Practically Efficient Secure Distributed Exponentiation without Bit-Decomposition

Abdelrahaman Aly, Aysajan Abidin, and Svetla Nikova

imec-COSIC KU Leuven,
Kasteelpark Arenberg 10, B-3001 Leuven-Heverlee, Belgium
{firstname.lastname}@esat.kuleuven.be

Abstract. Bit-decomposition is a powerful tool which can be used to design constant round protocols for bit-oriented multiparty computation (MPC) problems, such as comparison and Hamming weight computation. However, protocols that involve bit-decomposition are expensive in terms of performance. In this paper, we introduce a set of protocols for distributed exponentiation without bit-decomposition. We improve upon the current state-of-the-art by Ning and Xu [1,2], in terms of round and multiplicative complexity. We consider different cases where the inputs are either private or public and present privacy-preserving protocols for each case. Our protocols offer perfect security against passive and active adversaries and have constant multiplicative and round complexity, for any fixed number of parties. Furthermore, we showcase how these primitives can be used, for instance, to perform secure distributed decryption for some public key schemes, that are based on modular exponentiation.

1 Introduction

The use of Internet connected devices has become essential in people's daily lives. However, serious privacy concerns have been raised as more and more sensitive private information is transmitted (over the Internet), processed and stored in third party databases or the cloud. This movement of information has been fostered by decades of research in cryptography, helping to develop tools and techniques to perform computational tasks in a privacy-friendly manner. Secure multiparty computation (MPC) is one such tool that allows the computation of functions on private inputs by mutually non-trusted parties. The initial techniques and problems proposed in the seventies and eighties have indeed evolved into a growing field with not only theoretical results but into the implementation of practical applications (e.g., [3,4,5]). More recently, thanks to the advent of frameworks such as SPDZ [6] or MASCOT [7], it is possible to implement any functionality in a relatively *efficient* fashion.

One of the basic and commonplace problems in applied MPC is distributed modular exponentiation. Classical approaches for distributed modular exponentiation over arithmetic circuits rely on bit-decomposition, which was first proposed by Damgård et al. in [8]. However, bit-decomposition is an expensive procedure in terms of performance. Recently, Ning and Xu proposed protocols

for modular reduction and exponentiation without bit-decomposition in [1,2]. In this paper, we build upon [1,2], and present mechanisms to compute modular exponentiation in a simpler and practically efficient fashion. Our approach does not require exhaustive parallelization and has a constant round complexity. Finally, we show how our approach can be used in, for example, distributed decryption of public key protocols.

Bit-decomposition. The process of decomposing secretly held finite field elements into their bit-representations for exponentiation was originally introduced by Damgård et al. [8]. The same decomposition mechanism was then used not only to construct secure exponentiation mechanisms, but also as a basic building block for (in)equality tests and modular operations [9]. Although highly practical, the cost of operating bitwise on arithmetic circuits cannot be dismissed. An extended polynomial representation, such as a bit extension, implies a large bound on the amount of work i.e. multiplication performed.

It can be argued that in many cases involving bit-decomposition, the operations are not co-dependent and hence can be parallelized (low round complexity). Even if we put aside the increase in transmission costs due to increased number of shares to be transmitted, the fact that the total amount of work (amount of multiplications) depends on the input size, cannot be addressed only by parallelization. For example, 4096 concurrent multiplications would require a batching structure composed of several threads to compute the multiplications not only in a single round, but in an equivalent amount of CPU time. This would necessitate the need for developing practically efficient mechanisms for modular operations, such as exponentiation, without relying on bit-decomposition.

Related Work. As previously mentioned, initial alternatives focus on bit-decomposition. Namely, the work introduced by Damgård et al. [8], that offered security against both active and passive adversaries for all three cases that we consider in this paper. However, for the case concerning the publicly available exponent, certain adaptations are needed to achieve security against malicious adversaries. It also requires the production of additional randomness r and r^a, where a is the public exponent, which would require additional communication rounds. Our protocol gets rid of such requirements and offers security against malicious adversaries.

The other two exponentiation protocols introduced in [8] make use of bit-decomposition, which has the aforementioned performance drawbacks. None of our protocols require bitwise operations, and optimizes the process in a simple straight-forward fashion. The main advantage of our approach is the reduction of non-linear operations which are, in general terms, expensive [10], as well as the decoupling the protocol complexity from the input size.

Recently, Ning and Xu [1,2], introduced several protocols aimed to remove the necessity of decomposing secret inputs into their bit representations. They achieve this by using a series of constructions based on bitwise operations over some randomness. Their work achieves constant round, with a linear asymptotic

bound on the amount of work (on the size of input). We not only simplify their process, but also reduce round complexity making use of constant amount of multiplications.

On a similar line, Grassi et al. introduced a mechanism to perform secure exponentiation over a publicly available base to a secret shared exponent in [11]. Their protocol was used in the context of the implementation of symmetric key primitives. Their protocol, however, was designed such that the output of the protocol is disclosed to the computational parties every time. Our protocols, on the other hand, allow the parties to keep the output secret and produce the output only when needed.

Note that, although the input size is not a factor in any of our protocols' complexity, this is not the case for the number of parties performing the computation. However, it is safe to assume this number remains unaltered throughout common practical applications giving room to specialized frameworks for a constant number of parties; see, e.g., [3,5,12,13]. This is not the case for the input size, given their variability in practical applications.

Our Contribution. We propose distributed modular exponentiation protocols without bit-decomposition, for the following configurations:

- **Public Base:** the base is public and the exponent is secret;
- **Public Exponent:** the base is private and the exponent is public; and
- **Private Exponentiation:** both the base and the exponent are privately held.

Note that, since the output of these functionalities would remain secret, our protocols can be used as sub-routines for more complex functionalities. This is indeed congruent with the literature on this topic e.g., [1,2], without prejudice to its security under composition [14]. Our protocols outperform their most recent counterparts in terms of round complexity and amount of work, which in our case are both constant for a fixed number of parties. Table 1 contrasts our results with the results by Ning and Xu [2]. Note that in this context l stands for the bit-size of the input and n for the number of parties.

As an application, we explore the case where decryption of common public key encryption schemes (such as RSA [15] and El-Gammal [16]) performed over MPC. The scenario that we consider is as follows: a user encrypts a message using a public key pk, corresponding to a private key sk, such that pk is publicly available and sk is secretly shared among several parties. An MPC decryption would not only facilitate what is commonly called threshold decryption, a powerful technique used for example in voting systems [17], but also transforms any ciphertext into secretly held shares of the message.

Table 1. Round/Multiplication complexity for modular exponentiation without bit-decomposition.

Protocol	Semi-Honest [2]		Semi-Honest(this work)	
Protocol	Rounds	Multiplications	Rounds	Multiplications
$\texttt{exp}([b], a)$	5	$33\ (10 \cdot n + 3)$	8	$8\ (4 + 2 \cdot \lfloor log(n) \rfloor)$
$\texttt{exp}(b, [a])$	N/A	N/A	3	$3\ (1 + \lfloor log(n) \rfloor)$
$\texttt{exp}([b], [a])$	20	$157 \cdot l + 7 \cdot l + 7 + 5$	13	$13\ (7 + 3 \cdot \lfloor log(n) \rfloor)$

Outline. This work is organized as follows: We introduce the necessary background material in Section 2. We then give a detailed description of our results on exponentiation in Section 3. We provide an overview on the usage of our protocols for public key decryption in Section 4. Section 5 concludes the paper.

2 Preliminaries

2.1 Notation

We follow the square notation introduced in [18]. Let $[x]$ denote a secretly shared input x. Let P be the set of n parties, P_i be the i-th party for $i = 1, \cdots, n$ and q be a large prime number. Let $\mathbb{Z}_q^*$ be the multiplicative group $\mathbb{Z}_q - \{0\}$. To distinguish secretly shared elements of $\mathbb{Z}_q$ from that of $\mathbb{Z}_q^*$, we use $[x]_q$ for $x \in \mathbb{Z}_q$ and $[x]_{q^*}$ for $x \in \mathbb{Z}_q^*$. Additionally, consider $p = q - 1$, such that we can define $\mathbb{Z}_p$ and $\mathbb{Z}_p^*$ accordingly. Furthermore, our protocols make use of the infix notation, to make calls to the secure functionality e.g. $[z] \leftarrow [x] + [y]$. Note that in practice, basic operations are carried out by the underlying MPC protocol e.g., [13,19,20,6,7]. Our exponentiation protocols do not make direct use of any specific representation for signed values. However, other protocols such as the *modular* operation used in later sections may need such support definitions.

To denote a shared value x we use $[x] \leftarrow \texttt{Share}(x)$. To denote open or reconstruct the shared value we use $x \leftarrow \texttt{Open}([x])$. Note that for simplicity, we assume, in this work, that the inputs have been secretly shared using Shamir's Secret Sharing Scheme [21] which uses public constants α_i (coefficients of a Lagrange polynomial) in the reconstruction of the secret. To be specific, $x \leftarrow \texttt{Open}([x])$ is computed as $x = \sum_{i=1}^{|P|} \alpha_i \cdot x_i$, where x_i is the i-th party's share. Fan-in multiplication of shared secrets resulting in the sharing of their product $[c] \leftarrow \texttt{Product}([a], [b])$. Similarly inversion (in the considered finite field) of a shared value resulting in the sharing of the inverse value $[x^{-1}] \leftarrow \texttt{Inverse}([x])$.

2.2 Complexity Metrics

Typically, complexity is measured by the number of non-concurrent operations executed by the parties in charge of executing the functionality. We follow the relevant literature in the field, and differentiate between the operations that require any level of communication exchange in between the participants and those

operations that can be executed locally. Indeed, following [10], we consider that the linear operations such as additions or scalar multiplications to be of negligible cost, given they do not require any message exchange and hence, are *free*. On the other hand, non-linear operations, such as multiplications, require such exchanges and are substantially more expensive. Furthermore, we define round complexity as the number of sequential (non-parallelizable) invocations to functionality that requires at least one communication round i.e. the multiplicative depth of an arithmetic circuit. However, this is not the only metric that we use, the number of multiplications and equivalent operations, also referred to as the amount of work holds great importance as well, given that it would determine the volume of the information exchanged.

2.3 Secure Multiparty Computation

Secure multi-party computation lets us compute any functionality that can be composed into either an arithmetic or boolean circuit, among any number of mutually distrustful parties. Moreover, notable results known as BGW [19] and CCD [22] prove that any functionality can be calculated with perfect security, as long as a majority of players remained honest for the semi-honest case, and two thirds for the malicious case, thanks to the use of Verifiable Secret Sharing (VSS).

More recent results have focus on offering security against malicious adversaries in the presence of dishonest majorities e.g., [6,20,7], at the cost of an offline phase that provides cryptographic security. Furthermore, sub-protocols implementing functions can be securely composed thanks to their Universal Composability (UC) security [14]. Indeed, we make use of well known mechanisms designed to work on MPC settings, namely the following:

Secure comparison. We specifically refer to equality and inequality tests for integer ring elements. Several protocols for secure comparison have been introduced by the literature, designed under the same model used by this paper achieving different security guarantees i.e. perfect security [8] or statistical security [23,24]. Such functionality can be further defined as follows:

$$[z] \leftarrow \texttt{Equal}([x],[y]) \quad ([x] \stackrel{?}{=} [y]) \quad \text{where} \quad [z] \in \{0,1\}, \tag{1}$$

$$[z] \leftarrow \texttt{Compare}([y],[y]) \quad ([x] \stackrel{?}{<} [y]) \quad \text{where} \quad [z] \in \{0,1\}. \tag{2}$$

Randomization. Protocols for efficient random number generation have been introduced by [25]. This work, commonly referred to as PRNG (Pseudo-Random Number Generation) introduced techniques to generate secret shared randomness at a negligible cost i.e., no communication cost associated to it. Furthermore all randomness can be easily pre-computed before the execution of any of our protocols. This includes the generation of the multiplication triples (see [6,7]) and any other process randomness needed across the underlying MPC protocols and our results.

Security. Security of MPC is typically defined following the Universal Composability (UC) framework [14,26], which is a general framework allowing arbitrary MPC protocols to be represented and analyzed. Informally speaking, in the UC framework, for every real protocol an ideal functionality is first defined that may include a trusted third party which securely interacts with all involved players, performs computations on the players' private inputs and distributes the output to the parties. That is, for every real world protocol an ideal functionality $\mathcal{F}$ must be defined, which takes all the inputs from the players and performs the desired computation in an ideal way. Then the real protocol is said to be secure if whatever can be done in the real world by an adversary can also be simulated in the ideal world by an (ideal-world) simulator. Hence, UC security of a protocol ensures that the security provided by the ideal functionality is not stronger than that provided by the real protocol. This is done by introducing an environment $\mathcal{Z}$. The environment $\mathcal{Z}$ chooses all the inputs to every involved party, receives all outputs, and communicates freely with the adversary $\mathcal{A}$ throughout the entire protocol run. Basically, in the ideal case, adversary $\mathcal{A}$ is replaced by a simulator $\mathcal{S}$, which internally simulates $\mathcal{A}$ and acts as a buffer between $\mathcal{Z}$ and $\mathcal{F}$.

The ideal functionality for MPC is modeled by *arithmetic blackbox (ABB)* [18]. ABB allows the players (or users) to provide input/output values that are to be secret shared and that performs arithmetic operations on the values of the secret shares over a finite field, say $\mathbb{Z}_q$. It can be thought of as a generic procedure for secure computation. Any party (or parties) can send its (or their) private input to ABB and ask it to compute any computable function. The computation results are stored in the internal state of ABB so that they can be used in the subsequent computations. Stored values can only be made public if majority of the players agree on it. ABB provides us with abstraction of the details of MPC operations and of secret sharing. The ABB functionality is defined as follows.

Definition 1 (ABB Functionality $\mathcal{F}_{ABB}$). *The ideal functionality $\mathcal{F}_{ABB}$ for MPC, where $\nu \in \{q, q-1\}$, is defined as follows:*

- `Input`*: Receive a value $x \in \mathbb{Z}_\nu$ or x from some party and store x.*
- `Share`(x)*: Create a share $[x]$ of x.*
- `Product`$([x],[y])$*: Compute $z = x \cdot y$ and store $[z]$.*
- `Compare`$([x],[y])$*: Compare x and y, and return 0 if $x < y$ and 1 otherwise.*
- `Equal`$([x],[y])$*: Check if $x = y$; return 1 if $x = y$, 0 otherwise.*
- `sRand`$(\mathbb{Z}_\nu)$*: Sample $r \xleftarrow{R} \mathbb{Z}_\nu$ and store $[r]$.*
- `Open`$([x])$*: Send the value x to all players.*

Addition and scalar multiplication are denoted by their corresponding conventional symbols $+$ and $\cdot$.

Definition 2 (UC-security [14]). *A real protocol π is UC-secure if, for all adversaries $\mathcal{A}$, there exists a simulator $\mathcal{S}$ for which no environment $\mathcal{Z}$ can distinguish with a non-negligible probability if it is interacting with $\mathcal{A}$ and players running π or $\mathcal{S}$ and players using the ideal functionality $\mathcal{F}$.*

As we shall see later, our protocols for exponentiation offer perfect security against active and passive adversaries.

2.4 Exponentiation based on Bit-Decomposition

Currently, the literature offers mechanisms to solve exponentiation, using bit-wise decomposition techniques. Initially, work introduced by Damgård et al. [8], showed how to achieve this with perfect security. However, as mentioned before, the cost related to the bit-decomposition of the inputs is relatively high, and has been seen as restrictive by other works in the field e.g. [2,24]. For instance, bit-decomposition protocols, such as the one described by [8], require bit-wise addition. A naive implementation of a carry for the addition would require at least as many sequential multiplications as the bit-wise input size. The costs related to the random bit generation have to be considered as well. In this section, we give a more detailed view and explanation on the Damgård et al. methods for exponentiation. The protocols follow the same notation as the rest of the paper. Furthermore, we assume the existence of the function $[x]_{bits} \leftarrow \texttt{bd}([x])$, which receives a secret shared input $[x]$ and return its shared bit-decomposition $[x]_{bits}$.

Exponentiation protocol to a public value: This is achieved by executing a fan-in multiplication in $\mathbb{Z}_q$ as expressed by the following equation

$$[b^a] \leftarrow \texttt{Product}_{i=1}^{a}([b]). \tag{3}$$

This naive approach provides perfect security for both passive and malicious cases. However, the process has its own drawback in complexity terms, given that it directly depends on a i.e. $\mathcal{O}(a)$, in case no optimized implementation of the fan-in operation is used. Damgård et al. [8] introduced a more efficient procedure that can be executed in constant time. Protocol 1 shows its implementation.

Protocol 1: Secure Damgård et al. `exp([b],a)` operation to a public exponent

Input: secretly shared base $[b]$ in $\mathbb{Z}_{q^*}$, publicly available exponent a in $\mathbb{Z}_p$
Output: secret shared $[b^a]$

1 $[r], [r^a] \leftarrow \texttt{sRand}(\mathbb{Z}_q^*, a)$;
2 $[c] \leftarrow \texttt{Product}([b], [r])$;
3 $c \leftarrow \texttt{Open}([c])$;
4 $[b^a] \leftarrow (c^a) \cdot \texttt{Inverse}([r^a])$;

As its naive counterpart, it provides security against passive adversaries, but requires extra processing for the active case. Basically, during the generation of $[r]$ and $[r^a]$ an adversary could maliciously select the share corresponding to $[r^a]$. We invite the reader to revise [8] to explore the malicious case.

Exponentiation protocol with a public base: To achieve this, the authors propose securely bit-decomposing the inputs (`bd`). In this case, the protocol decomposes the secretly shared exponent into bits, and uses fan-in multiplications of a specially crafted term to achieve the desired behavior. On complexity, this process delivers the results in a constant round complexity for fixed input sizes, but it linearly grows with the input size. Protocol 2 shows its implementation.

Protocol 2: Secure `exp(b,[a])` operation on a public base

Input: publicly available base b in $\mathbb{Z}_q$,secretly shared exponent $[a]$ in $\mathbb{Z}_p$, size of the inputs in bits l

Output: secret shared $[b^a]$

1 $[a]_{bits} \leftarrow \texttt{bd}([a])$; // $([a_0],[a_1],...,[a_{l-1}])$ `s.t.` $a_i \in \{0,1\}$

2 $[b^a] \leftarrow \texttt{Product}_{i=0}^{l-1}([a_i] \cdot b^{2^i} + [1] - [a_i])$;

Privacy preserving exponentiation: This case considers that both, base and exponent are secretly held, can be achieved in a similar fashion. In this case, a secure `exp([b],a)` should be used instead of the plain text base exponentiation step, as shown by Protocol 3.

On extending exponentiation for $\mathbb{Z}_q$: Damgård et al [8] presented an easy to implement technique to avoid revealing the secret $[b]$ when it is equal to 0, and hence extending this functionality to $\mathbb{Z}_q$ instead of $\mathbb{Z}_q^*$. This is achieved by simply adding the result of the following equality test $[b] \stackrel{?}{=} [0]$ to $[b]$ such that $b + ([b] \stackrel{?}{=} [0])$, and then subtracting the equality test at the end of the computation. This can be achieved in $log(l)$ rounds. This way the secret $[b]$ is not disclosed. The same can be applied to Protocol 1 and Protocol 3, to preserve the privacy of the inputs after being multiplicatively masked, hence we do not revisit this issue.

3 Secure Distributed Exponentiation

In this section, we explore different protocols for exponentiation based on publicly available and privately held inputs of any set of parties. Note that the protocols are designed to work for the case when the base b is secret shared in $\mathbb{Z}_q$ and its exponent a in $\mathbb{Z}^*{}_p$.

3.1 Public Base Exponentiation

Intuitively, this is the case where a publicly available base b is raised to a secretly shared exponent $[a]$. Its ideal functionality is defined as follows:

Protocol 3: Secure `exp([b],[a])` operation

Input: secretly shared base $[b]$ in $\mathbb{Z}_{q^*}$,secretly shared exponent $[a]$ in $\mathbb{Z}_p$, size of the inputs in bits l

Output: secret shared $[b^a]$

1 $[a]_{bits} \leftarrow \texttt{bd}([a])$; // $([a_0],[a_1],...,[a_{l-1}])$ s.t. $a_i \in \{0,1\}$

2 $[b^a] \leftarrow \texttt{Product}_{i=0}^{l-1}(\texttt{Product}([a_i],\texttt{exp}([\texttt{b}],2^{\texttt{i}})) + [1] - [a_i])$;

Definition 3. *Let base b and exponent a be elements of $\mathbb{Z}_q$ and $\mathbb{Z}*_p$, respectively, and let b be publicly available and a be privately preserved that is hosted by any subset of honest parties. The ideal functionality $\mathcal{F}_{\texttt{exp}(b,[a])}$ takes b and the secret $[a]_{p^*}$ as input and returns $[b^a]$ as output.*

Protocol 4 shows how this functionality can be achieved, for the semi-honest case. We further upgrade the construction in Protocol 4 to provide perfect security against active adversaries later in this section. Note that, without loss of generality, we assume the inputs have been secretly shared, using Shamir's scheme [21], where α_i are the publicly available interpolation coefficients as it was previously described.

Protocol 4: Secure `exp(b,[a])` operation with public base

Input: publicly available base b in $\mathbb{Z}_q$, secret shared exponent $[a]$ in $\mathbb{Z}_{p^*}$

Output: secret shared $[b^a]$

1 each party P_i locally computes $c_i \leftarrow b^{\alpha_i \cdot a_i}$;

2 $[c_i]_q \leftarrow \texttt{Share}(c_i)$;

3 $[b^a]_q \leftarrow \texttt{Product}_{i=1}^{|P|}([c_i]_q)$;

Protocol 4 returns $b^{[a]_{p^*}}$ by directly reconstructing the shares of $[a]_{p^*}$ on the exponent. This requires every party P_i to multiply locally its share a_i by its corresponding α_i constant. Let us analyze Shamir's scheme, for instance: parties should make use of the Lagrange interpolation multipliers (which are publicly held constants). Note that for the case of additive secret sharing, it suffices for the parties to apply the share of the exponent. Parties then proceed to calculated c_i locally, secret share it and multiply the resulting shares, finally obtaining $b_{p^*}^{[a]}$ as a result. This is a standard and common technique used for threshold decryption, for instance in voting systems e.g., [27], or more recently by some symmetric key techniques over MPC [11].

However, one issue becomes immediately obvious - the protocol, as described, cannot offer security for the malicious case. Protocol 4 description is secure against passive adversaries, that is because each party can choose what its share as c_i. The protocol, nonetheless can be extended to provide malicious security at the cost of adding communication rounds.

Malicious Case Let us first explore the basic naive approach to achieve malicious security: First, we calculate the result provided by the $\texttt{exp}(b, [a]_{p^*})$ functionality, and then we compute $\texttt{exp}(b, [a']_{p^*})$ for $[a']_{p^*}$ which is the product of $[a]_{p^*}$ with some randomness $[r]_{p^*}$. We then just verify both calculated inputs are equal. We show how to achieve this in Protocol 5.

Protocol 5: Secure `exp(b,[a])` operation with public base against malicious adversaries

Input: publicly available base b in $\mathbb{Z}_q$, secret shared exponent $[a]$ in $\mathbb{Z}_{p^*}$
Output: secret shared $[b^a]$

1 $[b^a]_q \leftarrow \texttt{exp}(b, [a]_{p^*})$;
2 $[r]_{p^*} \leftarrow \texttt{sRand}(\mathbb{Z}_p^*)$;
3 $[a']_{p^*} \leftarrow \texttt{Product}([r]_{p^*}, [a]_{p^*})$;
4 $[b^{a'}]_{p^*} \leftarrow \texttt{exp}(b, [a']_{p^*})$;
5 $r \leftarrow \texttt{Open}([r]_{p^*})$;
6 $[r']_q \leftarrow \texttt{sRand}(\mathbb{Z}_q^*)$;
7 $[v]_q \leftarrow \texttt{Product}([r'], \texttt{exp}([b^a]_q, r) - [b^{a'}]_q) + [1]$; // `correct if` $[v] == 1$
8 $v_q \leftarrow \texttt{Open}([v]_q)$;

A similar process can be implemented by means of performing additional calls to any semi-honest $\texttt{exp}(b, [a]_{p^*})$ functionality, including ours. To achieve this we take a somewhat different approach as shown in Protocol 6, in the sense that we *sacrifice* randomness, and use b as the base of every call to $\texttt{exp}(b, [a]_{p^*})$, so that we can arithmetically operate over the exponents for verification.

Protocol 6: Secure $\texttt{exp}^+\texttt{(b,[a])}$ operation with public base against malicious adversaries

Input: publicly available base b in $\mathbb{Z}_q$, secret shared exponent $[a]$ in $\mathbb{Z}_{p^*}$
Output: secret shared $[b^a]$

1 $[b^a]_q \leftarrow \texttt{exp}(b, [a])$;
2 $[r]_{p^*} \leftarrow \texttt{sRand}(\mathbb{Z}_p^*)$;
3 $[a']_{p^*} \leftarrow \texttt{Product}([a]_{p^*}, [r]_{p^*} - [1])$;
4 $[b^{a'}]_q \leftarrow \texttt{exp}(b, [a']_{p^*})$;
5 $[w]_{p^*} \leftarrow \texttt{Product}([a]_{p^*}, [r]_{p^*})$;
6 $w \leftarrow \texttt{Open}([w]_{p^*})$;
7 $[r']_q \leftarrow \texttt{sRand}(\mathbb{Z}_q^*)$;
8 $[v]_q \leftarrow \texttt{Product}([r']_q, \texttt{Product}(\texttt{Product}([b^a]_q, [b^{a'}]_q), b^{-w}) - [1]) + [1]$; // `correct if` $[v] == 1$
9 $v \leftarrow \texttt{Open}([v]_q)$;

Given that all factors being multiplied are powers of the same base b, protocol 6 then performs the following operation: $[a]_{p^*} + \texttt{Product}([a]_{p^*}, [r]_{p^*} - [1]) - [w]$.

This in turn, translates to $[a]_{p^*} + \texttt{Product}([a]_{p^*}, [r]_{p^*} - [1]) - \texttt{Product}([a]_{p^*}, [r]_{p^*})$. Therefore we can correctly validate if the intermediate calls to the $\texttt{exp}(b, [a]_{p^*})$ functionalities are correct. Succinctly speaking $[v]$ would be 1 if, and only if integrity was maintained. Note that no other information is leaked by means of applying $[r']_q$ to the output, this is also true for Protocol 5. Note that the protocol keeps the basic structure of its naive counterpart.

Security. Let $\pi_{\texttt{exp(b,[a])}}$ be the Protocol 6. Then the ideal functionality $\mathcal{F}_{\texttt{exp(b,[a])}}$ for the procedure `exp(b,[a])` exactly $\mathcal{F}_{ABB}$ extended with $\texttt{exp}(r, [e])$. This means that $\pi_{\texttt{exp(b,[a])}}$ uses only the MPC operations provided by $\mathcal{F}_{ABB}$. Therefore, it is straightforward to see that $\pi_{\texttt{exp(b,[a])}}$ is secure. Formally, we have the following.

Theorem 1. *The protocol $\pi_{exp(b,[a])}$ securely implements $\mathcal{F}_{exp(b,[a])}$ in the $\mathcal{F}_{ABB}$ framework.*

Proof. Since $\mathcal{F}_{\texttt{exp(b,[a])}}$ is the same as $\mathcal{F}_{ABB}$, the security of $\pi_{\texttt{exp(b,[a])}}$ inherits the security of the MPC operations in $\mathcal{F}_{\texttt{ABB}}$. □

3.2 Public Exponent Case

We now present our constant time secure exponentiation protocol for the case when the base b is privately held and the exponent a is public. The results on this section make use of the $\texttt{exp}(b, [a]_{p^*})$ functionality introduced by this work, however, any other mechanism that implement such functionality could be used instead. Furthermore, we assume $\texttt{exp}(b, [a]_{p^*})$ can be realized providing perfect security against semi-honest and malicious adversaries. This case ideal functionality can be defined as follows:

Definition 4. *Let base b and exponent a be elements of $\mathbb{Z}_q$ where $[b]$ is privately preserved, a is publicly available and hosted by any subset of honest parties. The ideal functionality $\mathcal{F}_{\texttt{exp}([b],a)}$ retrieves the secret b and $[a]$ and returns to the adversary $[b^a]$.*

We show how to implement our $\texttt{exp}([b], a)$ functionality in Protocol 7, note that it directly provides security against passive and active adversaries.

The protocol itself works as follows: Parties agree on a unique generator g and some secret shared randomness $[\bar{r}]_q$. The protocol, then makes use of $[\bar{r}]_q$ to mask the base $[b]_q$ by calculating $[c]_q = \texttt{Product}([\bar{r}]_q, [b]_q)$. Note that $[c]_q$ is then immediately made public. The protocol proceeds to compute $[c^a]_q$ by means of calling $\texttt{exp}(b, [a]_{p^*})$ functionality. Finally, the protocol is able to construct the expression $[r'] \cdot a - [r'] \cdot a$ as the exponent of an $[r]$ whilst multiplying $[b^a]$, which is equivalent to $\texttt{Product}([r^0], [b^a]) = [b^a]$.

Protocol 7: Secure `exp([b],a)` operation with public exponent

Input: secret shared base $[b]$ in $\mathbb{Z}_q$, public available exponent a in $\mathbb{Z}_{p^*}$
Output: secret shared $[b^a]$

1 $g \leftarrow \texttt{getGenerator}(\mathbb{Z}_q)$; $[r']_{p^*} \leftarrow \texttt{sRand}(\mathbb{Z}_{p^*})$;
2 $[\bar{r}]_q \leftarrow \texttt{exp}(g, [r']_{p^*})$; // $g^{r'}$
3 $[c]_q \leftarrow \texttt{Product}([\bar{r}]_q, [b]_q)$;
4 $c \leftarrow \texttt{Open}([c]_q)$;
5 $c' \leftarrow c^a$; // $c' = g^{[r']\cdot a} \cdot [b]^a$
6 $[e]_{p^*} \leftarrow -a \cdot [r']_{p^*}$;
7 $[b^a]_q \leftarrow c' \cdot \texttt{exp}(g, [e]_{p^*})$; // $c' \cdot r^e$

Security. Let $\pi_{\texttt{exp([b],a)}}$ be the protocol described in Protocol 7. Then the ideal functionality $\mathcal{F}_{\texttt{exp([b],a)}}$ for the procedure `exp([b],a)` is $\mathcal{F}_{ABB}$ extended with $\texttt{exp}(b, [a])$, which is described by Protocol 6. But $\mathcal{F}_{\texttt{exp}}(b, [a])$ is the same as $\mathcal{F}_{ABB}$, hence $\mathcal{F}_{\texttt{exp([b],a)}}$ is also the same as $\mathcal{F}_{\texttt{ABB}}$. Therefore, the security of this protocol is also straightforward.

Theorem 2. *The protocol $\pi_{exp([b],a)}$ securely implements $\mathcal{F}_{exp([b],a)}$ in the $\mathcal{F}_{ABB}$ framework.*

Proof. The same as the proof of Theorem 1. □

3.3 Privacy Preserving Exponentiation

We now introduce our constant time protocol for secure exponentiation given a privately held base b and exponent a. Its ideal functionality can be expressed as follows:

Definition 5. *Let base $b \in \mathbb{Z}_q$ and exponent $a \in \mathbb{Z}_{p^*}$ be privately preserved and hosted by any subset of honest parties. The ideal functionality $\mathcal{F}_{\texttt{exp}([b],[a])}$ takes $[b]_q$ and $[a]_{p^*}$ as input and returns $[b^a]$ as output.*

Our construction assumes that `exp(b,[a])` functionality is available. Our construction, showcased by protocol 8, offers perfect security against both, passive and active adversaries.

In this case, we make use of many of the mechanisms introduced by Protocol 7 with some basic dissimilarities. Basically, to obtain $[c^a]_{p^*}$, the protocol makes use of the secure $\texttt{exp}(b, [a])$ functionality. This way the protocol can obtain $[c']_q$, instead of c'. Despite this, protocol behaves just in the same way as described for Protocol 7. Note that also plain and scalar arithmetic operations are replaced by their equivalent privately preserving counterparts.

Security. Let $\pi_{\texttt{exp([b],[a])}}$ be the protocol described in Protocol 8. Then the ideal functionality $\mathcal{F}_{\texttt{exp([b],[a])}}$ for the procedure `exp([b],[a])` is again $\mathcal{F}_{ABB}$ extended with $\texttt{exp}(b, [a])$, the ideal functionality of which is nothing but $\mathcal{F}_{ABB}$. Hence, the security of $\mathcal{F}_{\texttt{exp([b],[a])}}$ is also straightforward.

Protocol 8: Secure `exp([b],[a])` operation with shared exponent and base

Input: secret shared base $[b]$ in $\mathbb{Z}_q$, and exponent $[a]$ in $\mathbb{Z}_{p^*}$
Output: secret shared $[b^a]$

1 $g \leftarrow \texttt{getGenerator}(\mathbb{Z}_q)$; $[r']_{p^*} \leftarrow \texttt{sRand}(\mathbb{Z}_{p^*})$;
2 $[\bar{r}]_q \leftarrow \texttt{exp}(g, [r']_{p^*})$ $\quad //g^{[r']}$
3 $[c]_q \leftarrow \texttt{Product}([\bar{r}]_q, [b]_q)$;
4 $c \leftarrow \texttt{Open}([c]_q)$;
5 $[c']_q \leftarrow \texttt{exp}(c, [a]_{p^*})$; $\quad // \; g^{[r']\cdot[a]} \cdot [b]^{[a]}$
6 $[e]_{p^*} \leftarrow -1 \cdot \texttt{Product}([r']_{p^*}, [a]_{p^*})$;
7 $[b^a]_q \leftarrow \texttt{Product}([c']_q, \texttt{exp}(g, [e]_{p^*}))$;

Theorem 3. *The protocol $\pi_{exp([b],[a])}$ securely implements $\mathcal{F}_{exp([b],[a])}$ in the $\mathcal{F}_{ABB}$ framework.*

Proof. The same as the proof of Theorem 1. □

Remark on security. As we can see, the security of our protocols follow directly from the actual security of the MPC operations in $\mathcal{F}_{ABB}$ achieved in practice. As we have mentioned in the previous section, seminal results such as BGW [19] and CCD [22] showed that any functionality can be achieved with perfect security, as long as a majority of the players are honest for the semi-honest case, and two thirds for the malicious case. Security against malicious adversaries in the presence of dishonest majorities can also be achieved at the cost of an offline computation phase [6,20,7]. Furthermore, sub-protocols (i.e., the MPC operations in $\mathcal{F}_{ABB}$) are UC secure, hence they can be securely composed.

3.4 Complexity

All of our protocols have constant round and multiplicative complexity with respect of the size of the input. Note that in all our protocols the amount of work and their multiplicative depth grows linearly with respect to the number of parties n. Given that the focus of industry and academia have been centered on developing and optimizing protocols for the 2-Party and 3-Party case e.g. [28,29,12,13], we have introduced in our complexity analysis both scenarios. It is worth notice that other protocols designed for n-parties such as [6,30] have been mainly used on either the 2 or 3 party scenario e.g. [31]. Table 2 showcases how such protocol complexities vary on these different scenarios. Note that round complexity (r.) or multiplicative depth is the same as the multiplicative complexity (amount of work) for all our protocols.

Although the multiplicative depth of the protocol is altered by the number of parties, its effect is constrained to the linear behavior we mentioned above. For the malicious case the asymptotic complexity of all protocols is the same i.e., $\mathcal{O}(log(n))$, albeit the function constants would be larger (protocols would require to use the $\texttt{exp}(b, [a])$ variant, that is secure against active adversaries,

Table 2. Protocol Complexity for Exponentiation Protocols

	Semi-Honest		
Protocol	2-P	3-P	n-P
$\texttt{exp}([b], a)$	$\mathcal{O}(1) \rightarrow 6$ r.	$\mathcal{O}(1) \rightarrow 8$ r.	$\mathcal{O}(log(n)) \rightarrow (4 + 2 \cdot \lceil log(n) \rceil)$
$\texttt{exp}(b, [a])$	$\mathcal{O}(1) \rightarrow 2$ r.	$\mathcal{O}(1) \rightarrow 3$ r.	$\mathcal{O}(log(n)) \rightarrow (1 + \lceil log(n) \rceil)$
$\texttt{exp}^{+}(b, [a])$	$\mathcal{O}(1) \rightarrow 2$ r.	$\mathcal{O}(1) \rightarrow 3$ r.	$\mathcal{O}(log(n)) \rightarrow (1 + \lceil log(n) \rceil)$
$\texttt{exp}([b], [a])$	$\mathcal{O}(1) \rightarrow 10$ r.	$\mathcal{O}(1) \rightarrow 13$ r.	$\mathcal{O}(log(n)) \rightarrow (7 + 3 \cdot \lceil log(n) \rceil)$

instead of its passive counterpart, which requires a proportionally larger amount of work).

3.5 Performance

To estimate our protocols performance, we consider that the cost of an atomic non-linear (or equivalent) operation is negligible (additions and scalar multiplications), given they do not depend on communication rounds. Given that our multiplicative complexity is equivalent to our round complexity for all our protocols, it follows that: we can easily project the computational time that our protocols need by measuring one multiplication. For instance, we have used the Ben-Or, Goldwasser and Wigderson [19] (BGW) protocol implementation described by [32] to test its multiplicative performance. The implementation is a `C++` self-contained library based on Number Theory Library (NTL) [33] designed for 3 parties in a semi-honest setting. For our estimations, we have averaged two million of multiplications (using Gennaro's protocol [34]) on a 64-bit server with 2*2*10-cores Intel Xeon E5-2687 at 3.1GHz where only the minimum 2 core pero process needed where used. The results yield a $2.08 \cdot 10^{-5}$ seconds time per each communication round consisting of a single multiplication. We can then extrapolate the computational time of our protocols as follows: *i*). $\texttt{exp}(a, [b])$ is $6.24 \cdot 10^{-5}$ seconds; *ii*). $\texttt{exp}([a], b)$ is $1.665 \cdot 10^{-4}$ seconds and; *iii*). $\texttt{exp}([a], [b])$ is $2.7 * 10^{-4}$ seconds. Similar estimations could be performed for other settings using any other suitable MPC protocol e.g., SPDZ [6,7].

4 Public Key Decryption

We now show how to use our primitives to build a particular secure distributed decryption scheme that can be utilized for threshold decryption. Under this scenario a private key sk is shared among n parties, and a publicly available ciphertext c (of a message m) has to be decrypted, such that the output of the decryption is a share under any linear secret sharing scheme used for MPC. In other words from c and $[\mathsf{sk}]$, we have to obtain $[m]$. This way we can use public key decryption as a subprotocol for any other MPC functionality. We explore such scenario for two basic and well known public key schemes, namely RSA [15], ElGamal [16]. We start by giving an overview about these protocols. Note that

all schemes described in this section are IND-CPA-secure. In all the protocols below, λ denotes the security parameter.

We stress that such distributed decryption can be of interest in applications, such as privacy-preserving biometric authentication. For instance, one can encrypt users' biometric templates and share the decryption key among different parties. Then, when a user wants to authenticate, the user provides an encrypted fresh biometric template, which is then compared with the stored template in a distributed fashion. At the end, the result indicating whether there is a match is jointly decrypted by the parties following the procedures that we present in this section. We plan to demonstrate this in the future.

4.1 RSA

The RSA encryption is based on the hardness of prime factorization of integers.

- KeyGen: The key generation algorithm takes a security parameter λ as input and outputs a public key $\mathsf{pk} = (n, e)$ and a private key $\mathsf{sk} = (p, q, d)$, i.e., $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(\lambda)$. Here and throughout forward, for the purposes of this section p and q are large distinct primes, $n = pq$, e is such that $\gcd(e, \phi(n)) = 1$, and d satisfies $de \equiv 1 \mod \phi(n)$, where $\phi()$ is the Euler's totient function.
- Enc: The encryption algorithm takes the public key $\mathsf{pk} = (n, e)$ and a message which is converted into a number $m < n$ as input and outputs a ciphertext $c \equiv m^e \mod n$; i.e., $c \leftarrow \mathsf{Enc}(\mathsf{pk}, m)$.
- Dec: The decryption algorithm takes $\mathsf{sk} = (p, q, d)$ and a ciphertext c as input and outputs a message $m \equiv c^d \mod n$; i.e., $m \leftarrow \mathsf{Dec}(\mathsf{sk}, c)$.

4.2 ElGamal

The ElGamal encryption is based on the hardness of discrete logs.

- KeyGen: The key generation algorithm takes a security parameter λ as input and outputs a public key $\mathsf{pk} = (G, g, q, h)$ and a private key $\mathsf{sk} = x$, i.e., $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KeyGen}(\lambda)$. This is done first by choosing a large prime p and a generator for the multiplicative group $G = \mathbf{Z}_p^*$, whose order is $q = p - 1$, and then by choosing a random $x \in G$ and computing $h \equiv g^x \mod p$.
- Enc: The encryption algirithm takes the public key $\mathsf{pk} = (G, g, p, h)$ and a message which is converted into an element $m \in G$ as input and outputs a ciphertext $c = (c_1, c_2)$, i.e., $c \leftarrow \mathsf{Enc}(\mathsf{pk}, m)$, by computing $c_1 \equiv g^y \mod p$ and $c_2 \equiv h^y m$, for a randomly chosen $y \in G$.
- Dec: The decryption algorithm takes $\mathsf{sk} = x$ and a ciphertext $c = (c_1, c_2)$ as input and outputs a message $m \equiv c_1^{-x} c_2 \mod p$; i.e., $m \leftarrow \mathsf{Dec}(\mathsf{sk}, c)$.

4.3 Privacy Preserving decryption

The data oblivious implementation of the decryption protocols of this public key schemes can be easily achieved by reusing both the exponentiation functionality

described in Section 3 and the modulo operation showcased above. In our case, We assume that the ciphertext is publicly available, whereas the private key is secret shared. The objective for us is to have a secret shared version of the message without leaking its content to any party involved in the computation.
Note that protocols to perform secure modulo operations for MPC, have indeed been extensively studied by the literature. We can name for instance the construction from the seminal paper by Damgård et al. [8]. In recent years, more efficient results although sometimes with more limited security capabilities e.g. statistical instead of perfect security, have been introduced as well e.g. [9].

RSA For this case, given that the decryption protocol $m \leftarrow \mathsf{Dec}(\mathsf{sk}, c)$, where $c \equiv m^e$, the computation of $[m]$ can be summarized by the computation of: $m = c^d = (m^e)^d \mod N$. Thus, with the functionality provided by this work it suffices for the parties computing the decryption to do $[m] \leftarrow \mathtt{exp}(c, [d]) \mod N$.

ElGamal The decryption protocol in this case consists on some basic operations on what we assume to be publicly available c_1 and c_2 values, as follows: $m = \frac{c_2}{c_1^x} \mod p$. To implement such protocol on MPC, where the sk $[x]$ is secretly shared we make use of the following: a scalar multiplication, our exponentiation method, the multiplicative inverse of such result, which can be computed in one round, and a secure mod operation: $[m] = c_2 \cdot \mathtt{Inverse}(\mathtt{exp}(c_1, [x])) \mod p$.

5 Conclusions

In this paper, we introduce secure mechanisms to perform exponentiation over MPC for arithmetic circuits without bit decomposition. Our protocols are simple and easy to follow mechanisms that have constant round/multiplicative complexity and offer security against semi-honest and malicious adversaries. Our protocols, besides being lean and simplified, improves the current state of the art for such kind of mechanisms. Additionally, we included a possible application for our techniques, in the form of public key decryption. Further work should explore the viability of these results on other related MPC fundamental applications such as comparisons. Another direction for future work would be to validate our protocols in practical applications such as in biometric settings, where templates need to be securely transmitted to some MPC based processing server, using PKI infrastructure.

Acknowledgements. This work was supported in part by the Research Council KU Leuven: C16/15/058. In addition, this work was supported by the European Commission through and H2020-ICT-2014-644371 WITDOM and the Flemish Government through the imec Distributed Trust program and through ICON Diskman. The authors would like to thank Prof. Nigel Smart and the anonymous reviewers for their comments and inputs.

References

1. Ning, C., Xu, Q.: Multiparty computation for modulo reduction without bit-decomposition and a generalization to bit-decomposition. In Abe, M., ed.: Advances in Cryptology - ASIACRYPT 2010: 16th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 5-9, 2010. Proceedings, Berlin, Heidelberg, Springer Berlin Heidelberg (2010) 483–500
2. Ning, C., Xu, Q.: Constant-rounds, linear multi-party computation for exponentiation and modulo reduction with perfect security. In Lee, D.H., Wang, X., eds.: Advances in Cryptology – ASIACRYPT 2011: 17th International Conference on the Theory and Application of Cryptology and Information Security, Seoul, South Korea, December 4-8, 2011. Proceedings, Berlin, Heidelberg, Springer Berlin Heidelberg (2011) 572–589
3. Bogetoft, P., Christensen, D.L., Damgård, I., Geisler, M., Jakobsen, T., Krøigaard, M., Nielsen, J.D., Nielsen, J.B., Nielsen, K., Pagter, J., Schwartzbach, M., Toft, T.: Secure multiparty computation goes live. In Dingledine, R., Golle, P., eds.: Financial Cryptography and Data Security: 13th International Conference, FC 2009, Accra Beach, Barbados, February 23-26, 2009. Revised Selected Papers, Berlin, Heidelberg, Springer Berlin Heidelberg (2009) 325–343
4. Bogdanov, D., Kamm, L., Laur, S., Sokk, V.: Rmind: a tool for cryptographically secure statistical analysis. IEEE Transactions on Dependable and Secure Computing (99) (2016) 1–1
5. Aly, A., Van Vyve, M.: Practically efficient secure single-commodity multi-market auctions. In Grossklags, J., Preneel, B., eds.: Financial Cryptography and Data Security: 20th International Conference, FC 2016, Christ Church, Barbados, February 22–26, 2016, Revised Selected Papers, Berlin, Heidelberg, Springer Berlin Heidelberg (2017) 110–129
6. Damgård, I., Pastro, V., Smart, N.P., Zakarias, S.: Multiparty computation from somewhat homomorphic encryption. In: CRYPTO. Volume 7417 of LNCS., Springer (2012) 643–662
7. Keller, M., Orsini, E., Scholl, P.: Mascot: Faster malicious arithmetic secure computation with oblivious transfer. In: Proceedings of ACM SIGSAC. CCS '16, ACM (2016) 830–842
8. Damgård, I., Fitzi, M., Kiltz, E., Nielsen, J.B., Toft, T.: Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In: TCC. (2006) 285–304
9. Catrina, O., de Hoogh, S.: Improved primitives for secure multiparty integer computation. In: SCN. (2010) 182–199
10. Cramer, R., Damgrd, I., Nielsen, J.: Secure Multiparty Computation and Secret Sharing. Cambridge University Press (2015)
11. Grassi, L., Rechberger, C., Rotaru, D., Scholl, P., Smart, N.P.: Mpc-friendly symmetric key primitives. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. CCS '16, New York, NY, USA, ACM (2016) 430–443
12. Bogdanov, D., Laur, S., Willemson, J.: Sharemind: A Framework for Fast Privacy-Preserving Computations. In: ESORICS. Volume 5283 of LNCS., Springer (2008)
13. Araki, T., Furukawa, J., Lindell, Y., Nof, A., Ohara, K.: High-throughput semi-honest secure three-party computation with an honest majority. In: Proceed. of the ACM SIGSAC. (2016) 805–817

14. Canetti, R.: Security and composition of multiparty cryptographic protocols. Journal of Cryptology (Jan 2000) 143–202
15. Rivest, R.L., Shamir, A., Adleman, L.: A method for obtaining digital signatures and public-key cryptosystems. Communications of the ACM (1978) 120–126
16. ElGamal, T.: A public key cryptosystem and a signature scheme based on discrete logarithms. In Blakley, G.R., Chaum, D., eds.: Advances in Cryptology: Proceedings of CRYPTO 84, Berlin, Heidelberg, Springer Berlin Heidelberg (1985) 10–18
17. Szepieniec, A., Preneel, B.: New techniques for electronic voting. Number 809 (2015) 30
18. Damgård, I., Nielsen, J.B.: Universally composable efficient multiparty computation from threshold homomorphic encryption. In: CRYPTO. Volume 2729 of LNCS., Springer (2003) 247–264
19. Ben-Or, M., Goldwasser, S., Wigderson, A.: Completeness theorems for non-cryptographic fault-tolerant distributed computation. In: STOC, ACM (1988) 1–10
20. Bendlin, R., Damgård, I., Orlandi, C., Zakarias, S.: Semi-homomorphic encryption and multiparty computation. In Paterson, K.G., ed.: Advances in Cryptology – EUROCRYPT 2011: 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tallinn, Estonia, May 15-19, 2011. Proceedings, Berlin, Heidelberg, Springer Berlin Heidelberg (2011) 169–188
21. Shamir, A.: How to share a secret. Commun. ACM **22**(11) (1979) 612–613
22. Chaum, D., Crépeau, C., Damgård, I.: Multiparty unconditionally secure protocols. In: STOC, ACM (1988) 11–19
23. Catrina, O., de Hoogh, S.: Secure multiparty linear programming using fixed-point arithmetic. In: ESORICS. (2010) 134–150
24. Lipmaa, H., Toft, T.: Secure equality and greater-than tests with sublinear online complexity. In: ICALP (2). (2013) 645–656
25. Cramer, R., Damgård, I., Ishai, Y.: Share conversion, pseudorandom secret-sharing and applications to secure computation. In Kilian, J., ed.: Theory of Cryptography. Volume 3378 of LNCS. Springer Berlin Heidelberg (2005) 342–362
26. Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. In: FOCS '01. (2001) 136–145
27. Peeters, R., Nikova, S., Preneel, B.: Practical rsa threshold decryption for things that think. In: 3rd Benelux Workshop on Information and System Security. (2008)
28. Goldreich, O., Micali, S., Wigderson, A.: How to play any mental game or a completeness theorem for protocols with honest majority. In: STOC, ACM (1987) 218–229
29. Huang, Y., Evans, D., Katz, J., Malka, L.: Faster secure two-party computation using garbled circuits. In: USENIX Security Symposium. (2011)
30. Damgård, I., Keller, M., Larraia, E., Pastro, V., Scholl, P., Smart, N.P.: Practical covertly secure mpc for dishonest majority or: Breaking the SPDZ limits. In: ESORICS. Volume 8134 of LNCS. Springer (2013) 1–18
31. Damgrd, I., Damgrd, K., Nielsen, K., Nordholt, P.S., Toft, T.: Confidential benchmarking based on multiparty computation. Cryptology ePrint Archive, Report 2015/1006 (2015) `http://eprint.iacr.org/2015/1006`.
32. Aly, A.: Network Flow Problems with Secure Multiparty Computation. PhD thesis, Universté catholique de Louvain, IMMAQ (2015)
33. Nishide, T., Ohta, K.: Multiparty computation for interval, equality, and comparison without bit-decomposition protocol. In: PKC. (2007) 343–360
34. Gennaro, R., Rabin, M.O., Rabin, T.: Simplified vss and fast-track multiparty computations with applications to threshold cryptography. In: PODC, ACM (1998)