates $z = x + y - 2 \cdot x \cdot y$.

The runtime has another method, rt.xor_bit which is used when the shares represent values from GF(2^8) in which exclusive-or is simply addition and thus can be made with no communication at all.

Pseudo-random secret sharing The runtime can create a secret sharing of a uniformly pseudo-random number using no communication by the technique of pseudo-random secret sharing described by Cramer et al. [9]. The parties simply execute

```
rand = rt.prss_share_random(field)
```

The field variable indicates the field, either $\mathbb{Z}_p$ or GF(2^8). The parties can share a particular value by doing

```
x, y, z = rt.prss_share(v_i)
```

each with their own value for v_i. The sharing costs a broadcast which needs not be encrypted. Pseudo-random secret sharing has no corresponding command in the real world model.

Comparison The comparison protocol by Toft [36] is used by executing

```
bit = rt.greater_than(x, y)
```

This makes bit a GF(2^8) share of 0 (if $x \leq y$) or of 1 (if $x > y$). The greater_thanII method gives a result secret shared in $\mathbb{Z}_p$. The protocol is faster for large bit lengths (unpublished work by Tomas Toft).

The comparison protocols correspond to a series of primitive commands in the real world model, which means that they are secure because the primitives are secure under arbitrary composition.

3.8 Performance Results

The initial aim of VIFF was to create a simple and flexible system for implementing secure multi-party computation. The automatic parallel scheduling of primitive operations should improve performance since VIFF will start on the next available operation even before the results have arrived from the previous.

We have benchmarked VIFF in an attempt to verify that the scheduling done by VIFF really does pay off. The benchmark was run over the Internet using full TLS encryption. Computers located in three countries were used: thyra02 at the University of Aarhus, Denmark, serengeti12 at the Norwegian University of Science and Technology (Trondheim, Norway) and bazooka located in Los

Country	Hostname	CPU Type	CPU Speed	RAM
Denmark	thyra02	Intel Xeon	3.06 GHz	2 GiB
Norway	serengeti12	Intel Pentium 4	2.60 GHz	756 MiB
USA	bazooka	AMD Opteron	2.20 GHz	4 GiB

Table 3.1: Specification of the machines used in the VIFF benchmark.

Angeles, USA. The last machine is the webserver hosting the VIFF homepage, and is thus shared with other users. At the time of the benchmarking, the load on bazooka was around 5–10, but would occasionally increase to 15 or more.[5] The specifications of these computers can be found in Table 3.1. The code for the benchmarks is revision 8803c64a055d which will be released as VIFF version 0.4.

When evaluating cryptographic protocols, the number of multiplications is a common measure. It is therefore interesting to see how fast secret shared values can be multiplied in VIFF. By default many multiplications will be started before the first finishes (parallel execution) and Figure 3.4 shows the timing results when executing more and more multiplications. In all graphs we have marked the results from bazooka to show where the data points are. As expected, the time grows linearly in the number of multiplications and takes about 1.3 ms per multiplication for large number of multiplications.

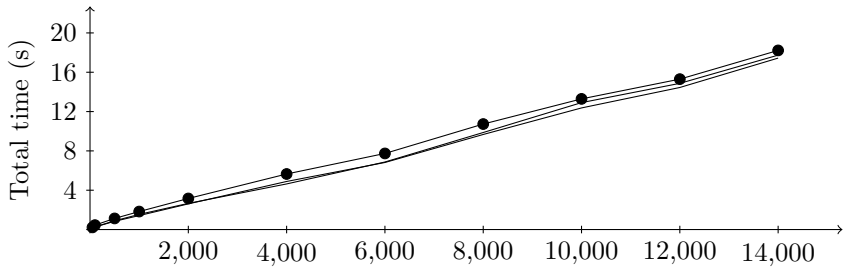


Figure 3.4: Parallel multiplication benchmark results.

To check that the automatic parallel scheduling works, another benchmark run was made, but this time the next multiplication was only scheduled when the current multiplication was complete. This forces VIFF to execute the multiplications in rounds and thwarts the normal eager scheduling. Figure 3.5 shows the benchmark results when multiplying up to 1,000 pairs of numbers in this way. We see that the average time per multiplication has risen to about 185 ms in comparison to about 1.3 ms with parallel execution. The time taken for a serial multiplication matches nicely with the observed round-trip time from thyra02 and serengeti12 to bazooka. The standard assumption is that time spent on local computations (CPU time) is negligible and these benchmark results show that the time spent on multiplying and on TLS encryption is a very low overhead compared to the time it takes for a simple ping packet to move back and forth between the hosts.

[5]The "load" on a UNIX system is defined as the average number of processes waiting to access the CPU in a given interval. The load is typically given for the last 1, 5, and 15 minutes.

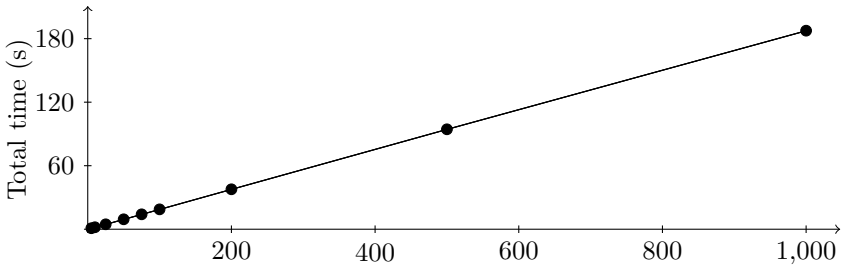


Figure 3.5: Serial multiplication benchmark results.

The graph in Figure 3.4 shows a linear function between the number of multiplications and the total total time used. Figure 3.6 shows a graph of the time per multiplication as a function of the number of multiplications. We see that the average time per multiplication drops rapidly at first until it stabilizes at around 1.3 ms. This shows that the graph in Figure 3.4 is only linear at a large scale. An explanation for this could be that the machines sit idle for a brief period when doing only a few multiplications. When the clock is started the local multiplications are done quickly and the shares are sent out over the network. The clock is only stopped when the resharing is done, i.e., when shares have been received from all other parties. When more multiplications are done, this final waiting period becomes proportionally smaller in comparison to the total time used, and the average time per multiplication drops.

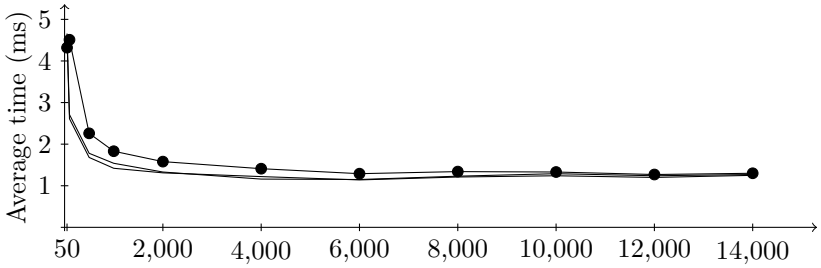


Figure 3.6: Parallel multiplication benchmark results.

We can analyze the graph in Figure 3.5 in a similar way to get Figure 3.7. There we see that the time per multiplication is around 187 ms and that it is much more constant. The curves for the two machines thyra02 and serengeti12 both start out lower, but they increase to match the time on bazooka. No good explanation has been found for why the two machines are able to finish the first 25 multiplications faster than bazooka.

In addition to multiplication, we have also benchmarked comparisons. The comparison protocol is by Toft [36] and is implemented in the greater_than method in VIFF. Comparisons are much slower than multiplications, so only a few benchmark runs were made and the results are shown in Figure 3.8. We see that the average time falls to around 800 ms when more comparisons are made, just as for multiplications.

There has unfortunately not been time to run benchmarks for the improved comparison protocol implemented by Tomas Toft. The other methods in the

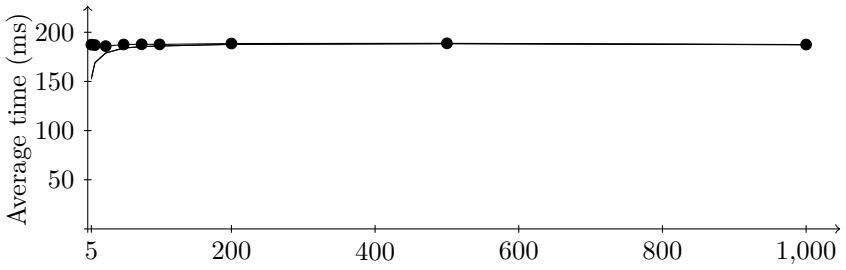
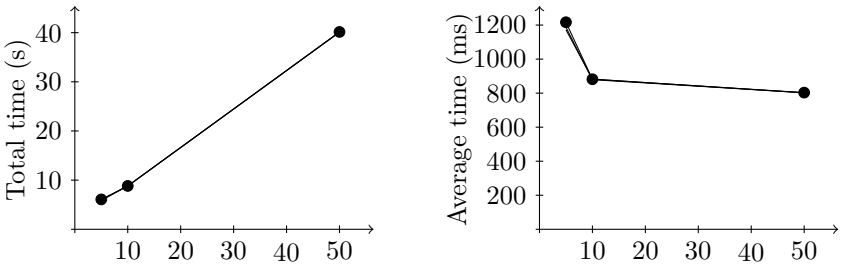


Figure 3.7: Serial multiplication benchmark results.



(a) Total time. **(b)** Time per comparison.

Figure 3.8: Parallel comparison benchmark results.

VIFF runtime (pseudo-random secret sharing, exclusive-or, bit conversion, etc.) should also be benchmarked. All the benchmarks presented here were done using a 65 bit prime, but benchmarks using other primes should be conducted as well to see how VIFF scales.

4 Future Work

There is still work left to be done. SIMAP and VIFF will continue for at least another year and we will now describe some of the planned futures.

4.1 SIMAP

In the near future it would be natural to extend the SIMAP runtime to handle active adversaries. There known protocols for this, although they are slower than in the passive case, it would be interesting to see *how* much slower they are in real life.

Another goal of the SIMAP project is to investigate the cost of protocols with full security threshold, i.e., where $t = n - 1$. In such a situation we say that we have *self-trust* since each party need only trust himself to be honest. This is clearly needed in two-party protocols, but more parties might also want this kind of security. It is known that such protocols cannot be achieved with information theoretical security[6] – we must use cryptographic primitives which

[6]Two players cannot even securely compute the logical AND of two bits, which renders general multi-party computations impossible as well [8].

rely on a computational hardness assumption. The right homomorphic encryption scheme might be the solution. As in Section 2.2, a homomorphic encryption scheme allows you to either add or multiply values while they are encrypted. The hard part is to find a homomorphic encryption scheme where *both* addition and multiplication is easy. This will probably be the focus of some of the future work in the SIMAP project.

The SIMAP runtime currently knows two types of variables: Booleans and integers. Implementing division and floating point arithmetic would be very useful for many applications. One example would be solving linear programs, something which plays an important role in economics.

4.2 VIFF

The current version of VIFF is functional and many interesting problems can be solved with it, but it still lacks many desirable features. We will describe some of them here, but the reader is referred to the TODO file in the VIFF distribution for details.

Like the SIMAP runtime, VIFF should be extended with protocols that are secure against active adversaries. Apart from new protocols, this involves better error-handling at all levels in VIFF since parties may then crash in arbitrary ways in addition to sending bogus values. VIFF should also have protocols that enable it to be used for two party protocols or protocols with full threshold.

Another important feature is pre-processing. Many protocols can be divided into two parts of which only one is dependent of the actual input values. The other part is independent of the inputs to the computation and can thus be executed in advance in a pre-processing step. A typical pre-processing step is the joint generation of random values unknown to all parties. Implementing a general framework in VIFF for saving and loading such pre-processed values would increase the efficiency of some protocols and make benchmarking easier.

Support for accurate and detailed benchmarking would be a valuable tool to compare different protocol implementations and to find bottlenecks in VIFF itself. The profiling should collect data on the amount time spent on communication, local calculations, and pre-processing. There is currently no special support for such instrumentation in VIFF. With such a system in place, it would be possible to run nightly regression tests and immediately see how a code change affects the execution time of different protocols. VIFF has a test suite, but it can only catch regressions in functionality.

Finally, all parts of the VIFF source code should be audited for security problems. One known problem is the generation of pseudo-random values. Presently, they are generated by an ad-hoc procedure where a random seed is hashed repeatedly (using SHA-1) until enough pseudo-random bits have been generated. The security of this procedure is unknown. A replacement algorithm needs to be secure, but also fast since it will be used by essentially all parts of VIFF. A fast stream cipher (or a block cipher running in counter mode) might be good candidates providing both speed and security.

Although there will be no time to do it in the near future, it is interesting to think about how VIFF would look like if it were implemented in a strongly typed programming language. The choice of Python and Twisted has served VIFF well, but using a strong, expressive type system has its benefits as well. Given a type system in which you could distinguish between "n is an integer",

"p is a prime", and "q is a Blum prime" would enable us to write a program in which the requirements of the primitives chosen at the lowest levels would be carried through to the higher levels. A good example is the comparison protocol by Toft [36], which in principle works over any prime field $\mathbb{Z}_p$. But because of how VIFF implements the protocol by Damgård et al. [10] for generating shares of random bits, p must actually be a Blum prime, i.e., $p \equiv 3 \pmod 4$. The problem boils down to how VIFF calculates square roots of field elements. When p is a Blum prime, one can calculate the square root of x by $\sqrt{x} = x^{(p+1)/4}$. This is easy to verify by squaring, which gives

$$(\sqrt{x})^2 = x^{(p+1)/2} = xx^{(p-1)/2} = x(\tfrac{x}{p}).$$

In our case we know that x is a quadratic residue and so the Legendre symbol $(\frac{x}{p}) = 1$. But when p is not a Blum prime, $(p+1)/4$ is not an integer and something different from the square root is calculated. Such subtle errors could have been avoided if the Python type system would allow us to express that square roots can only be calculated when p is a Blum prime.

The functional language Haskell [30] is an example of a strongly typed language with a very expressive type system. Haskell is a compiled language and has strong support for numeric calculations and multi-threaded programs. Much of the VIFF code is currently written in a functional style, and Haskell has been the inspiration for many of the functional constructs in Python. One could therefore hope that a Haskell implementation of VIFF would be even more "natural" than the current implementation in Python. Unfortunately the author does not (yet) have the skills to reimplement VIFF in Haskell, but the language (and other functional languages like it) remains an interesting alternative to Python and to more traditional imperative languages like C++ or Java.

5 Conclusion

This report describes two years of PhD studies. The report started in Section 2 with a theoretic work on a new protocol for comparing a public and a secret integer using only two parties, which among other things has applications in on-line auctions. The comparison uses a new efficient homomorphic encryption scheme. Our benchmark results suggest that our new protocol is highly competitive and reaches an acceptably low time per comparison for real-world application.

A general framework for doing secure multi-party computation was described in Section 3 and its security was proven in the UC framework. VIFF is practical in the sense that it enables rapid prototyping of new protocols for multi-party computation and benchmarking shows that it is efficient enough for real applications running across the Internet.

The VIFF project is now run as an open-source project and the code can be downloaded freely from `http://viff.dk/`. It is the hope of the author that this will help foster new protocols for multi-party computation both within and outside the SIMAP project. The protocols for multi-party computation have been known for many years, but with VIFF researchers and developers finally have access to a freely available working implementation.

References

1. Baruch Awerbuch. Complexity of network synchronization. *Journal of the ACM*, 32(4):804–823, 1985.

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

3. Ian F. Blake and Vladimir Kolesnikov. Strong conditional oblivious transfer and computing on intervals. In Pil Joong Lee, editor, *ASIACRYPT*, volume 3329 of *Lecture Notes in Computer Science*, pages 515–529. Springer, 2004.

4. Ian F. Blake and Vladimir Kolesnikov. Conditional encrypted mapping and comparing encrypted numbers. In Di Crescenzo and Rubin [13], pages 206–220.

5. Peter Bøgetoft, Ivan Damgård, Thomas Jakobsen, Kurt Nielsen, Jakob Pagter, and Tomas Toft. A practical implementation of secure auctions based on multiparty integer computation. In Di Crescenzo and Rubin [13], pages 142–147.

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

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

8. Ronald Cramer and Ivan Damgård. Lecture notes on multiparty computation. Available from `http://www.daimi.au.dk/~ivan/CPT.html`, 2004.

9. Ronald Cramer, Ivan Damgård, and Yuval Ishai. Share conversion, pseudorandom secret-sharing and applications to secure computation. In Kilian [22], pages 342–362.

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

11. Ivan Damgård, Martin Geisler, and Mikkel Krøigaard. Efficient and secure comparison for on-line auctions. In Josef Pieprzyk, Hossein Ghodosi, and Ed Dawson, editors, *ACISP*, volume 4586 of *Lecture Notes in Computer Science*, pages 416–430. Springer, 2007.

12. Ivan Damgård, Martin Geisler, and Mikkel Krøigaard. Homomorphic encryption and secure comparison. *International Journal of Applied Cryptography (to appear)*, 2008.

13. Giovanni Di Crescenzo and Avi Rubin, editors. *Financial Cryptography and Data Security – FC 2006, 10th International Conference, FC 2006 Anguilla, British West Indies, February 27 – March 2, 2006, Revised Selected Papers*, volume 4107 of *Lecture Notes in Computer Science*, 2006. Springer.

14. Tim Dierks and Eric Rescorla, editors. *The Transport Layer Security (TLS) Protocol Version 1.1*, RFC 4346. Internet Engineering Task Force, April 2006. Available on-line: `http://ietf.org/rfc/rfc4346.txt`.

15. eBay Inc. Bid increments. Available on-line: `http://pages.ebay.com/help/buy/bid-increments.html`, October 2006.

16. Marc Fischlin. A cost-effective pay-per-multiplication comparison method for millionaires. In David Naccache, editor, *CT-RSA*, volume 2020 of *Lecture Notes in Computer Science*, pages 457–472. Springer, 2001.

17. Juan Garay, Berry Schoenmakers, and José Villegas. Practical and secure solutions for integer comparison. In Tatsuaki Okamoto and Xiaoyun Wang, editors, *PKC*, volume 4450 of *Lecture Notes in Computer Science*, pages 330–342. Springer, 2007.

18. Martin Geisler. VIFF: Virtual ideal functionality framework. Homepage: `http://viff.dk/`, 2007.

19. 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.

20. Jens Groth. Cryptography in subgroups of $\mathbb{Z}_n$. In Kilian [22], pages 50–65.

21. Martin Hirt, Jesper Buus Nielsen, and Bartosz Przydatek. Cryptographic asynchronous multi-party computation with optimal resilience. In Ronald Cramer, editor, *EUROCRYPT*, volume 3494 of *Lecture Notes in Computer Science*, pages 322–340. Springer, 2005.

22. Joe Kilian, editor. *Theory of Cryptography, Second Theory of Cryptography Conference, TCC 2005, Cambridge, MA, USA, February 10-12, 2005, Proceedings*, volume 3378 of *Lecture Notes in Computer Science*, 2005. Springer.

23. Glyph Lefkowitz, Itamar Shtull-Trauring, et al. Twisted. Release 2.5.0, Twisted Matrix Laboratories, January 2007. Homepage: `http://twistedmatrix.com/`.

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

25. David L. Mills. A brief history of NTP time: Memoirs of an Internet timekeeper. *Computer Communication Review*, 33(2):9–21, 2003.

26. Moni Naor, Benny Pinkas, and Reuban Sumner. Privacy preserving auctions and mechanism design. In *EC*, pages 129–139, New York, 1999. ACM Press.

27. Janus Dam Nielsen and Michael I. Schwartzbach. A domain-specific programming language for secure multiparty computation. In *PLAS*, pages 21–30. ACM, 2007.

28. Dan Page. CAO: A cryptography aware language and compiler. Homepage: `http://www.cs.bris.ac.uk/home/page/research/cao.html`, 2007.

29. 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.

30. Simon L. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003. Available on-line: `http://haskell.org/haskellwiki/Language_and_library_specification`.

31. Jonathan B. Postel, editor. *Internet Protocol*, RFC 791. Internet Engineering Task Force, September 1981. Available on-line: `http://ietf.org/rfc/rfc791.txt`.

32. Jonathan B. Postel, editor. *Transmission Control Protocol*, RFC 793. Internet Engineering Task Force, September 1981. Available on-line: `http://ietf.org/rfc/rfc0793.txt`.

33. Michael O. Rabin. Randomized Byzantine generals. In *FOCS*, pages 403–409. IEEE, 1983.

34. Mette Rode. Debat: Reglerne er strammet op. *Morgenavisen Jyllands-Posten*, 2004.

35. Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, November 1979.

36. Tomas Toft. *Secure Integer Computation with Applications in Economics*. PhD progress report, University of Aarhus, Denmark, June 2005.

37. Tomas Toft. *Primitives and Applications for Multi-party Computation*. PhD thesis, University of Aarhus, Denmark, March 2007.

38. Sam Toueg. Randomized Byzantine agreements. In *PODC*, pages 163–178. ACM, 1984.

39. Guido van Rossum et al. Python. Release 2.5, Python Software Foundation, September 2006. Homepage: `http://python.org/`.

40. Andrew Chi-Chih Yao. Protocols for secure computations. In *FOCS*, pages 160–164. IEEE, 1982.

41. Andrew Chi-Chih Yao. How to generate and exchange secrets. In *FOCS*, pages 162–167. IEEE, 1986.