Efficient Implementation of the Orlandi Protocol Extended Version

Thomas P. Jakobsen[1], Marc X. Makkes[2], and Janus Dam Nielsen[1]

[1] The Alexandra Institute
Aabogade 34, 8200 Aarhus N
Denmark
{thomas.jakobsen, janus.nielsen}@alexandra.dk
[2] Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven
Netherlands
m.x.makkes@student.tue.nl

Abstract. We present an efficient implementation of the Orlandi protocol which is the first implementation of a protocol for multiparty computation on arithmetic circuits, which is secure against up to $n-1$ static, active adversaries. An efficient implementation of an actively secure self-trust protocol enables a number of multiparty computation where one or more of the parties only trust himself. Examples includes auctions, negotiations, and online gaming. The efficiency of the implementation is largely obtained through an efficient implementation of the Paillier cryptosystem, also described in this paper.

Keywords: Secure multiparty computation, MPC, homomorphic encryption, protocols

1 Introduction

Secure multiparty computation is a cryptographic technique allowing n parties to jointly compute the result of a function $f(x_1, x_2, ..., x_n)$ while ensuring that the input x_i of each party P_i is kept private, even with a number t of the parties acting maliciously. The only information that is allowed to be revealed is the result of the function.

In the 80s it was proved that secure multiparty computation could in fact be applied to any computable function, making it an extremely general and useful technique, at least in theory. This was first done by Yao [34] in the restricted case of two parties, but soon followed similar results for the general case of n parties [4, 13]. These results were, however, mostly of theoretical interest due to the complexity of the protocols.

Since then a large number of results have been obtained using different security- and adversary- models, underlying network assumptions, and improvements of previously known results.

In recent years, the theory has advanced enough to allow practical implementations of secure multiparty computation. Examples of practical systems which support evaluation of general multiparty computation are the FairPlay [22], VIFF [32], ShareMind [7], and SIMAP [9] systems. However, many applications are still infeasible in practice, especially those that rely on quick response times like online auctions. Also, in order to be practical, the aforementioned systems tend to either be restricted to a limited number of parties or to loosen up the security model. Some examples of the latter could be assuming that the corrupted parties do not deviate from the protocol (the passive security model) or that at most a certain threshold t of parties gets corrupted (threshold security model). Especially the active security model have until recently been regarded as too complex for practical implementations. However, recently Lindell, Pinkas, and Smart showned that active security in the two-party case is indeed practical [21], and Damgård, Geisler, Krøigaard, and Nielsen showed that active security can be practical if less than $n/3$ parties out of n are corrupted [14].

In this paper we go one step further and document an implementation of the Orlandi protocol [27] for secure multiparty computation which is both actively secure and tolerates up to $n-1$ corruptions. We further describe our benchmarks of this implementation, compare it to benchmarks of related protocols and argue that the protocol is indeed practical.

The rest of the paper is organized as follows. Section 2 gives an introduction to the Orlandi protocol. Section 3 describes our implementation, and we introduce our benchmarks, their setup, and discuss our results in Section 4. In Section 5 we describe and discuss how we removed the main performance bottlenecks using a high-performance implementation of the Paillier cryptosystem. A description of related work is given in Section 6, and we conclude in Section 7 along with discussing future work.

2   The Orlandi Protocol

The Orlandi protocol [27] is a protocol for secure multiparty computation on arithmetic circuits, which is secure against up to $n-1$ static, active adversaries. We will introduce the protocol in this section by giving a high-level description of the protocol and a detailed account of the parts of the protocol which have been the target of our optimizations. The Orlandi protocol is based on a Verifiable Secret Sharing (VSS) scheme, secure against a dishonest majority, augmented with a protocol for generating random shared multiplicative triples, based on a homomorphic cryptosystem. The Orlandi protocol needs a group $\mathbb{G}$ of some prime order p which is specified by the generator $g \in \mathbb{G}$. The order p and the generator g are part of the public parameters. A secret $x \in \mathbb{Z}_p$ is shared in the Orlandi protocol using additive secret sharing. Every party of the computation holds a share x_i of the secret, two uniform randomly chosen additively secret shared elements $\rho_{i,1}$ and $\rho_{i,2}$ in $\mathbb{Z}_p$ and a public commitment C. The two random elements $\rho_{i,1}$ and $\rho_{i,2}$ are needed in order to compute the commitment to the secret, and the commitment is used when reconstructing the secret to check

that no party contributed a wrong share. The commitment is computed using a double trapdoor Pedersen commitment scheme [30] based on the hardness of the discrete logarithm in the group $\mathbb{G}$. The commitment C is computed as $C = \text{Com}(x, \rho_{i,1}, \rho_{i,2}) = g^x h_1^{\rho_1} h_2^{\rho_2}$ where $h_i = g^{t_i}$ for $i \in \{1, 2\}$ and t_i is a trapdoor. We denote h_1, h_2 as the public key of the commitment scheme. A share in the Orlandi protocol is a four-tuple $(\mathbb{Z}_p \times \mathbb{Z}_p \times \mathbb{Z}_p \times \mathcal{C})$, consisting of the share of the secret, $x_i \in \mathbb{Z}_p$, two uniformly randomly chosen numbers $\rho_1, \rho_2 \in \mathbb{Z}_p$, and a commitment $C \in \mathcal{C}$ to the secret. We write a share of the secret x as $[x]$. The protocol is secure in the Common Reference String (CRS) Model [12], and a proof of the security is sketched in Orlandi's PhD progress report [27] under the assumption of the hardness of the discrete logarithm problem in $\mathbb{G}$, the availability of a secure broadcast protocol, and the semantic security of the homomorphic cryptosystem. The security of the protocol holds, up to the security level 2^{-s} if λ and d are chosen such that

$$s < d \, \log_2(M) + (d+1) \log_2(\ln(1+\lambda)) + 2 \qquad (1)$$

where M is the number of multiplicative triples needed for a given computation. We refer the reader to Orlandi's PhD progress report [27] for the intuition behind the above expression. The parameters λ and d are used in the definition of the commands below.

The protocol can be divided into two parts: a preprocessing part where multiplicative triples are generated and an online part where arithmetic expressions are evaluated. The online part provides the commands one would usually expect from a VSS scheme such as commands for sharing a given value $\texttt{Input}(x)$, reconstructing a secret $\texttt{Open}([x])$, creating a random secret $\texttt{Rand}()$, addition, subtraction, and multiplication ($\texttt{Mul}([x], [y], [a], [b], [c])$) of shared numbers. We will not explain these commands further, except for the multiplication command. We instead refer the reader to Orlandi's PhD progress report [27]. The preprocessing part is divided into a number of building blocks (*leak-tolerant multiplication*, *triple generation*, and *triple test*), which are composed into the final triple generating (*random triple generation*) functionality which produces a list of triples. We will describe online multiplication, triple generation, and random triple generation below.

Basic Multiplication We define the multiplication of the shares $[x]$ and $[y]$ as $[z] = \texttt{Mul}([x], [y], [a], [b], [c])$ where we assume that the parties are given a random triple $([a], [b], [c])$ s.t. $c = a \cdot b$ from a honest dealer. The multiplication is realized as follows:
1. $d = \texttt{Open}([x] - [a])$ and $e = \texttt{Open}([y] - [b])$
2. $[z] = e[x] + d[y] - de + [c]$

The basic multiplication is used both as a building block in the preprocessing phase and also for performing online multiplications. This is the main reason why multiplicative triples are generated in the preprocessing, so that they can be used in online multiplications. It also indicates that one multiplication requires one triple.

The leak-tolerant multiplication of shares $[x]$ and $[y]$ is defined as $[z] = \mathtt{LTMul}([x], [y], \mathcal{M})$ where $\mathcal{M} = \{([a_i], [b_i], [c_i])\}_{i \in \{1,\ldots,2d+1\}}$ is a set of multiplicative triples. Leak-tolerant multiplication is an extension of the basic multiplication with the property that if $d+1$ triples (a_i, b_i, c_i) are uniformly random in view of the adversary then the protocol leaks no information about x, y, and $x \cdot y$.

$\mathtt{TripleGen}()$ generates a triple by having each party first choose random shares $[a]$ and $[b]$ including the needed randomness and the commitments. Second, each party encrypts $(\mathrm{Enc}_{\mathrm{ek}_i}(a_i))$ his share a_i using his public key ek_i and a homomorphic cryptosystem. Then he broadcasts the encrypted share, the corresponding commitment, and the commitment for b_j. The share of the product $[c] = [a] \cdot [b]$ is computed by using the homomorphic property of the received encrypted values to multiply the shares $[a_i]$ and $[b_j]$. The product is then masked with some randomness $d_{i,j}$ and sent. The share c_i is then computed by decrypting $\mathrm{Dec}_{\mathrm{sk}_i}(\gamma_{i,j})$ the product shares, adding them up and subtracting the randomness. The private key of party i is sk_i.

Triple Generation The triple generation command $\mathtt{TripleGen}()$ creates a multiplicative triple which is shared among the parties. The triple generation is realized as follows:

1. Every party P_i chooses $a_i, r_{i,1}, r_{i,2} \in_R \mathbb{Z}_p \times \mathbb{Z}_p \times \mathbb{Z}_p$, computes $\alpha_i = \mathrm{Enc}_{\mathrm{ek}_i}(a_i)$, $A_i = \mathrm{Com}(a_i, r_{i,1}, r_{i,2})$, and broadcasts them
2. Every party P_j does:
   (a) choose $b_j, s_{j,1}, s_{j,2} \in_R \mathbb{Z}_p \times \mathbb{Z}_p \times \mathbb{Z}_p$, compute $B_j = \mathrm{Com}(b_j, s_{j,1}, s_{j,2})$ and broadcast B_j
   (b) Party P_j does, for every other party P_i: choose $d_{i,j} \in_R \mathbb{Z}_{p^3}$ compute and send $\gamma_{i,j} = \alpha_i^{b_j} \mathrm{Enc}_{\mathrm{ek}_i}(1;1)^{d_{i,j}}$ to P_i
3. Every party P_i does:
   (a) compute $c_i = \sum_j \mathrm{Dec}_{\mathrm{sk}_i}(\gamma_{i,j}) - \sum_j d_{i,j} \bmod p$
   (b) pick $t_{i,1}, t_{i,2} \in_R \mathbb{Z}_p \times \mathbb{Z}_p$, compute and broadcast $C_i = \mathrm{Com}(c_i, t_{i,1}, t_{i,2})$
4. Everyone computes $(A, B, C) = (\prod_i A_i, \prod_i B_i, \prod_i C_i)$
5. Every party P_i outputs:
   $([a]_i, [b]_i, [c]_i) = ((a_i, r_{i,1}, r_{i,2}, A_i), (b_i, s_{i,1}, s_{i,2}, B_i), (c_i, t_{i,1}, t_{i,2}, C_i))$

The computation inside encrypted values gives rise to the requirement that the modulus of the cryptosystem N must be much larger than the modulus of the shares and the commitment scheme p. This is not an issue in practice because the key size of a factorization based cryptosystem is usually much bigger than the order of the group of points on an elliptic curve, if the same level of security is to be obtained.

The triple test command $\mathtt{TripleTest}()$ creates one multiplicative triple from two. The first triple is used to check the correctness of the second triple. This removes the risk of overflow in the encrypted computation in $\mathtt{TripleGen}()$ with overwhelming probability. The overflow may occur due to the differrence in the modulus of the cryptosystem and the shares and the commitment scheme.

Random triple generation $\mathtt{RandomTriple}()$ creates a set of multiplicative triples $\mathcal{M}$ of size M, which we call the *result set*. The result set is created

by first generating a larger *distillation set* $\mathcal{D}$ of triples using `TripleTest()`. The size of the distillation set depends on the security parameter and M. The result set is distilled from the distillation set by first choosing a uniformly random subset called the *test set* $\mathcal{T} \subset \mathcal{D}$ of size $\lambda(2d+1)M$. The triples in the test set are checked for correctness, and if any inconsistency is detected the protocol is aborted. Second the remaining triples $\mathcal{D}\backslash\mathcal{T}$ are partitioned into M random sets of size $(2d+1)$. The result set is generated using these sets and the $\mathcal{F}_{\mathrm{PP}}(\mathbf{rand}, \ldots)$ functionality.

Random Triple Generation The implementation of the random triple generation command `RandomTriple()` creates a set $\mathcal{M}$ of multiplicative triples of size M which is shared among the parties. The random triple generation is realized as follows:

1. $\mathcal{D} = \emptyset$. For $i = 1, \ldots, (1+\lambda)(2d+1)M$ do: $\mathcal{D} = \mathcal{D} \cup \texttt{TripleTest()}$
2. Coin-flip a subset $\mathcal{T} \subset \mathcal{D}$ of size $\lambda(2d+1)M$
3. For all $i \in \mathcal{T}$ the parties reveal the randomness used for `TripleTest()`
4. Check that the randomness is consistent with the view. Check that $a_i, b_i < p$ and $d_{i,j} < p^3$. Abort otherwise.
5. Partition $\mathcal{D}\backslash\mathcal{T}$ in M random subsets $\mathcal{D}_i$ of size $(2d + 1)$
6. For $i = 1, \ldots, M$ do:
   (a) $[a] = \mathcal{F}_{\mathrm{PP}}(\mathbf{rand}, \ldots)$, $[b] = \mathcal{F}_{\mathrm{PP}}(\mathbf{rand}, \ldots)$, $[r] = \mathcal{F}_{\mathrm{PP}}(\mathbf{rand}, \ldots)$
   (b) $[c] = \texttt{LTMul}([a], [b], \mathcal{D}_i)$ and $\texttt{Open}([c] + [r])$
   (c) Add $([a], [b], [c])$ to $\mathcal{M}$

The $\mathcal{F}_{\mathrm{PP}}(\mathbf{rand}, \ldots)$ functionality used in the random triple generation creates a random share using the $\prod_{\mathrm{comm}}$ protocol described in Chapter 4 of Orlandi's PhD progress report [27]. The difference between using the `Rand()` function and the $\mathcal{F}_{\mathrm{PP}}$ functionality is twofold. First, they differ in the security model of the overall protocol. If one uses the `Rand()` function then the protocol provides stand-alone security [11] whereas if one uses the $\mathcal{F}_{\mathrm{PP}}$ functionality then it is secure in the CRS model. Second, they differ in speed. The `Rand()` function is faster than the $\mathcal{F}_{\mathrm{PP}}$ functionality because the latter generates random shares using Universal Composable commitments whereas the first does not. In the implementation we use the `Rand()` function and thus achieve stand-alone security.

In the original protocol it is assumed that the public key for the commitment scheme is provided to the parties by a trusted third party (TTP), so that the key is randomly chosen. However in a real world setting, where the parties don't trust each other, it might not be the case that there is a single TTP that all parties trust. Other ways of generating the public key might include: measuring some physical random quantity, running a coin-flip protocol, or modeling a hash function with a random oracle (e.g. the first party can choose a random string r and publish $(r, H(r))$ and everyone parses $H(r)$ as the public key). The security of the whole protocol will reflect the security of the method used to generate the public key.

3 Implementation of the Orlandi Protocol

In this section we describe how we implemented the Orlandi protocol using VIFF, the Virtual Ideal Functionality Framework [14, 32]. VIFF is an open source framework implemented in Python for executing general multiparty computations. It is possible to extend VIFF with new protocols for evaluation of arithmetic circuits. Such protocols are called *runtimes* in VIFF lingo and are materialized by the `Runtime` class, which new runtimes must subclass. A share in VIFF is represented by instances of the `Share` class. A `Share` instance represents a value to be compute in the future, and one can attach callbacks which will be executed once the share gets a concrete value. A share in the Orlandi protocol is represented using the `OrlandiShare` class which extends `Share`. The concrete value held by an `OrlandiShare` form a tuple as described in Section 2.

The Orlandi protocol is implemented as the `OrlandiRuntime`, a subclass of `Runtime` and as such overloading the usual addition, subtraction and multiplication operators. It also provides some further methods largely corresponding to the commands described in Section 2. The implementation of the various commands follows the protocol closely, except that we combine steps and/or schedule them in parallel whenever possible. An example where we combine steps is step 1, 2.a, and 2.b of `TripleGen()` where we save one broadcast operation. An example of scheduling operations in parallel is the `TripleTest()` command where two `TripleGen()` commands are scheduled in parallel with one `Open()` and a `Rand()` command.

To speed-up the computation it can be observed that in step 2.c of the `TripleGen()` function that $\text{Enc}_{\text{ek}i}(1;1)$ will result in g^{N+1} when encrypting with the Paillier system. Hence, γ_{ij} can be computed by using a simultaneous multi-exponentiation method as described in Section 5 i.e. $\gamma_{ij} = \alpha_i^{b_j}(g^{N+1})^{d_{ij}}$. In addition, when using homomorphic properties of the Paillier cryptosystem, step 3.a can be rewritten to $c_{ij} = \text{Dec}_{\text{sk}}(\prod_j \gamma_{ij} \mod N^2) - \sum_j d_{ij}$, which results in just doing one exponentiation in total instead of one per party.

The security of the Orlandi protocol is based on the assumptions of the hardness of the discrete logarithm of the group used and the presence of a broadcast channel. We satisfy the hardness assumption of the discrete logarithm by computing the commitments in a group defined by an elliptic curve over the field $\mathbb{F}_p$ with prime p of 192-bits with the generator g and the public key h_1, h_2 which consists of points on the curve. The computation of a commitment can be reformulated in terms of the corresponding operations on an elliptic curve. Multiplication of two group elements becomes addition, division becomes subtraction, and exponentiation by a scalar becomes multiplication by a scalar. A commitment is thus computed as $x_i \cdot g + \rho_{i,1} \cdot h_1 + \rho_{i,2} \cdot h_2$ and the result is a point on the curve. We have implemented the commitment scheme as a Python C extension using the industry strength PrimeInk ECC library v. 6.4.0 [1]. The main obstacle was the conversion from integers in base 2^{15}, which is used as the internal representation of arbitrary precision integers in Python, to base 2^{32} which is the representation used by PrimeInk ECC. The broadcast channel assumed by the Orlandi protocol is implemented using an instance of the *weak-*

crusader broadcast. The weak-crusader broadcast is a variant of the *crusader* broadcast [16] where we allow a malicious adversary to make some honest parties output a message while others abort. The crusader broadcast is not needed in the Orlandi case since the protocol is already vulnerable to denial of service attacks, e.g. an adversary can just refrain from sending messages at all. By relaxing the requirements on the broadcast protocol we also get a more efficient implementation since we do not need a signature scheme. The protocol consists of two rounds. In the first round the *senders* send a value to each of the receivers, who then computes a collision resistant hash of the received value, and sends it to the other receivers in the second round, who check the correctness. We generally use the broadcast protocol in the implementation for broadcasting from a set of parties to all parties, except for the share reconstruction command in the case where only some subset of the parties should learn the output.

Broadcast $ls = \text{broadcast}(value, senders, receivers)$, where the result ls is a list of received values.
1. Each party $P_j \in senders$ sends *value* to every party $P_i \in receivers$
2. Every party P_i in *receivers* computes a collision resistant hash on the received value and sends the hash to every other party in *receivers*
3. Each party in *receivers* checks that the received hash is equal to the hash computed by the party in the previous step, and returns *value* if true, or aborts if not

4 Benchmarks

In this section we describe how we have benchmarked our implementation with various levels of optimization, and discuss the results. We have chosen to benchmark the three commands `Mult`, `TripleGen`, and `RandomTriple`, because the other commands are not much different than the commands in a standard additive secret sharing scheme. `Mult` and `TripleGen` are straightforward to benchmark since they do not depend on the security parameter. The execution of `RandomTriple` on the other hand depends on the security parameter and the needed number of triples.

The `RandomTriple` command generates a set of triples which is distilled into a smaller set that is the result of the command. The total number of triples generated is $(1 + \lambda)(2d + 1)M$ where M is the size of the result set, and λ and d have to satisfy Equation 1. The overhead of distilling M triples is $(1 + \lambda)(2d + 1) - 1$, and it is clear from Equation 1 that the overhead increases as the security parameter goes up, but also that it decreases as the number of needed triples M increases. This gives two interesting dimensions along which to investigate the execution time.

It is infeasible to benchmark every possible combination of security parameter and number of triples so we chose the security parameter values 1 (covert security [2]), 16, and 21, and 5, 10, and 30 triples, because they are representative and feasible for the interval of interesting security parameters $[1, 32]$. They

are feasible in the sense that they can be computed in a reasonable amount of time. We have chosen λ and d such that the overhead is minimal.

The benchmarks are created using the VIFFBench Framework [33], which automates benchmarking of VIFF protocols. The benchmarks are defined as a small program parametrized with the number of parties and the VIFF repository revision. The results are automatically stored in a relational database. We have chosen to hardwire to numbers t_1, t_2 into the implementation, in order to avoid unnecessary complexity. The numbers are used to compute the public keys (as g^{t_1}, g^{t_2}) for the commitment scheme. This breaks the security of the implementation if one is to use the implementation for practical applications. It does, however, not influence the efficiency of the commands, because the key can be computed in a setup phase, before the preprocessing phase. We have performed three benchmarks which were executed for each of the VIFF revisions containing significant improvements to the commands. Except for random triple generation which is only benchmarked for revision 1435. The online multiplication benchmark consist of 100 multiplications run in parallel. If we only executed one multiplication we would get too close to the resolution of the system clock that it would affect the precision of our measurements. The triple generation and the random triple generation benchmarks, on the other hand, only execute one invocation of the corresponding commands, because the execution time is much longer. For each revision we have repeated each benchmark 50, 50, and 1 times for online multiplication, triple generation, and random triple generation, respectively, in order to eliminate random noise. Executing the random triple generation 50 times for each revision would be prohibitively time consuming. All the benchmarks are performed using 1024-bits key size for the Paillier cryptosystem. We do not investigate how the implementation behaves as the latency on the network changes. The benchmarks were performed by using up to 10 identical computers equipped with 1 GHz dual-core AMD Opteron 2216 processors with 2x1 Mb level 2 cache and 2 Gb RAM each. The hosts are running Red Hat Enterprise Linux 5.2 on a 64-bit x86 architecture and were connected using gigabit Ethernet with a round-trip latency of 0.104 ms. One of the machines was chosen as the coordinator, whose responsibility it was to distribute and execute the benchmarks on the needed subset of the nine other machines. VIFFBench chooses the subset randomly.

4.1 Benchmark Results

The results of the basic multiplication benchmarks are shown in Table 1 where the average execution time for one multiplication is presented for two to nine parties along with the standard deviation. We clearly see that the implementation is efficient and achieves 15.9 ms per multiplication for three parties. The Figure also shows that the average execution time increases linearly as the number of parties increases which is as expected due to the broadcast. A multiplication basically consists of two `Open()` operations with execution time linear in the number of parties. The execution time for two parties is not as expected. Based on the protocol we would expect it to be faster than for three parties, but the

parties	2	3	4	5	6	7	8	9
time (ms)	27.4	15.9	19.7	22.8	25.6	26.7	28.2	35.9
stdvar (ms)	0.1	3.5	4.7	6.7	7.4	6.8	8.1	8.3

Table 1. The average *execution time* in ms. of selected Basic Multiplication benchmarks as function of the number of *parties.*

measurements shows that it is slower than for three, four, five, six, and even seven parties. We contemplate that the cause of this anomaly is that for two parties the implementation is CPU bound and not network bound as we have observed for three parties. The standard deviations are large compared to the measurements, and indicates some variation in our measurements. However the timings are meaningful and the basic multiplication is useful in practice even if we take the standard deviation into account.

Table 2 shows the average execution time of triple generation for two, three, and nine parties and the data is visualized as a graph in Figure 1. We only show a subset of our measurements, please see Table 5 in the Appendix for the full set of measurements. We have benchmarked different revisions of our implementation corresponding to the various optimizations we have performed. Revision 1231 is the initial unoptimized implementation which uses the implementation of Paillier in VIFF, revision 1355 in-lined step 1, 2.a, and 2.b of `TripleGen()`, 1370 uses our efficient implementation of the Paillier cryptosystem, 1393 moves step 2.c into C, 1399 moves step 3.a into C, and 1400 is a minor technical optimization.

The performance of the final revision is below 200 ms for all but two and four parties. This is encouraging for practical uses of the protocol. Based on the definition of `TripleGen()` we would expect to see the execution time increase linearly ($\mathcal{O}(n)$) in the number of players n. This is also the case until revision 1393. One explanation is that random noise is more dominant when the measured time is small. We again see that two parties are slower than even nine parties, but it seems like the anomaly is introduced in revision 1370, where we use a more efficient implementation of Paillier. This is consistent with our earlier observation that the two party case is CPU bound and the other are network bound. It is

parties		1231	1355	1370	1393	1399	1400
2	time	3519.6	3519.6	894.6	243.8	226.5	224.2
2	stdvar	1.0	0.8	3.2	0.9	0.7	0.7
3	time	3972.7	4012.1	376.3	155.0	168.3	170.9
3	stdvar	94.8	157.4	72.1	59.2	35.9	38.2
9	time	8937.4	8849.7	846.9	237.0	188.9	188.4
9	stdvar	460.2	281.2	27.0	36.5	20.7	29.0

Table 2. The average *execution time* in ms. of triple generation as a function of number of *parties.*
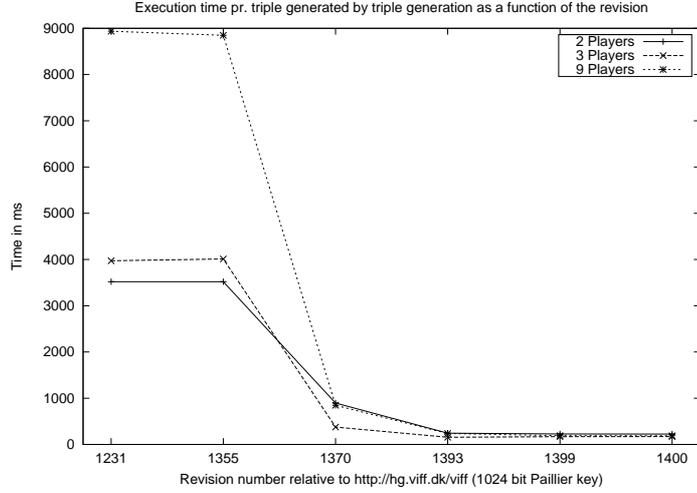
Fig. 1. The average *execution time* in ms. of selected Triple Generation benchmarks as a function of the changes to the implementation.

clear from Figure 1 that the use of an efficient implementation of Paillier gives a substantial improvement of the execution time and is the main contributor to the efficiency of the Orlandi implementation. The Figure shows that rewriting step 2.c in C gives a larger performance increase then rewriting step 3.a does, which is as we would expect. Step 2.c is more computational intensive. The improvements we have done in the Python code in revision 1355 and 1400 are dwarfed by the other improvements.

Table 3 shows the average execution time of random triple generation defined in revision 1435 for two, three, and nine parties. The full set of measurements can be found in Table 6. One would expect the benchmarks to show that the execution time per triple increases as the security parameter increases ($\mathcal{O}(d \log \ln(\lambda))$) and decreases as the number of triples increases. The measurements shows that the execution time increases as the security parameter increases. And, in most cases the execution time also decreases as the number of triples increases. Ran-

s	1	1	1	16	16	16	21	21	21
t	5	10	30	5	10	30	5	10	30
2	1.872	1.511	1.370	15.879	15.157	11.641	20.959	16.560	16.453
3	1.598	0.952	1.059	11.796	11.883	10.944	16.931	15.981	15.269
9	2.238	1.799	1.794	25.931	24.444	25.638	31.901	32.572	37.545

Table 3. The average execution time in seconds of random triple generation as a function of *parties* (2, 5, and 9), *security parameter*, and *number of triples*.

dom noise may explain the cases where we do not see the a decrease in execution time. We have only run the benchmarks once for each combination of security parameter and number of triples due to time considerations.

4.2 Performance Comparison

We are aware of two other implementations of secure multiparty computation protocols with active security: A protocol by Damgård, Geisler, Krøigaard, and Nielsen (DGKN) which has been implemented in VIFF [14], and a protocol by Lindell, Pinkas, and Smart (LPS) [21]. The performance of these implementations cannot be directly compared to the performance of the Orlandi protocol since they rely on different security models and the benchmarks have been done on different hardware. However, in this section we will try to elaborate on the difference between the performance of the systems.

The DGKN protocol provides evaluation of arithmetic circuits and is secure against an adaptive active adversary up to a threshold of $n/3$ corrupted parties. An adversary may halt the computation up to a synchronization point, not after - in which case termination is guaranteed. The LPS protocol is a 2-party protocol for evaluation of boolean circuits. The Orlandi protocol is a full-threshold multiparty protocol. Both the Orlandi and the LPS protocol are secure against a static active adversary, the security is based on cryptographic assumptions, and they are "unfair" in the sense that a corrupt party can prevent honest parties from getting any result while the corrupt party get results himself.

It is difficult to make a direct comparison between our results and those reported for the LPS protocol, since they do not benchmark multiplications, but rather comparisons of 16-bit integers.

Results have been reported for the DGKN protocol for 4, 7, 10, 13, 16, 19, 22, and 25 parties. If we compare the numbers for 4 and 7 parties, which is the setups we have numbers for, then DGKN takes 4 and 6 ms which is only a factor 5 (roughly) better than our results for the online case. Whereas in the preprocessing case the DGKN uses 5 ms and 22 ms in the best case, which is a factor 80-240 better then our implementation even with the smallest security parameter. Based on these numbers our implementation may seem inferior, but remember that the Orlandi protocol provides full threshold. And the benchmarks show that it is possible to use the Orlandi protocol in practice.

5 High-performance Paillier

Various non-deterministic cryptosystems have been proposed based on randomized encryption schemes which encrypt a message m by raising a base g to the power m and suitably randomizing this result [5, 15, 17, 25, 26, 28]. The security of these systems is based on the intractability of various "residuosity" problems. As an important consequence of this encryption technique, those schemes have homomorphic properties. These homomorphic properties enable computation on

- **Key Generation**: Let N be a RSA modulus $N = pq$, where p and q are large prime integers. Let $g \in \mathbb{Z}_{N^2}^*$ be chosen such that its order is a multiple of N. Let $\lambda(N) = \mathrm{lcm}(p-1, q-1)$. The public key is (g, N), and the private key is $\lambda(N)$
- **Encryption**: To encrypt a message $m \in \mathbb{Z}_N$, randomly chose $r \in \mathbb{Z}_N^*$ and compute the ciphertext $c = g^m r^N \mod N^2$.
- **Decryption**: The decryption of c is defined by $\frac{L(c^\lambda \mod N^2)}{L(g^\lambda \mod N^2)} \mod N$ Where the $L(\mu)$ function is defined as $\frac{\mu - 1}{N}$ and takes inputs of $S_N = \{u < N^2 \mid u = 1 \mod N\}$.

Fig. 2. The main variant.

- **Key Generation**: Let N be a RSA modulus $N = pq$, where p and q are large prime integers. Let $\lambda(N) = \mathrm{lcm}(p-1, q-1)$ and choose α such that it divides $\lambda(N)$. Let $h \in \mathbb{Z}_{N^2}^*$ such that is has maximal order of $n\lambda(N)$, and $g = h^{\lambda/\alpha} \mod N^2$. The public key is (g, N), and the private key is α
- **Encryption**: To encrypt a message $m \in \mathbb{Z}_N$, randomly chose $r \in \mathbb{Z}_N^*$ and compute the ciphertext $c = g^{m + r \cdot N} \mod N^2$.
- **Decryption**: The decryption of c is defined by $m = \frac{L(c^\alpha \mod N^2)}{L(g^\alpha \mod N^2)} \mod N$ Where the $L(\mu)$ function is defined as $\frac{\mu - 1}{N}$ and takes inputs of $S_N = \{u < N^2 \mid u = 1 \mod N\}$.

Fig. 3. The subgroup variant.

ciphertexts without knowing the context. This allows for a wide spread of applications such as secure multiparty computations, like Orlandi's protocol. In this section we discuss the Paillier cryptographic system and its implementation and optimization.

5.1 Description of the Paillier Schemes

Paillier has presented multiple closely related cryptosystems [28, 29]. We will focus on the main- and subgroup variants of these cryptosystems shown in Figure 2 and 3, respectively. The subgroup variant is slightly different as it computes residues in a subgroup of order $\lambda(N)$.

5.2 Paillier Performance Evaluation

A common task in implementations of many public-key cryptosystems is multiexponentiation in commutative groups. This is also the case for the Paillier cryptosystem, namely computing $g^m r^N \mod N^2$ for the main variant and $g^m (g^N)^r \mod N^2$ for the subgroup. Many algorithms have been proposed to speed-up the computation a single exponentiation [10, 24, 35, 20, 23]. These algorithms can be modified to compute a product over of multiple exponentiations in such a way that it is faster then a product of single exponentiations. In the following

subsections we show how to reduce this overhead with different simultaneous multi-exponentiation algorithms.

The simultaneous 2^k-ary method was first introduced by Brauer [10], the idea behind the method is slicing the binary representation of an exponent into pieces using a windows of length k and processing the exponent in a larger basis. For each evaluation of the exponent the intermediate results get raised by power of 2^k and multiplied by its base raised to the power of the evaluated bits in the exponent. The powers $\{0, 1, 2, \ldots, 2^k - 1\}$ of base g are precomputed in an auxiliary table.

In order to make the 2^k-ary method evaluate two powers at the same time (i.e. $g_1^{e_1} g_2^{e_2} \mod n$). Two separate auxiliary tables with there powers of g_1 and g_2 are required. Each time both exponents get evaluated at the same time. First the intermediate result is raised to 2^k and is multiplied by each separate base raised to the power of the evaluated bits from the corresponding exponent. This saves a squaring for every bit that is evaluated.

Algorithm 1 2^k-ary Method

Input: $aux_a, aux_b, b = 2^k - 1, e_1, e_2$
Output: $g_1^{e_1} \cdot g_2^{e_2} \mod n$
  $A \leftarrow 1$
  for $j = \lfloor (b-1)/w \rfloor w$ down to 0 do
  $A \leftarrow A^{2^k}$
    if $(e_1[j + w - 1 \ldots j])$
      $A \leftarrow A \cdot aux_{g_1}[e_{1,j+1}, e_{1,j+2}, \ldots e_{1,j+w-1}]$
    if $(e_1[j + w - 1 \ldots j])$
      $A \leftarrow A \cdot aux_{g_2}[e_{2,j+1}, e_{2,j+2}, \ldots e_{2,j+w-1}]$

The simultaneous 2^k-ary matrix method is a slight modification of the simultaneous 2^k-ary method. The main difference is the computation of the auxiliary table which consists of a $k \times k$ table entries which holds for $0 < i, j < 2^k - 1$ the product of $g_1^i g_2^j$ in entry $aux[i][j]$. Building such a table requires more precomputation, but gives one less multiplication per evaluated window of length k.

The simultaneous sliding window exponentiation method of Yen, Laih, and Lenstra [35] is an improvement of the 2^k-ary method. Just like the 2^k-ary method the sliding window method consists of slicing the binary representation of e_i into pieces using a window of length ω and processing the part one by one. The addition of letting the window slide allows us to skip consecutive zeros in e_i while squaring the intermediate result. As a result, evaluation of two even exponents are avoided, and computation of the entries of these entries in the auxiliary table can be avoided. This results in a generally faster algorithm for evaluating exponents.

To evaluate two exponents simultaneously we apply the same trick as for simultaneous 2^k-ary method. But with the change that we check if both evaluated bits are zero. Additional bookkeeping is needed to keep track of the bits.

When comparing both decryption version of the Paillier scheme, the basic computation consists of one fixed exponentiation and a multiplication in $\mathbb{Z}_{N^2}^*$ and a multiplication in $\mathbb{Z}_N^*$. The subgroup variant requires the same operation, except the size of the exponent α is smaller, which makes the subgroup variant faster. Paillier has suggested an alternative decryption method by means of the Chinese Remainder Theorem (CRT). By defining $L_p = \frac{\mu-1}{p}$ and $L_q = \frac{\mu-1}{q}$ we can decrypt by separately computing the message modulo p and q and combining m_p and m_q with CRT. First compute $h_p = L_p(g^{p-1} \mod p^2)$ and $h_q = L_q(g^{q-1} \mod q^2)$ then $m_p = L_p(c^{p-1} \mod p^2)h_p \mod p$ and $m_q = L_q(c^{q-1} \mod q^2)h_q \mod q$ and finally recombine using CRT. Additional speed-up can be found by computing $L(\mu)$ efficiently with as $\mu \cdot n^{-1} \mod 2^{|n|}$. This is just a multiplication and a logical `AND`. Another way to make computations more efficient is a careful choice of parameters. For instance if one chooses $g = 1 + n$ then the exponentiation g^m can be executed using only one multiplication, namely $g^m = (1+n)^m \equiv (1+mn) \mod n^2$. This only works for the encryption in the main variant of the Paillier scheme. Such optimizations can provide substantial speed-ups as we show in the next subsection.

5.3 Results

In this section we describe how we benchmarked our implementation with various optimizations and discuss the results. We have chosen to benchmark the three above simultaneous multi-exponentiation algorithms and their parameters for key sizes N ranging from 1024-bit to 4096-bit with increments of 1024-bits.

To benchmark the speed of encryption of the Paillier cryptosystem we ran the main variant with $g = N + 1$ and the subgroup variant with the 3 different simultaneous multi-exponentiation algorithms, with window size k as parameter. For windows size k we select $1 < k \leq 5$, as choosing k higher then 5 will result in longer pre-computation for these bit sizes.

The benchmarks can be found in Figure 4. The speed of the algorithms becomes clear as the key-sizes increases. As it is infeasible to test all combinations of random element r and message m, we have chosen m and r to have the same size as the sub-group length in bits this is approximately $1/4$ of N. This is due to the requirement of the Paillier cryptosystem. The benchmarks are taken without creating of the auxiliary table, the timings for creating the auxiliary table can be found in table 5.3.

It is clear that the main variant of the Paillier with optimized parameters is slowest, this is mostly due to the computation of r^N with N having $\{1024, \ldots 4096\}$-bits. The simultaneous 2^k-ary normal and matrix- variant as well as simultaneous sliding window method perform better with a greater k if the key length gets bigger. Also, we can see that 2^k-ary matrix performs better than 2^k-ary method as the key size grows. This is 2^k-ary matrix has larger pre-computation but has one multiplication less for evaluating one bit of the
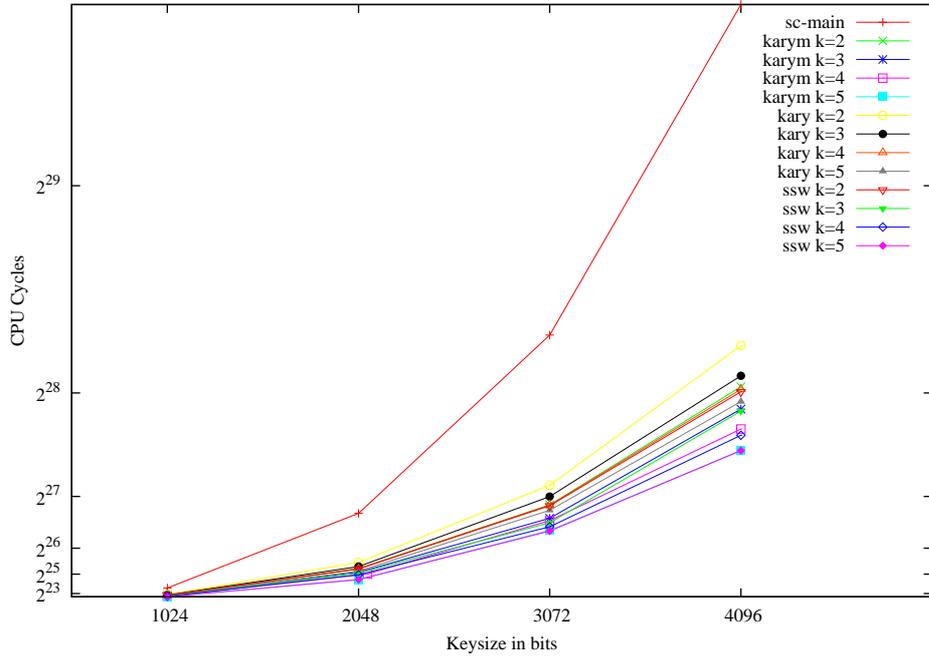
Fig. 4. The execution time in CPU cycles with different keys sizes with simultaneous sliding window method (ssw), simultaneous 2^k-ary method (kary) and simultaneous 2^k-ary method (karym) parameter k is the size of the window in bits and the main variant (sc-main) with $g = n + 1$.

exponent. The simultaneous sliding window method is in most cases the best performing algorithm, this is due the fact that it skips consequent zeros.

The benchmarks were performed by using a 2 GHz Intel Pentium E2180 dual core with 1024KB cache per core and 2 GB Ram. The system is running Fedora release 8 with kernel 2.6.26.8-57.fc8 in 64-bit mode. For benchmarking we used *cpucycles* with is part of eBACS [6] to measure the amount of CPU cycles used by the execution.

6   Related Work

Several practical systems for general multiparty computation have been implemented during the recent years. FairPlay [22] is the earliest implementation that the authors are aware of. In the system one can specify computations in a high-level, procedural programming language. Using the FairPlay compiler, the high-level programs are then compiled to low-level representations of one-pass boolean circuits. These circuits are then used for secure computation as described by Yao [34]. The timings reported on FairPlay show that FairPlay is efficient, but

	1024-bits(ms)	2048-bits(ms)	3072-bits(ms)	4096-bits(ms)
2^k-ary, $k=2$	0.04	0.11	0.24	0.41
2^k-ary, $k=2$	0.11	0.34	0.73	1.22
2^k-ary, $k=2$	0.27	0.80	1.72	2.85
2^k-ary, $k=2$	0.58	1.72	3.66	6.09
2^k-ary m, $k=2$	0.12	0.38	0.80	1.32
2^k-ary m, $k=3$	0.55	1.77	3.68	6.20
2^k-ary m, $k=4$	2.26	7.31	15.25	25.62
2^k-ary m, $k=5$	9.14	29.54	61.51	103.77
ssw $k=2$	0.03	0.12	0.24	0.41
ssw $k=2$	0.33	0.87	1.73	2.86
ssw $k=2$	0.39	1.22	2.39	4.03
ssw $k=2$	5.14	16.78	33.61	56.40

Table 4. The *execution time* in miliseconds with differnt keysizes for creation of the auxiliary tables. With 2^k-ary, 2^k-ary (matrix) and ssw (simultaneous sliding window)

it should be noted that it only supports two-party computation in the passive security model. FairPlay has later been supplemented by FairPlayMP [3] which is capable of handling the case with more than two parties in the passive security model assuming less than $n/2$ corrupted parties. A two-party protocol for secure computation which is secure against a static active adversary has recently been implemented [21]. A further optimized version is reported in [31] where it is used for a practical implementation of two-party AES. Like the protocol used in FairPlay, these protocols are also based on boolean circuits.

Another practical system for general multiparty computation was created by Bogetoft et al. [9] in the SIMAP project. The system was used for the first known large-scale commercial application of secure multiparty computation [8]. It supports general multiparty computation in a passive threshold security model assuming less than $n/2$ corrupted parties. Like FairPlay it lets users express programs in a high level language, but contrary to FairPlay, it evaluates the programs as arithmetic rather than boolean circuits. The downside of this strategy is that comparison of integers becomes more complex and time consuming. The protocol used in the SIMAP system has also been implemented in the VIFF framework [32]. In addition, VIFF contains a passively secure two-party protocol based on the Paillier cryptosystem as well as an implementation of a multiparty protocol described in Section 4. The ShareMind system [7] represents yet another efficient approach to practical multiparty computation based on arithmetic circuits and additive sharing. It only supports three parties in the passive model and assumes that at most one party gets corrupted. None of the above implementations, though, support the combination of active security and self-trust, that is available with the implementation of the Orlandi protocol described in this paper.

7 Conclusion and Future Work

In this paper we presented an implementation of the Orlandi protocol, which is the first implementation of a MPC protocol based on arithmetic circuits, which is secure against up to $n-1$ static, active adversaries. We showed that the protocol can be implemented efficiently in the presence of an efficient implementation of a double trapdoor Petersen commitment scheme and a homomorphic cryptosystem. We also described an efficient implementation of the Paillier cryptosystem.

Practical uses is an interesting direction of future work e.g. auctions, benchmarks, and online games. Also the implementation of a suitable setup phase is interesting. Furthermore it would be interesting to implement and benchmark the $\mathcal{F}_{PP}(\mathbf{rand}, \ldots)$ functionality. This would also make the implementation secure in the common reference string model.

A further direction of future work would be to implement and benchmark the Lim/Lee [20] and the fractional window exponentiation [23] algorithms, which we expect would provide further speed up.

Acknowledgements

References

1. Cryptomatic A/S. PrimeInk ECC library v. 6.4.0. `http://www.cryptomatic.com`.
2. Yonatan Aumann and Yehuda Lindell. Security against covert adversaries: Efficient protocols for realistic adversaries. In Salil P. Vadhan, editor, *Theory of Cryptography*, volume 4392 of *Lecture Notes in Computer Science*, pages 137–156. Springer, 2007.
3. Assaf Ben-David, Noam Nisan, and Benny Pinkas. FairplayMP: a system for secure multi-party computation. In Peng Ning, Paul F. Syverson, and Somesh Jha, editors, *ACM Conference on Computer and Communications Security*, pages 257–266. ACM, 2008.
4. Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In J'anos Simon [18], pages 1–10.
5. Josh Danieln Cohen Benaloh. *Verifiable Secret-Ballot Elections*. PhD thesis, Yale University, 1978.
6. Daniel J. Bernstein and Tanja Lange. eBACS: ECRYPT benchmarking of cryptographic systems. `http://bench.cr.yp.to`.

7. Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A framework for fast privacy-preserving computations. In Sushil Jajodia and Javier López, editors, *ESORICS 2008, 13th European Symposium on Research in Computer Security*, volume 5283 of *Lecture Notes in Computer Science*, pages 192–206. Springer, 2008.

8. Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, Michael I. Schwartzbach, and Tomas Toft. Secure multiparty computation goes live. In Roger Dingledine and Philippe Golle, editors, *Financial Cryptography*, volume 5628 of *Lecture Notes in Computer Science*, pages 325–343. Springer, 2009.

9. Peter Bogetoft, Ivan Damgård, Thomas P. Jakobsen, Kurt Nielsen, Jakob Pagter, and Tomas Toft. A practical implementation of secure auctions based on multiparty integer computation. In Giovanni Di Crescenzo and Aviel D. Rubin, editors, *Financial Cryptography*, volume 4107 of *Lecture Notes in Computer Science*, pages 142–147. Springer, 2006.

10. Alfred Brauer. On addition chains. *Bulletin of the American Mathematical Society*, 45(10):736–739, 1939.

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

12. Ran Canetti and Marc Fischlin. Universally composable commitments. In *CRYPTO '01: Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology*, volume 2139 of *Lecture Notes in Computer Science*, pages 19–40, London, UK, 2001. Springer.

13. David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty unconditionally secure protocols (extended abstract). In J'anos Simon [18], pages 11–19.

14. Ivan Damgård, Martin Geisler, Mikkel Krøigaard, and Jesper Buus Nielsen. Asynchronous multiparty computation: Theory and implementation. In Stanislaw Jarecki and Gene Tsudik, editors, *Public Key Cryptography*, volume 5443 of *Lecture Notes in Computer Science*, pages 160–179. Springer, 2009.

15. Ivan Damgård, Martin Geisler, and Mikkel Krøigard. Homomorphic encryption and secure comparison. *International Journal of Applied Cryptography*, 1(1):22–31, 2008.

16. Danny Dolev. The byzantine generals strike again. Technical report, Stanford University, Stanford, CA, USA, 1981.

17. Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28(2):270–299, 1984.

18. *STOC '88: Proceedings of the twentieth annual ACM symposium on Theory of computing*. ACM, May 1988.

19. Pil Joong Lee and Chae Hoon Lim, editors. *Information Security and Cryptology - ICISC 2002, 5th International Conference Seoul, Korea, November 28-29, 2002, Revised Papers*, volume 2587 of *Lecture Notes in Computer Science*. Springer, 2003.

20. Chae Hoon Lim and Pil Joong Lee. More flexible exponentiation with precomputation. In Yvo Desmedt, editor, *CRYPTO*, volume 839 of *Lecture Notes in Computer Science*, pages 95–107. Springer, 1994.

21. Yehuda Lindell, Benny Pinkas, and Nigel P. Smart. Implementing two-party computation efficiently with security against malicious adversaries. In Rafail Ostrovsky, Roberto De Prisco, and Ivan Visconti, editors, *Security and Cryptography for Networks*, volume 5229 of *Lecture Notes in Computer Science*, pages 2–20. Springer, 2008.

22. Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. Fairplay - Secure Two-Party Computation System. In *USENIX Security Symposium*, pages 287–302. USENIX, 2004.

23. Bodo Möller. Improved techniques for fast exponentiation. In Lee and Lim [19], pages 298–312.

24. Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of computation*, pages 519–521, 1985.

25. David Naccache and Jacques Stern. A new public-key cryptosystem. In Walter Fumy, editor, *Advances in Cryptology*, volume 1233 of *Lecture Notes in Computer Science*, pages 27–36. Springer, 1997.

26. Tatsuaki Okamoto and Shigenori Uchiyama. A new public-key cryptosystem as secure as factoring. In Kaisa Nyberg, editor, *Advances in Cryptology*, volume 1403, pages 308–318. Springer, 1998.

27. Claudio Orlandi. LEGO and Other Cryptographic Constructions - PhD Progress Report. `http://www.cs.au.dk/~orlandi/`, March 2009.

28. 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.

29. Pascal Paillier and David Pointcheval. Efficient public-key cryptosystems provably secure against active adversaries. In Kwok-Yan Lam, Eiji Okamoto, and Chaoping Xing, editors, *ASIACRYPT*, volume 1716 of *Lecture Notes in Computer Science*, pages 165–179. Springer, 1999.

30. Torben P. Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In Joan Feigenbaum, editor, *Advances in Cryptology*, volume 576 of *Lecture Notes in Computer Science*, pages 129–140. Springer, 1991.

31. Benny Pinkas, Thomas Schneider, Nigel P. Smart, and Stephen C. Williams. Secure two-party computation is practical. In *15th Advances in Cryptology – ASIACRYPT 2009*, volume 5912 of *Lecture Notes in Computer Science*, pages 250–267. Springer, 2009. Full version available at `http://eprint.iacr.org/2009/314`.

32. VIFF - The Virtual Ideal Functionality Framework. `http://viff.dk`.

33. VIFFBench Framework. `http://bitbucket.org/tpj/viffbench`.

34. Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *Foundations of Computer Science*, pages 162–167. IEEE, 1986.

35. Sung Ming Yen, Chi Sung Laih, and Arjen K. Lenstra. Multi-exponentiation. *Computers and Digital Techniques*, 141(6):325–326, 1994.

A   Homomorphic properties of Paillier cryptosystem

The encryption of message m is denoted by $E(m)$ and decryption ciphertext c is denote by $D(c)$.

$$\forall m1, m2 \in \mathbb{Z}_N \text{ and } k \in \mathbb{N}$$

$$\mathrm{D}(\mathrm{E}(m_1)\mathrm{E}(m_2) \mod N^2) = m_1 + m_2 \mod N$$

$$\mathrm{D}(\mathrm{E}(m_1)g^{m_2} \mod N^2) = m_1 + m_2 \mod N$$

$$\left.\begin{array}{l} \mathrm{D}(\mathrm{E}(m_1)^{m_2} \mod N^2) \\ \mathrm{D}(\mathrm{E}(m_2)^{m_1} \mod N^2) \end{array}\right\} = m_1 m_2 \mod N$$

$$\mathrm{D}(\mathrm{E}(m)^k \mod N^2) = km \mod N$$

B Tables and Graphs

parties	revision count	time (ms)	stdvar (ms)
2	1231	3519.6	1.0
	1355	3519.6	0.8
	1370	894.6	3.2
	1393	243.8	0.9
	1399	226.5	0.7
	1400	224.2	0.7
3	1231	3972.7	94.8
	1355	4012.1	157.4
	1370	376.3	72.1
	1393	155.0	59.2
	1399	168.3	35.9
	1400	170.9	38.2
4	1231	4607.8	416.4
	1355	4399.1	2.3
	1370	388.6	24.0
	1393	131.0	40.4
	1399	161.1	64.7
	1400	206.1	30.9
5	1231	5288.6	49.6
	1355	5279.0	2.4
	1370	482.2	33.9
	1393	166.4	42.6
	1399	209.2	39.3
	1400	165.1	43.3
6	1231	6166.2	40.5
	1355	6143.7	134.8
	1370	575.2	30.5
	1393	297.2	71.2
	1399	161.6	30.3
	1400	155.6	31.2
7	1231	7034.0	55.3
	1355	7086.1	184.8
	1370	668.7	29.7
	1393	244.1	51.7
	1399	160.0	25.6
	1400	167.0	33.7
8	1231	7678.9	183.9
	1355	7802.3	125.0
	1370	823.7	212.1
	1393	234.6	29.5
	1399	192.0	30.8
	1400	173.6	23.8
9	1231	8937.4	460.2
	1355	8849.7	281.2
	1370	846.9	27.0
	1393	237.0	36.5
	1399	188.9	20.7
	1400	188.4	29.0

Table 5. The average execution time in ms. of triple generation as a function of the changes to the implementation.

s	d	λ	t	2	3	4	5	6	7	8	9
1	0	1	5	1.872	1.598	1.211	1.280	1.464	1.568	1.767	2.238
1	0	1	10	1.511	0.952	1.230	1.356	1.169	1.165	1.606	1.799
1	0	1	30	1.370	1.059	1.019	1.472	1.211	1.510	1.826	1.794
16	9	1	5	15.879	11.796	13.289	16.460	17.108	21.643	18.677	25.931
16	9	1	10	15.157	11.883	12.203	15.156	15.998	17.023	22.755	24.444
16	9	1	30	11.641	10.944	11.999	13.138	14.963	18.271	22.319	25.638
21	8	2	5	20.959	16.931	16.960	19.712	20.838	28.218	31.572	31.901
21	8	2	10	16.560	15.981	15.334	18.856	22.848	25.139	26.847	32.572
21	8	2	30	16.453	15.269	16.869	19.211	21.400	25.469	30.479	37.545

Table 6. The average execution time in seconds of random triple generation as a function of the security parameter and the number of triples.