A New Approach to Efficient and Secure Fixed-point Computation
(Full version)

Tore Kasper Frederiksen[1], Jonas Lindstrøm[2], Mikkel Wienberg Madsen[3], and Anne Dorte Spangsberg[3]*

[1] Zama, France
[2] Mysten Labs
[3] The Alexandra Institute, Denmark
tore.frederiksen@zama.ai, jonas@mystenlabs.com, mikkel.wienberg@alexandra.dk, a.d.spangsberg@alexandra.dk

Abstract. Secure Multi-Party Computation (MPC) constructions typically allow computation over a finite field or ring. While useful for many applications, certain real-world applications require the usage of decimal numbers. While it is possible to emulate floating-point operations in MPC, fixed-point computation has gained more traction in the practical space due to its simplicity and efficient realizations. Even so, current protocols for fixed-point MPC still require computing a *secure* truncation after *each* multiplication gate. In this paper, we show a new paradigm for realizing fixed-point MPC. Starting from an existing MPC protocol over arbitrary, large, finite fields or rings, we show how to realize MPC over a *residue number system* (RNS). This allows us to leverage certain mathematical structures to construct a secure algorithm for efficient approximate truncation by a static and public value. We then show how this can be used to realize highly efficient secure fixed-point computation. In contrast to previous approaches, our protocol does not require any multiplications of secret values in the underlying MPC scheme to realize truncation but instead relies on preprocessed pairs of correlated random values, which we show can be constructed very efficiently, when accepting a small amount of leakage and robustness in the strong, covert model. We proceed to implement our protocol, with SPDZ [DPSZ12] as the underlying MPC protocol, and achieve significantly faster fixed-point multiplication.

Keywords: MPC, fixed-point, malicious security, covert security, UC, residue number systems

1 Introduction

Secure multi-party computation (MPC) is the area of cryptography concerned with the computation of arbitrary functions over private data held by mutually distrusting parties. Since its inception by Andrew Yao [Yao82], it has received a large amount of research interest [GMW87, NO09, BDOZ11, DPSZ12, LP12, LOS14, KOS16, HOSS18]. Computation is usually represented as a Directed Acyclic Graph (DAG) of 2-input, 1-output gates where each gate operates on the inputs, most commonly over bits [LOS14, NO09, LP12, HOSS18] (*boolean MPC*), large fields [BDOZ11, DPSZ12] or rings [CDE$^+$18](*arithmetic MPC*). Even though operations in an arithmetic MPC scheme over a large field are typically less efficient than operations in an MPC scheme over bits, they are significantly more efficient than a corresponding emulation by an MPC scheme over bits. Furthermore, arithmetic MPC computations are required in a plethora of different applications, for example in auctions [BCD$^+$09], benchmarks [DDN$^+$16], machine learning [DEF$^+$19], private smart contracts [BCT21, BCDF22], private database querying [VSG$^+$19], threshold signatures [DOK$^+$20] and RSA key modulus generation with unknown factorization [FLOP18, CDK$^+$22]. However, in certain

* This work has received funding from the Alexandra Institute's performance contracts for 2021-24 with the Danish Ministry of Higher Education and Science and by Innovation Fund Denmark in Grand Solution CRUCIAL 1063-00001B. Tore and Jonas performed part of their work while at the Alexandra Institute. This preprint has not undergone peer review. The Version of Record of this contribution will be published in "*The 22nd International Conference on Applied Cryptography and Network Security (ACNS 2024)*", and will be available online at https://doi.org/[TBA]

applications, such as statistics [DA01, DDN$^+$16] and machine learning [LP00, WGC19, YSMN21, CGOS21] it is not sufficient to work over the integers, but instead require decimal numbers. While there are multiple ways of efficiently emulating decimal arithmetic using integers, doing so in MPC, without impacting efficiency significantly has proven elusive as we will discuss in the related works section below. Looking ahead, research in the area has pointed in the direction of fixed-point arithmetic, where integers are interpreted as decimal numbers with a static amount of digits after the decimal point, being the most efficient way of realizing secure decimal computation. This has the advantage that addition is exactly the same as for integer arithmetic, and multiplication is *almost* the same; in the sense that the multiplication operation is the same, but the result needs to be truncated with a constant. Such a truncation has proven expensive in MPC [CS10] and most researchers in that area [DSZ15, EGK$^+$20, RW19] have gone in the direction of trying to combine MPC over bits and integers to realize bit operations, such as truncation, efficiently. In this paper, we take a different direction and ask if it is 1) possible to efficiently realize fixed-point arithmetic in MPC without emulating bit operations and 2) if it is doable only assuming black-box access to an *arithmetic* MPC protocol. By representing integers in a *residue number system (RNS)* and accepting small additive errors in the truncation we answer both of these in the affirmative.

An RNS is a method for representing integers by their residues modulo a fixed set of coprime integers. This allows the representation of large integers using smaller integers, and it has received some attention in recent years because multiplication and addition may be done in parallel by the smaller integers, allowing large computations to be parallelized. This again can lead to advantages in high-performance computation [ST67, DCT$^+$18]. While our proposed scheme enjoys these advantages when preprocessing multiplication triples, it is the mathematical structure of an RNS we take advantage of to realize efficient truncation, and hence, efficient fixed-point computation.

Contributions In this paper we show how to realize secure computation over a 2-component RNS, resulting in secure computation over a biprime ring. We build this from two separate MPC instances, using any MPC scheme supporting computation over arbitrary prime fields. An example of such a scheme is SPDZ [DPSZ12], which relies on additive secret sharing of values, along with additively shared information-theoretic MACs for its underlying security against a statically and maliciously corrupted dishonest majority.

First, this provides us with an MPC system with more efficient preprocessing over very large domains, compared with the underlying MPC scheme used. This is advantageous in certain applications such as distributed generation of an RSA modulus [FLOP18, CDK$^+$22]. However, the main motivating factor behind our scheme is a highly efficient realization of secure *approximate truncation*, with a few bits of leakage of the least significant bits. Concretely we introduce an algorithm working over a maliciously UC-secure MPC functionality over arbitrary fields and show how this can be used to realize approximate truncation in a preprocessing model. Both our main online protocol and the offline preprocessing it relies on, only require a constant number of rounds of communication and *no* secure multiplications or other heavy cryptographic machinery. We prove both information-theoretically UC-secure based on black-box access to a functionality for maliciously secure MPC over a large ring. We show the *robustness* of this construction secure in the *strong covert* model [AL10], while retaining maliciously *privacy*, against an adversary corrupting a majority of parties. We highlight that we are not the first authors considering such a compromise between efficiency and security for MPC protocols [DKL$^+$13]. The cost of this efficient preprocessing is a small amount of leakage during the online protocol. Either $\log_2(n)$ bits or a single bit (if one can accept the addition of $O(n^2)$ extra multiplications during the online phase, where n is the number of parties). Furthermore, we note that due to an artifact of our simulation proof, we require the simulator to be allowed to perform $O(2^n)$ computations with low, but non-negligible probability. Hence, $O(2^n)$ will be polynomially bounded by any computational security parameter required to realize the underlying maliciously secure MPC functionality. Still, we highlight this has *no* influence on the actual efficiency of our protocols.

Based on this efficient preprocessing protocol and online phase we show how to construct an efficient fixed-point MPC scheme. Finally, we benchmark our scheme, based on SPDZ, through micro-benchmarks along with the versatile application of the Fast Fourier Transform. Depending on the size of the computation domain and network latency, our results show that the small reduction in security for approximate truncation allows us to get a 3.2-42x faster online phase. Suppose preprocessing is included (based on MASCOT [KPR18]) then our protocol is 36-1,400x faster than the protocol of Catrina and Saxena [CS10] for fixed-point computation, which also exclusively requires access to a black box MPC scheme. We furthermore find our protocol competitive for large domain computation, when counting the preprocessing time.

1.1 Related Work

Our work is far from the first usage of RNS in cryptography, although, to the best of our knowledge, this is the first time it is used to achieve efficient MPC for fixed-point arithmetic.

On the other hand, RNS has been used as a tool for optimizing the implementation of large field arithmetic, which is highly relevant in many public-key systems such as RSA [Qui82] or schemes based on general elliptic curves [AHK17] and isogenies [JMR22]. It has also been shown that such RNS-based implementations can help in thwarting certain types of fault-injection attacks [FPBS16]. The fact that RNS is an application of the Chinese Remainder Theorem (CRT) has in itself been used to optimize distributed RSA key generation [CCD$^+$20, DMRT21, CHI$^+$21].

As previously mentioned, working with decimal numbers is crucial in many MPC applications, such as statistics [DA01, DDN$^+$16] and machine learning [LP00, WGC19, YSMN21, CGOS21]. Many early works that require decimal numbers do not consider a formal treatment of how to represent decimal number systems in MPC but are instead rather pragmatic in how to achieve their specific computation goals. Examples include MPC computation of the natural logarithm through Taylor series expansion [LP00], or approximating $1/p$ through the use of Newton iteration [ACS02]. Du and Atallah [DA01] presented a custom two-party protocol for general secure division to implement linear regression in the two-party setting. However, Kiltz *et al.* [KLM05] showed that the protocol of Du and Atallah leaks some information, and demonstrated a leakage-free protocol for division in the two-party setting. Atallah *et al.* [ABL$^+$04], presented yet further protocols for realizing floating point division for use in secure linear regression, this time working with more than two parties. Fouque *et al.* [FSW03] showed how to do general computation over rationals, through the use of Paillier encryption. Although their scheme supports more than two parties, it only allows a limited number of consecutive operations. Another approach to floating point representation of values for two-party computation, based on Paillier encryption and oblivious PRFs (OPRFs), was given by Franz *et al.* [FDH$^+$10], who thoroughly formalized a system closely related to the IEEE 754 standard [iee19] used by regular CPUs for floating point arithmetic. Later a full implementation of secure evaluation of IEEE 754 was done by Franz and Katzenbeisser based on garbled circuits [FK11]. Boyle *et al.* [BCG$^+$21] showed how to achieve efficient fixed-point computation based on function secret sharing in the two-party model. Catrina and Saxena [CS10] showed how to realize secure fixed-point computation using *any* secret sharing-based MPC scheme for an arbitrary number of parties. Their approach is to do an efficient *approximate* truncation after each multiplication to move the decimal point back down. This approach, however, requires the computation of a bit-decomposed random number, which again requires $O(k)$ multiplications in MPC, when working over a field of at most 2^k elements. Besides the advantage of working over a generic MPC scheme with general inputs, the work by Catrina and Saxena also has the advantage of working over both positive and negative numbers and does not lose accuracy regardless of the number of computations. Later Catrina and de Hoogh [Cd10] showed how to realize this without the need for probabilistic computation in the truncation. Their idea is to compute a bit-comparison circuit in MPC using the bit decomposition of the random pad, which leads to a constant penalty in complexity. Unfortunately, all the approaches referenced above for secure computation with decimal numbers only work in the semi-honest security model. In many real-world situations, passive security is not sufficient. However, it turns out, that due to the black-box construction of the solution by Catrina and Saxena, using an MPC scheme that is maliciously secure is enough to realize their algorithms maliciously securely, as shown by Damgård *et al.* [DEF$^+$19]. Damgård *et al.* also showed that with only minor modifications, their probabilistic truncation algorithm could be used when the underlying MPC scheme performs computation over a ring.

Motivated by the need for bit-decomposition for truncation (and more general computation over bits) a series of works have been focused on efficiently combining MPC schemes over bits and fields/rings [DSZ15, MR18, MRVW21], Being able to perform a mix of arithmetic and bit operations can prove essential in certain applications, e.g. when computing the digest of a hash function in MPC. Dedicated schemes with a preprocessing phase generating raw material that works in both the binary and arithmetic domains have also been constructed [RW19, EGK$^+$20], yielding significantly more efficient protocols in practice than the approach of Catrina and Saxena.

Even without the recent protocols for efficient bit-decomposition, researchers have found secure fixed-point computations to generally be more efficient than secure floating-point computation [ABZS13, KLR16]. This is due to the complex computations needed for floating-point multiplications, which would be required to be carried out in MPC, compared to a single truncation in the fixed-point approach.

Finally, it should be mentioned that recently a lot of other authors have investigated MPC over rings [CDE$^+$18, ADEN21, DEN22], although their focus has generally been on arbitrary rings, or rings of the type $\mathbb{Z}_{2^k}$, as these afford a computation domain closer to traditional CPUs, when $k = 32$ or $k = 64$.

1.2 Construction Blueprint

The overall idea we present is to use two independent black-box realized MPC schemes, over $\mathbb{Z}_p$ and $\mathbb{Z}_q$, to realize an MPC scheme over $\mathbb{Z}_m$ with $m = p \cdot q$ by interpreting the elements in $\mathbb{Z}_p$ and $\mathbb{Z}_q$ as elements in an RNS over m, through the Chinese Remainder Theorem. Based on this we show that one can efficiently truncate elements in $\mathbb{Z}_m$ by p with an additive error of at most 1 without requiring any MPC multiplications, but only using sharings of correlated and uncorrelated randomness in both the MPC scheme over $\mathbb{Z}_p$ and $\mathbb{Z}_q$. Specifically, we need a random value $r \in \mathbb{Z}_p$ to be stored in an MPC scheme both over $\mathbb{Z}_p$ and $\mathbb{Z}_q$. We call a pair of such correlated random values a *noise pair* and show how to efficiently construct such pairs in the semi-honest and strong covert model but with an additive error in the sharing over $\mathbb{Z}_q$. We then use the RNS MPC scheme over $\mathbb{Z}_m$ to realize fixed-point computation in MPC with base p. By picking base p we can use our efficient truncation algorithm to perform the needed truncation after a fixed-point multiplication. We furthermore show that in this setting, the error in the noise pairs does not cause problems. Hence we circumvent the need for a general truncation of a 2-power, requiring a non-constant amount of multiplications, as is generally seen in fixed-point MPC computations [CS10, EGK$^+$20].

1.3 Outline

The rest of the paper is structured as follows: in section 2 we introduce the preliminaries we require, which concretely is the UC functionality for arithmetic MPC. Then in section 3 we present the main mathematical observation and abstractly show how it allows us to do divisions by p on elements stored in an MPC system over $\mathbb{Z}_m$, which is realized through two distinct MPC systems over $\mathbb{Z}_p$ and $\mathbb{Z}_q$. In section 4 we present the MPC protocols needed to realize the algorithm presented in section 3. In section 5 we formally show how to realize a UC-secure RNS MPC scheme with efficient truncation based on our construction. Finally, in section 6.1 we discuss an implementation of our scheme and benchmark it against the typical approach to fixed-point computation in MPC.

2 Preliminaries

In all our protocols we assume n mutually distrusting parties participate. We denote the set of all parties by $\mathcal{P}$ and use Adv to denote the adversary corrupting a subset of $\mathcal{P}$ of size at most $n-1$. We let s denote a statistical security parameter. Hence, the statistical distance between the real execution and simulation will be at most 2^{-s} for any fixed s. Furthermore, we define $\tilde{s} = s + 1$, which is an artifact used in the proofs to ensure 2^{-s} for the *entire* simulation. We let p and q be positive integers with $q > p \geq 2^s$ s.t. $\gcd(p, q) = 1$ and $m = p \cdot q$. We use $\sim$ in conjunction with standard distributions to denote "approximately distributed by" for certain variables. E.g. $x \sim \mathbf{IrwinHall}(n)$.

We let $a \leftarrow_R \mathbb{Z}_p$ mean that a is uniformly sampled from the ring $\mathbb{Z}_p$. Furthermore, in general, we use $a \leftarrow P$ to mean that the value a is computed by the procedure P. For integers y and $z > 0$, we let $y \bmod z$ denote the unique integer x for which $0 \leq x < z$ and $x \equiv y \pmod z$. Furthermore, throughout the paper, we use $[a]$ to denote the set of integers $\{1, \ldots a\}$.

We highlight that while most of our protocols are secure against a static adversary corrupting a dishonest majority of parties, our protocols for generating correlated randomness only achieve robustness in the *strong covert* [AL10] model. We recall that in this model the adversary may cheat but will get caught with a certain, non-negligible probability γ. Furthermore, if caught, the adversary gets no influence on the computation, nor learns anything about the honest parties' input, and all honest parties learn that cheating has occurred, along with the identification of one of the corrupt parties who cheated. If the adversary does *not* get discovered when cheating, then they learn the honest parties' input and get to influence the result of the computation. However, we emphasize, that our concrete protocols achieve even stronger security since the adversary does *not* learn the honest party's input, even if they cheat and don't get caught.

Table 1. The parameters and variables in play in this paper.

s	Statistical security parameter.
$\tilde{s}$	$\tilde{s} = s + 1$
γ	The covert deterrence factor.
p	Primes of at least $\tilde{s}$ bits. $p < q$.
q	
m	The RNS domain $= p \cdot q$.
n	The number of parties.
U	The maximum error.
Q	The maximum usable space modulo q.

We use U to denote the *maximum* additive error that can occur in our protocol. The error that can occur in our specific noise pair preprocessing will be at most $U - 1$, and our main algorithm, algorithm 5, adds 1 to any error of the preprocessed material.

We outline the different variables and their meaning in table 1.

2.1 UC Functionalities

We prove our construction secure in the UC framework [Can01] and hence no rewinding is used in our proofs and our protocol is secure under arbitrary composition.

In the ideal functionalities, we abstract away the session ID, `sid`, and simply assume that only a single instance of each subfunctionality is used for specific input parameters, since this is all we require for our protocol.

$\mathcal{F}_{\mathsf{CT}}(n, m)$

This functionality is parametrized by the number of parties participating, n, and a modulo describing the range of sampling.

Sample: Upon receiving $(\texttt{sample}, \texttt{ssid})$ from all parties, if `ssid` has not been seen before, then sample $x \leftarrow_R \mathbb{Z}_m$ and send $(\texttt{sample}, \texttt{ssid}, x)$ to all parties as adversarially delayed output. If `ssid` has been seen before, then return $(\texttt{sample}, \texttt{ssid}, x)$ as adversarially delayed output.

Fig. 2.1. Ideal functionality for coin-tossing

Coin-Tossing In fig. 2.1 we introduce the standard coin-tossing functionality, which allows maliciously secure sampling of uniformly random integers modulo some m.

$\mathcal{F}_{\mathsf{ABB}}(n, m)$

This functionality is parametrized by the number of parties participating, n, and the modulo which the computation is over, $\mathbb{Z}_m$.

Rand: Upon receiving $(\texttt{random}, \texttt{ssid})$ from all parties for a fresh ssid, the box samples $x \leftarrow_R \mathbb{Z}_m$ and stores $(\texttt{ssid}, x)$.

Input: Upon receiving $(\texttt{input}, i, \texttt{ssid}, x)$ for a fresh ssid from some party $i \in [n]$ and $(\texttt{input}, i, \texttt{ssid}, ?)$ from all other parties $j \in [n] \backslash \{i\}$, then store $(\texttt{ssid}, x \mod m)$.

Linear: Upon receiving $(\texttt{linear}, \alpha, \beta, \texttt{ssid}_1, \texttt{ssid}_2)$ for $\alpha, \beta \in \mathbb{Z}_m$ and a fresh $\texttt{ssid}_2$ and existing $\texttt{ssid}_1$ from all parties $i \in [n]$, retrieve $(\texttt{ssid}_1, x)$ and store $(\texttt{ssid}_2, \alpha \cdot x + \beta)$.

Add: Upon receiving $(\texttt{add}, \texttt{ssid}_1, \texttt{ssid}_2, \texttt{ssid}_3)$ for a fresh $\texttt{ssid}_3$ and existing $\texttt{ssid}_1$ and $\texttt{ssid}_2$ from all parties $i \in [n]$, retrieve $(\texttt{ssid}_1, x)$, $(\texttt{ssid}_2, y)$ and store $(\texttt{ssid}_3, x + y)$.

Mult: Upon receiving $(\texttt{mult}, \texttt{ssid}_1, \texttt{ssid}_2, \texttt{ssid}_3)$ for a fresh $\texttt{ssid}_3$ and existing $\texttt{ssid}_1$ and $\texttt{ssid}_2$ from all parties $i \in [n]$, retrieve $(\texttt{ssid}_1, x)$, $(\texttt{ssid}_2, y)$ and store $(\texttt{ssid}_3, x \cdot y)$.

Output: Upon receiving $(\texttt{output}, \texttt{ssid}, \mathcal{P})$ for an existing ssid with $\mathcal{P} \subseteq [n]$ from all parties $i \in [n]$, retrieve $(\texttt{ssid}, x)$ and send x to the adversary Adv. Wait for Adv to return either deliver or abort. If the adversary returns deliver, output x to all parties in $\mathcal{P}$, otherwise output abort.

Abort: Adv may at any point input abort at which point the functionality returns abort to all parties and aborts by not accepting any more calls for the current sid.

Fig. 2.2. Ideal functionality for a maliciously secure arithmetic black box with abort.

MPC Functionality We are building our protocol on top of any already existing *maliciously* secure MPC scheme where an adversary may *statically* corrupt a *dishonest majority* of the participating parties, and where computations can be modeled as an arithmetic black box over a field or ring with a domain of size at least 2^s. We observe the model fits well with many modern schemes such as SPDZ [DPSZ12] and SPD$\mathbb{Z}_{2^k}$ [CDE+18] for any number of parties or OLE schemes [DGN+17, HIMV19] in the two-party setting. We describe the ideal functionality in fig. 2.2.

We refer the reader to Appendix A for more details on the MPC models.

When it comes to all MPC-related functionalities we implicitly assume each command contains a session ID, sid, for the specific execution instance of the functionality. We will, however, not write this explicitly. Furthermore, we will assume that all received commands get relayed verbatim to all parties, including the adversary. The only exception being input commands; $(\texttt{input}, i, \texttt{ssid}, x)$, where the private input x is removed and instead the message $(\texttt{input}, i, \texttt{ssid}, \perp)$ is relayed.

As a convenience, we denote a value x within the ABB box working on integers modulo m as $[x]_m$ and assume that $(\texttt{add}, \dots)$, $(\texttt{linear}, \dots)$ and $(\texttt{mult}, \dots)$ reflect the natural commands on $[\cdot]_m$. I.e. $[w]_m = (\alpha \cdot [x]_m + [y]_m) \cdot [z]_m$ implicitly defines a call to $(\texttt{linear}, \dots)$, $(\texttt{add}, \dots)$ and $(\texttt{mult}, \dots)$ in the natural way s.t. $w = (\alpha \cdot x + y) \cdot z \mod m$.

We furthermore assume that $[x]_m$ in practice consists of an additive sharing between the parties. That is, we assume party i for $i \in [n]$ hold x_i s.t. $x = \sum_{i \in [n]} x_i \mod m$. We note, that this does *not* require white-box usage of the underlying MPC scheme, as the command $\mathcal{F}_{\mathsf{ABB}}(n, m).(\texttt{random}, \dots)$ can be used to define $[x_i]_m$ for $i \in [n-1]$. Then $\mathcal{F}_{\mathsf{ABB}}(n, m).(\texttt{add}, \dots)$ can be used to define x_n and $\mathcal{F}_{\mathsf{ABB}}(n, m).(\texttt{random}, \dots)$ to sample a random additive sharing of $[x_n]_m = \sum_{i \in [n-1]} [x_i]_m$.

Similarly, we will use $\textsc{Share}_m(x) \rightarrow [x]_m$ from party i to denote the call $\mathcal{F}_{\mathsf{ABB}}(n, m).(\texttt{input}, i, \texttt{ssid}_x, x)$ by party i and the call $\mathcal{F}_{\mathsf{ABB}}(n, m).(\texttt{input}, j, \texttt{ssid}_x, ?)$ by all other parties $i \neq j \in [n]$ for a fresh subsession ID $\texttt{ssid}_x$ associated with x. That is, we use ? to denote a value defined by another party. Similarly, we will use $\textsc{Open}_m([x]_m) \rightarrow x$ from each party $i \in [n]$ to denote the call $\mathcal{F}_{\mathsf{ABB}}(n, m).(\texttt{output}, \texttt{ssid}_x, [n])$ where $\texttt{ssid}_x$ denotes the subsession ID of $[x]_m$. That is, to open a value towards all parties. We use a similar shorthand for opening values towards a subset of parties or a specific part i.e., we let $\textsc{Open}_m([x]_m, \mathcal{P}) \rightarrow x$ from each party $i \in [n]$ denote the call $\mathcal{F}_{\mathsf{ABB}}(n, m).(\texttt{output}, \texttt{ssid}_x, \mathcal{P})$ where $\texttt{ssid}_x$ denotes the subsession ID of $[x]_m$, and thus where party $j \in \mathcal{P}$ learns x and all other parties, learn nothing besides the fact that parties in $\mathcal{P}$ have learned the value associated with $\texttt{ssid}_x$. We will also assume this convenience when sampling random values, by letting $[r]_m \leftarrow_R \mathbb{Z}_m$ denote $\mathcal{F}_{\mathsf{ABB}}(n, m).(\texttt{random}, \texttt{ssid}_r)$ for a subsession ID $\texttt{ssid}_r$ associated with r.

Finally, we highlight that the command $\mathcal{F}_{\mathsf{ABB}}(n, m).(\texttt{random}, \dots)$ can be used to trivially realize $\mathcal{F}_{\mathsf{CT}}$.

3 Truncation

In this section, we present our approach for efficient truncation in MPC through the use of an RNS. Below we present theorem 1 which gives an efficient algorithm for truncation with the smaller moduli in an RNS with two moduli. Then we show how to work with an RNS in MPC in Section 3.1.

Consider an RNS of two components as follows: Let $p, q \in \mathbb{N}$ be distinct positive integers with $\gcd(p, q) = 1$ and let $m = p \cdot q$. The Chinese Remainder Theorem yields the existence of a ring isomorphism $\mathbb{Z}_m \cong \mathbb{Z}_p \times \mathbb{Z}_q$ which we will denote by $\phi : \mathbb{Z}_m \to \mathbb{Z}_p \times \mathbb{Z}_q$ defined by $\phi(x) = (x \bmod p, x \bmod q)$. To ease the notation later, we let $x^{(p)}$ denote $x \in \mathbb{Z}_p$ and $x^{(q)}$ denote $x \in \mathbb{Z}_q$. The inverse of $\phi(x)$ is

$$\phi^{-1}(x^{(p)}, x^{(q)}) = (bqx^{(p)} + apx^{(q)}) \bmod m$$

where $a, b \in \mathbb{N}$ are chosen such that

$$ap + bq = 1.$$

Note that this implies that $ap = 1 \mod q$ and $bq = 1 \mod p$, so $a = p^{-1} \mod q$ and $b = q^{-1} \mod p$. Representing integers in $\mathbb{Z}_m$ as their images under ϕ is an example of an RNS with ϕ being a ring isomorphism that implies addition and multiplication may simply be done coordinate-wise. An RNS representation may also be used to efficiently compute truncation by one of the components as shown in the following theorem:

Theorem 1. *Let $\gcd(p, q) = 1$ and $x \in \mathbb{Z}$ with $0 \le x < pq = m$ and let $(x^{(p)}, x^{(q)}) = \phi(x)$. Then*

$$\lfloor x/p \rfloor = a(x^{(q)} - x^{(p)}) \bmod q$$

where $a = p^{-1} \bmod q$.

Proof. Let $a = p^{-1} \bmod q$ and $b = q^{-1} \bmod p$. Write $x = kp + r$ with $k \in \mathbb{Z}$ and $0 \le r < p$. Then $r = x^{(p)}$ and $\lfloor x/p \rfloor = k$ and since $x < m$ we have $0 \le k < q$, so we just need to prove that $k \equiv a(x^{(q)} - x^{(p)}) \pmod q$. Now

$$kp + x^{(p)} = x \equiv bqx^{(p)} + apx^{(q)} \pmod m$$

since $(x^{(p)}, x^{(q)}) = \phi(x)$. This implies

$$kp \equiv (bq - 1)x^{(p)} + apx^{(q)} \pmod m.$$

Since $q \mid m$, we may consider this congruence modulo q, and noting that $ap \equiv 1 \pmod q$ we get

$$k \equiv a((bq - 1)x^{(p)} + x^{(q)}) \equiv a(x^{(q)} - x^{(p)}) \pmod q$$

as desired. □

3.1 RNS in MPC

Consider an MPC scheme working over $\mathbb{Z}_m$. A value in such a scheme is defined using two MPC instances, one over $\mathbb{Z}_p$ and one over $\mathbb{Z}_q$ such that $\gcd(p, q) = 1$. We do so using ϕ as defined above to represent a value in $\mathbb{Z}_m$ for $m = pq$ by a pair of values in $\mathbb{Z}_p \times \mathbb{Z}_q$.

Concretely, we use two MPC instances, $\mathcal{F}_{\mathsf{ABB}}(n, p)$ and $\mathcal{F}_{\mathsf{ABB}}(n, q)$, which together with the linear function ϕ induces another MPC instance $\mathcal{F}_{\mathsf{ABB}}(n, m)$. We will abuse notation to use $[x]_m$ to denote an RNS realized through shares $[x^{(p)}]_p$ in $\mathcal{F}_{\mathsf{ABB}}(n, p)$ and $[x^{(q)}]_q$ in $\mathcal{F}_{\mathsf{ABB}}(n, q)$ where $m = p \cdot q$ and hence

$$x = (q \cdot (q^{-1}x^{(p)} \mod p) + p \cdot (p^{-1} \cdot x^{(q)} \mod q)) \mod pq\ .$$

That is, where $\phi(x) \to (x \mod p, x \mod q) = (x^{(p)}, x^{(q)})$. Concretely this means that $[x]_m = ([x \bmod p]_p, [x \bmod q]_q) = ([x^{(p)}]_p, [x^{(q)}]_q)$.

These values can be defined from the MPC commands:

$$\textsc{Share}_m(x) = (\textsc{Share}_p(x \bmod p), \textsc{Share}_q(x \bmod q))$$

and

$$\textsc{Open}_m([x^{(p)}]_p, [x^{(q)}]_q, \mathcal{P}) = \phi^{-1}(\textsc{Open}_p([x^{(p)}]_p, \mathcal{P}), \textsc{Open}_q([x^{(q)}]_q, \mathcal{P})).$$

3.2 Fixed-point Arithmetic

Observe that fixed-point representation of a real number is given as follows: If we let $b \in \mathbb{N}$ with $b \geq 2$ be the *base*, we may represent a real number $x \in \mathbb{R}$ by its *fixed-point representation with base* b given by the integer $f_b(x) = \lfloor b \cdot x \rfloor$. The fixed-point representation allows us to approximate arithmetic (see [SBIT22] for an analysis of the error terms) on real numbers by integer arithmetic since $f_b(x) + f_b(y) \approx f_b(x+y)$ and $\lfloor \frac{1}{b} f_b(x) f_b(y) \rfloor \approx f_b(x \cdot y)$. The base b is usually a power of two since this allows the division by b after each multiplication to be done by bit shifts, but any integer $b \geq 2$ will work.

With this in mind, we can do fixed-point computation in MPC over a domain $\mathbb{Z}_m$ with base p when $m = p \cdot q$ for numbers p, q with $\gcd(p, q) = 1$, given a generic MPC construction that works over $\mathbb{Z}_p$ and $\mathbb{Z}_q$ through the RNS mapping in section 3.1. That is, given $[x]_m$ with an RNS decomposition $\phi([x]_m) = ([x^{(p)}]_p, [x^{(q)}]_q)$, let $[y]_q = (p^{-1} \mod q) \cdot [x^{(q)}]_q - [x^{(p)}]_p$. Then compute $\lfloor [x]_m / p \rfloor = ([y \mod p]_q, [y]_q)$ by applying theorem 1.

However, one problem remains; How do we move a value $[y]_q$ to $[y]_p$ and vice versa as these values live in two distinct MPC instances?

4 The Construction

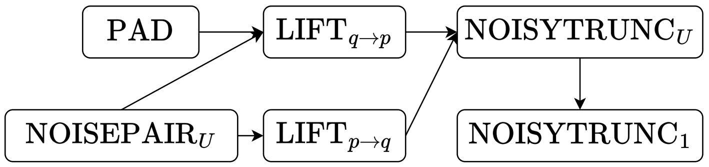

Fig. 1. Illustration of the dependencies between the different sub algorithms.

To facilitate the transfer of values between two different MPC instances we present two algorithms $\textsc{Lift}_{p \to q}$ (algorithm 3) and $\textsc{Lift}_{q \to p}$ (algorithm 4) which allow exactly this. Although the first of these has the side effect of a small additive error. Both of these algorithms, however, require secret correlated randomness in $\mathcal{F}_{\mathsf{ABB}}(n, p)$ and $\mathcal{F}_{\mathsf{ABB}}(n, q)$. Assume $q > p$, we preprocess a pair of values $([r]_p, [r + \epsilon \cdot p]_q)$ for a uniformly random $r \in \mathbb{Z}_p$ and some small non-negative integer $\epsilon < U$. We denote such a correlated randomness pair a *noise pair*.

We present a concrete algorithm for generating such noise pairs $\textsc{NoisePair}_U$ (1) for $U = n$, which is used as a black-box in $\mathcal{F}_{\mathsf{ABB}}(n, \cdot)$. Using a noise pair it is possible to compute and publicly open $[x^{(p)}]_p + [r]_p$ in $\mathcal{F}_{\mathsf{ABB}}(n, p)$, and input this into $\mathcal{F}_{\mathsf{ABB}}(n, q)$ since r will hide the secret value $x^{(p)}$. However, to go from $\mathcal{F}_{\mathsf{ABB}}(n, p)$ and $\mathcal{F}_{\mathsf{ABB}}(n, q)$ is not as easy, since $r \in \mathbb{Z}_p$ it cannot statistically hide a value $[x^{(q)}]_q$ when $q > p$. Thus to realize $\textsc{Lift}_{q \to p}$ we require another random value, denoted by ρ, which is larger than 2^s, and hence can statistically hide the part of the value $x^{(q)}$ which is larger than p. This is a standard technique known as *noise-drowning*. We specify the concrete randomness sampling procedure in algorithm $\textsc{Pad}$ (2). Using a pad and another noise pair, $([r']_p, [r' + \epsilon \cdot p]_q)$, it is possible to compute and open $[x^{(q)}]_q + [r' + \epsilon \cdot p]_q + p \cdot [\rho]_q$

and input this into $\mathcal{F}_{\mathsf{ABB}}(n,p)$ followed by the subtraction of $[r']_q$ in order to transfer a value from $\mathcal{F}_{\mathsf{ABB}}(n,q)$ to $\mathcal{F}_{\mathsf{ABB}}(n,p)$.

The approach might cause over and underflows. However, we show that to a large extent, by carefully picking parameters, this can be avoided. Specifically, we show how to combine $\textsc{Lift}_{p\to q}$ and $\textsc{Lift}_{q\to p}$, based on the idea of theorem 1, in $\textsc{NoisyTrunc}_U$, which computes the $\lfloor [x]_m/p \rfloor$ with an additive error of at most U. Furthermore, we show how to reduce this error to at most 1, with algorithm $\textsc{NoisyTrunc}_1$, which we present in Section 4.4.

In the sequel we formalize these algorithms, and formally prove them in section 5.

4.1 Preprocessing

Noise pairs A *noise pair* consists of correlated randomness $([r]_p, [\bar{r}]_q)$, where $r \leftarrow_R \mathbb{Z}_p$ is uniformly random and $\bar{r} = r + \epsilon p$ for an integer ϵ in the range $0 \leq \epsilon < U$ with a constant $U > 0$. Constructing correlated randomness over multiple MPC schemes is not a problem unique to us [RST+22]. Still, a small additive error typically only has minimal impact on fixed-point computations, and our truncation algorithm might introduce a small error *even* if $\bar{r} = r$. Hence instead of relying on previous results, we have tried to design a preprocessing protocol to facilitate transfer between two distinct MPC domains as lightweight as possible, *only* assuming black-box access to an *arithmetic* MPC scheme $\mathcal{F}_{\mathsf{ABB}}(n,\cdot)$, *but* with acceptance of a small additive error.

As a warm-up for our noise pair algorithm, first consider the semi-honest setting. In this setting, it is sufficient for each party to input random values $r^{(i)} \leftarrow_R \mathbb{Z}_p$ into the MPC computation both over $\mathbb{Z}_p$ and $\mathbb{Z}_q$. MPC can then be used to compute the values $[r]_p = \sum_{i\in[n]}[r^{(i)}]_p$ and $[\hat{r}]_q = [r + \epsilon \cdot p]_q = \sum_{i\in[n]}[r^{(i)}]_q$ where $\epsilon < n$ depended on the values of $r^{(i)}$. This approach is clearly correct, but only semi-honestly secure as a malicious party could simply input inconsistent values in $\mathbb{Z}_p$ and $\mathbb{Z}_q$.

Since the random value r is independent of any secret input, it is straightforward to make this covertly secure, through the standard covert paradigm of committing to multiple candidates, validating all but one, and then keeping the last one [AL10]. That is, each party i selects λ random values $r_1^{(i)}, \ldots, r_\lambda^{(i)} \leftarrow_R \mathbb{Z}_p$ for $\lambda = \lceil 1/(1-\gamma) \rceil$ with γ being the deterrence factor. Then input this into $\mathcal{F}_{\mathsf{ABB}}(n,p)$ and $\mathcal{F}_{\mathsf{ABB}}(n,q)$ through $\textsc{Share}_p(r_j^{(i)})$ and $\textsc{Share}_q(r_j^{(i)})$ for $i \in [n]$ and $j \in [\lambda]$. Using a coin-tossing protocol, the parties collaboratively select an index $c \leftarrow_R \mathbb{Z}_\lambda$ to keep and check $\textsc{Open}_p([r_j^{(i)}]_p = \textsc{Open}_q([r_j^{(i)}]_q)$ for $i \in [n]$ and $j \in [\lambda]\backslash\{c\}$. Finally the pair to keep is computed $[r]_p = \sum_{i\in[n]}[r_c^{(i)}]_p \mod p$ and $[\hat{r}]_q = [r + \epsilon \cdot p]_q = \sum_{i\in[n]}[r_c^{(i)}]_q$. However, one subtlety that occurs with this approach is that an adversary corrupting $n-1$ parties can now decide on a value $\tilde{\epsilon} < n-1$ and cause an error of $\epsilon = \tilde{\epsilon}$ or $\epsilon = \tilde{\epsilon} + 1$ (depending on the random choice of the honest party). This is because the adversary can choose not to pick the values $r_j^{(i)}$ randomly and hence control the amount of overflows modulo p that occurs when adding together the values modulo q.

Fortunately, this is easy to handle by having each party contribute part of each of the other party's random share modulo p. This ensures that as long as there is a single honest party, then all shares will be randomly distributed and thus the overflow cannot be controllable by the adversary, and is hence guaranteed to be Bernoulli distributed. We formally describe this in algorithm 1. However, we do note that this approach still does not prevent the adversary from knowing ϵ with 1 bit uncertainty.

Furthermore, observe that since we don't assume an underlying MPC scheme with identifiable abort, the adversary could cheat in one of the covertly generated pairs, and abort in case that pair gets selected for verification. To handle this problem we can use the folklore approach of *commit-and-open* wherein parties commit to the randomness they need to execute algorithm 1 and broadcast these commitments. If an abort happens in the underlying MPC scheme, then all parties will have to open their commitments. All parties can then recompute what all other parties were supposed to send and hence identify any party that has not followed the protocol. This of course reveals no secret information since algorithm 1 only does prepossessing, and hence is independent of any private input. For simplicity, we will however leave it out of the formal description.

The algorithm works in the $\mathcal{F}_{\mathsf{ABB}}$-, $\mathcal{F}_{\mathsf{CT}}$-hybrid model and produces a result that is contained *within* the ideal functionality of $\mathcal{F}_{\mathsf{ABB}}$. Hence we cannot prove the algorithm UC-secure on its own without resulting to

Algorithm 1 Covert $\textsc{NoisePair}_n() = ([r]_p, [\bar{r}]_q)$ with deterrence factor γ

```
Require: $\perp$
Ensure: $([r]_p, [\bar{r}]_q = [r + \epsilon \cdot p]_q)$ for $0 \leq \epsilon < n$ with $\epsilon \sim \mathbf{IrwinHall}(n)$
 1: Let $\lambda = \lceil 1/(1-\gamma) \rceil$
 2: for $k \in [\lambda]$ do
 3:     Each party $j$ samples $(r_{1,j,k}, \cdots, r_{n,j,k}) \leftarrow \mathbb{Z}_p^n$
 4:     $\textsc{Share}_p(r_{i,j,k}) \rightarrow [r_{i,j,k}]_p$ for each $i, j \in [n]$.
 5:     $[r_{i,k}]_p = \sum_{j \in [n]} [r_{i,j,k}]_p$ for $i \in [n]$.
 6:     $\textsc{Open}_p([r_{i,k}]_p, \{i\})$, so party $i$ learns $r_{i,k}$, for $i \in [n]$.
 7:     $\textsc{Share}_q(r_{i,k}) \rightarrow [r_{i,k}]_q$, for $i \in [n]$
 8: end for
 9: $\mathcal{F}_{\mathsf{CT}}.\mathtt{sample}(\mathtt{ssid}, \lambda) \rightarrow c$                ▷ From fig. 2.1
10: for For each party $i \in [n]$ and noise pair $k \in [\lambda] \setminus \{c\}$ do
11:     $r_{i,k}^{(p)} \leftarrow \textsc{Open}([r_{i,k}]_p)$ and $r_{i,k}^{(q)} \leftarrow \textsc{Open}([r_{i,k}]_q)$
12: end for
13: if $\exists r_{i,k}^{(p)} \neq r_{i,k}^{(q)}$ for any $i \in [n]$ and $k \in [\lambda] \setminus \{c\}$ then
14:     Output $(\mathtt{cheat}, i)$ and terminate the algorithm.
15: end if
16: $[r_c]_p = \sum_{i \in [n]} [r_{i,c}]_p \mod p$ for $i \in [n]$
17: $[r_c + \epsilon \cdot p]_q = [\bar{r}_c]_q = \sum_{i \in [n]} [r_{i,c}]_q \mod q$, for $i \in [n]$
18: return $([r_c]_p, [\bar{r}_c]_q)$
```

stateful UC-models such as GUC [CDPW07]. However, keeping a global MPC state does not seem the right choice, given that our secure computations are stand-alone. For this reason, we will instead prove our full protocol secure in a monolithic manner in section 5. The intuition in simulating the covert noise pairs is, however, very simple and revolves around choosing the honest parties' shares of r_c such that the right ϵ is achieved and choosing a proper c such that the right iterations are opened. This will be argued in the proof of lemma 7 as part of the monolithic proof.

Observe that we can somewhat accurately estimate the error ϵ as follows:

Remark 1. The variable ϵ in the output of algorithm 1 is approximately Irwin-Hall[4] distributed when executed with at least one honest party,

$$\epsilon \sim \mathbf{IrwinHall}(n).$$

The proof of this remark can be found in Appendix B.

Padding Our protocol also requires bounded randomness, used to hide overflow modulo p when padding a value from $\mathbb{Z}_p$ in $\mathbb{Z}_q$, but also when we wish to statistically hide a bounded value in $\mathbb{Z}_q$. In our protocol we handle this using noise-drowning, i.e. by hiding any overflow, which can leak secret information, using exponentially large noise. While simply using a random value in $\mathbb{Z}_q$ would be sufficient, it would risk affecting correctness since any overflow modulo q would yield an incorrect result. Hence we need to sample randomness that prevents an overflow modulo q, but which is exponentially large in the security parameter to ensure that it hides any modulo p overflow. In the semi-honest setting, this is easy to achieve by having each party sample uniform randomness of sufficient size and summing the contribution of each party. That is if the bound is A then each party i samples $\rho_i \leftarrow_R \mathbb{Z}_A$ and the parties compute $\rho = \sum_{i \in [n]} \rho_i$. Clearly $\rho \mod A$ is uniformly random if just a single party has been honest. Furthermore, if all parties follow the protocol then it will hold that $\rho < An$.

Because the input of each party (ρ_i) is independent of the underlying function we wish to evaluate in MPC, it becomes clear that we can perform this sampling with covert security following the same paradigm we used

[4] Recall that the Irwin-Hall distribution is the distribution of a sum of n independent random variables each of which are uniformly distributed on $[0, 1)$ and that the $\mathbf{Irwin\text{-}Hall}(n) \rightarrow \mathbf{N}(n/2, n/12)$ as $n \rightarrow \infty$.

for noise pairs above. That is, we sample λ ρ candidates and validate a random set of $\lambda - 1$ candidates. If they are all less than An we assume this will also hold for the last one and we will use this one. As in the case for the noise pairs we use a standard commit-and-open approach to ensure that the adversary cannot abort to avoid detection.

We formalize this in algorithm 2 in the $\mathcal{F}_{\mathsf{CT}}$, $\mathcal{F}_{\mathsf{ABB}}$-hybrid model.

Algorithm 2 Covert $\textsc{Pad}(A) \rightarrow [\rho]_q$ with deterrence factor γ.

```
Require: $An < q$
Ensure: $[\rho]_q$ with $\rho \mod A \leftarrow_R \mathbb{Z}_A$ and $\rho < An$
1: Let $\lambda = \lceil 1/(1-\gamma) \rceil$
2: Each party $i$ samples $\rho_{i,k} \leftarrow \mathbb{Z}_A$ for $k \in [\lambda]$.
3: Each party $i$ does $\textsc{Share}_q(\rho_{i,k}) \rightarrow [\rho_{i,k}]_q$.
4: $\mathcal{F}_{\mathsf{CT}}.\texttt{sample}(\texttt{ssid}, \lambda) \rightarrow c$        ▷ From fig. 2.1
5: $\rho_{i,k} \leftarrow \textsc{Open}_q([\rho_{i,k}]_q)$ for $i \in [n]$ and $k \in [\lambda] \backslash \{c\}$.
6: if $\exists i, k \ : \rho_{i,k} \geq A$ then
7:     Output $(\texttt{cheat}, i)$ and abort
8: end if
9: return $[\rho]_q = \sum_{i \in [n]} [\rho_{i,c}]_q$.
```

Like for the noise pair construction above, we will use the algorithm as part of the full protocol, $\mathit{\Pi}_{\mathsf{aABB}}$, which we present and prove secure in section 5. Again it is crucial for the simulation, to choose a proper c, to open the right iterations. Which will again be discussed in lemma 7 as part of the monolithic proof.

4.2 Lifting

Based on noise pairs and pads we now introduce the lifting algorithms $\textsc{Lift}_{p \rightarrow q}$ and $\textsc{Lift}_{q \rightarrow p}$ in algorithm 3 and algorithm 4 respectively, which we use to move values from $\mathcal{F}_{\mathsf{ABB}}(n, p)$ to $\mathcal{F}_{\mathsf{ABB}}(n, q)$ and vice versa.

Algorithm 3 Compute $\textsc{Lift}_{p \rightarrow q}([x]_p) = [y]_q$

```
Require: $[x]_p$
Ensure: $[y]_q = [x - \epsilon p]_q$ with $0 \leq \epsilon \leq U$        ▷ Using algorithm 1
1: Sample a pair $([r]_p, [\bar{r}]_q) \leftarrow \textsc{NoisePair}_U()$
2: $\bar{x} \leftarrow \textsc{Open}([x]_p + [r]_p)$
3: $[y]_q \leftarrow \bar{x} - [\bar{r}]_q$
4: return $[y]_q$
```

Lemma 1. *Algorithm 3 computes $[y]_q = [x - \epsilon \cdot p]_q$ where ϵ is an integer with $0 \leq \epsilon \leq U$ when $\bar{r} = r + \tilde{\epsilon} \cdot p$ for $0 \leq \tilde{\epsilon} < U$.*

Proof. Note that $\bar{x} = x + r - b \cdot p$ where $b \in \{0, 1\}$ and $b = 1$ if and only if $x + r \geq p$. Now, since $\bar{r} = r + \tilde{\epsilon} \cdot p$ for $0 \leq \tilde{\epsilon} < U$ we have

$$\bar{x} - \bar{r} = x - b \cdot p - \tilde{\epsilon} \cdot p.$$

Setting $\epsilon = b + \tilde{\epsilon}$ finishes the proof as this implies $\epsilon \leq U$. □

Algorithm $\textsc{Lift}_{q \rightarrow p}$ requires both a noise pair, $(r, \bar{r})$ and some auxiliary randomness, ρ. The bounded, yet exponentially large (in the security parameter) auxiliary randomness is used to statistically hide the value ϵ when viewing $\bar{x} = x' + \epsilon \cdot p$ for $x' < p$. However, this is non-trivial since it is not possible to sample ρ for the full domain $\mathbb{Z}_q$, as this could result in an incorrect result, and since ρ must be unknown to all parties.

Algorithm 4 Compute $\textsc{Lift}_{q\to p}([x]_q) = [y]_p$

```
Require: $[x]_q$ with $x < Q \le \frac{q-(U+1)p}{n(U+1)(2^{\tilde{s}}+1)}$
Ensure: $[y]_p = [x \mod p]_p$
1: $([r]_p, [\bar{r}]_q) \leftarrow \textsc{NoisePair}_U()$
2: $[\rho]_q \leftarrow \textsc{Pad}((U+1)2^{\tilde{s}}Q/p)$        ▷ Using Algorithm 2
3: $\bar{x} \leftarrow \textsc{Open}_q([x]_q + [\bar{r}]_q + [\rho]_q \cdot p)$
4: $[y]_p \leftarrow (\bar{x} \bmod p) - [r]_p$
5: return $[y]_p$
```

Lemma 2. *Algorithm 4 computes* $[x \bmod p]_p$ *when* $\rho < n(U+1)2^{\tilde{s}}Q/p$ *for* $p \le Q \le \frac{q-(U+1)p}{n(U+1)(2^{\tilde{s}}+1)}$.

Proof. Since $x < Q \le \frac{q-(U+1)p}{n(U+1)(2^{\tilde{s}}+1)}$ we have

$$\begin{aligned} x + \bar{r} + \rho p &< x + Up + p + (n(U+1)2^{\tilde{s}}Q/p)p \\ &= Q + (U+1)p + n(U+1)2^{\tilde{s}}Q \\ &< (U+1)p + n((U+1)2^{\tilde{s}}+1)Q \\ &\le (U+1)p + n((U+1)2^{\tilde{s}}+1)\frac{q-(U+1)p}{n(U+1)(2^{\tilde{s}}+1)} \\ &= (U+1)p + q - (U+1)p = q \end{aligned}$$

Hence no overflow modulo q will happen. Thus we can define $\bar{x} = x + \bar{r} + \rho p$ as integers which implies that $\bar{x} \bmod p = x + r \bmod p$, so $y = x \bmod p$ as desired. ◻

For security, we require $\rho > 2^s$ to ensure statistically hiding noise-drowning.

4.3 Probabilistic Truncation

We now show how to use $\textsc{Lift}_{p\to q}$ and $\textsc{Lift}_{q\to p}$ to do efficient truncation with a small error in algorithm $\textsc{NoisyTrunc}$. The error stems from $\textsc{Lift}_{p\to q}$, both due to the error permitted in $\textsc{NoisePair}_U$ and due to the possibility that $x + r > p$, which will result in an error of a single bit when it happens.

Algorithm 5 Compute $\textsc{NoisyTrunc}_U([x]_m) = [y]_m$

```
Require: $[x]_m = ([x_1]_p, [x_2]_q)$ with $0 \le x < pQ - Up$
Ensure: $[y]_m = [\lfloor x/p \rfloor + \epsilon]_m$ with $0 \le \epsilon \le U$ for $[y]_m = ([y_1]_p, [y_2]_q)$
    $[\bar{x}_1]_q \leftarrow \textsc{Lift}_{p\to q}([x_1]_p)$
    $[y_2]_q \leftarrow (p^{-1} \bmod q)([x_2]_q - [\bar{x}_1]_q)$
    $[y_1]_p \leftarrow \textsc{Lift}_{q\to p}([y_2]_q)$
    return $([y_1]_p, [y_2]_q)$
```

Lemma 3. *Algorithm 5 computes* $[\lfloor x/p \rfloor + \epsilon]_m$ *with* $0 \le \epsilon \le U$.

Proof. From algorithm 3 we get that $\bar{x}_1 = x_1 - \epsilon p$ for some integer ϵ with $0 \le \epsilon \le U$. Next, from theorem 1 we get that $(p^{-1} \bmod q)(x_2 - x_1) = \lfloor x/p \rfloor$, so

$$[y_2]_q = (p^{-1} \bmod q)([x_2]_q - [\bar{x}_1]_q) = (p^{-1} \bmod q)([x_2]_q - [x_1 - \epsilon p]_q) = [\lfloor x/p \rfloor + \epsilon]_q.$$

Now,

$$y_2 = \lfloor x/p \rfloor + \epsilon < \lfloor \frac{pQ - Up}{p} \rfloor + \epsilon = Q - U + \epsilon \leq Q,$$

hence the input size requirement of algorithm 4 is fulfilled and will thus yield the correct result. This concludes the proof. □

Error. Observe that the Algorithm algorithm 3 may inherently cause a 1-bit additive error, *even* if there is no error in the NoisePair used. Specifically, this occurs if $x + r > p$ as a modulo wrap-around will occur. This error is carried over to algorithm 5, hence always resulting in the potential of a 1-bit additive error in the truncation result. This is unfortunately inevitable with our algorithms. However, we argue that when our scheme is used for fixed-point computation, this will rarely cause any issues. The reason is that any error will only be present in the least significant digits of the result of a multiplication. Fixed-point computation is already an approximation of true values, hence any usage of such algorithms must already take into account the potential of a rounding error. Thus, on an intuitive level, we expect any algorithm using fixed-point to not be highly sensitive to a slight error in the least significant digits, as a half-bit error can *always* be expected implicitly as part of inevitable rounding. The sensitivity can of course be reduced by increasing the precision. Even so, a small error *may* still accumulate through repeated multiplications. This would for example be the case if computing exponentiation through repeated multiplications. Hence, one should take into account how the multiplicative depth of a given computation can cause an increase in error, and increase the fixed-point precision (i.e. the choice of p) accordingly.

This has also been confirmed (for a 1-bit additive error) by Mohassel and Zhang [MZ17] in the setting of machine learning regression training on standard datasets such as MNIST. In Section 6.1 we confirm that this is indeed also the case for the Fast Fourier Transform when using our protocol (even when allowing for an error up to n).

Reduction in computation space. Besides the potential of adding a small error, algorithm 5 also requires a reduction of the available computation space. This is because noise-drowning is required to prevent any leakage when moving a secret shared value from $\mathcal{F}_{\mathsf{ABB}}(n, q)$ to $\mathcal{F}_{\mathsf{ABB}}(n, p)$. While an RNS over $\mathbb{Z}_q$ and $\mathbb{Z}_p$ should give a ring $\mathbb{Z}_m$ with $m = p \cdot q$, algorithm 5 requires the value $x \in \mathbb{Z}_m$ to be less than $pQ - Up$ for $Q \leq \frac{q-(U+1)p}{n(U+1)(2^{\tilde{s}}+1)}$. Assuming we use algorithm 1 for preprocessing and hence that $U = n$, then the amount of usable bits in $\mathbb{Z}_q$ is approximately $\log(q) - 2\log(n) - \tilde{s}$. Hence the largest value we can represent will have to consist of less than approximately $\log(p) + \log(q) - 2\log(n) - \tilde{s}$ bits. That is, we lose approximately $2\log(n) + \tilde{s}$ bits of $\mathbb{Z}_m$. Since we require $Q > p$ (otherwise we could not fully represent values from $\mathbb{Z}_p$ in $\mathbb{Z}_q$ when lifting in algorithm 3), we can conclude that $q > n^2 p 2^{\tilde{s}}$. Such a domain size is significantly larger than 2^s which is typically the *minimally* required size by standard MPC schemes such as SPDZ [DPSZ12]. However, requiring a gap in computation space when computing truncation is common. Several previous works [CS10, EGK+20] require at least the s most significant bits to be 0 to be able to do truncation correctly. It is also worth stressing that both in our and previous works, the limit in computation space is *only* relevant for the value being truncated. Hence general computation can use the full domain in both cases.

Input constraints. Reduction in computation space is not the only constraint we encounter on the magnitude of *secret* values. Specifically corrupt parties are always allowed to choose their own input in MPC, and since the underlying scheme $\mathcal{F}_{\mathsf{ABB}}(n, q)$ supports the full domain $\mathbb{Z}_q$, we cannot simply hope that their input fulfills the constraints required by Lemma 4.2. However, this can be enforced by using a comparison operation [Cd10]. Still, depending on the computation, such a check might be superfluous in the security model. since the correctness of most computations will not be fulfilled if corrupt parties give malformed input. This is inherently something that cannot be prevented in MPC unless the input can be anchored in some manner. Hence, causing a computation to fail by giving bad input that yields a bad result, or giving bad input that yields an error in one of the underlying algorithms, might amount to the same thing.

Negative numbers. One final problem that occurs when constraining the computation domain is that representing negative numbers using two's complement is no longer possible. This is because a negative number with a small absolute value, will not fulfill the input constraint of algorithm 4. However, it fortunately turns out to be easy to still facilitate computation over signed values when q is odd and $p \mid (q-1)/2$ by applying the following approach: Given unsigned input $x \in \mathbb{Z}_m$, let $x > m/2$ represent the negative integer $x - m$, similarly to two's complement.

However, before any truncation is computed we increase the unsigned input $x \in \mathbb{Z}_m$ by $p(q-1)/2 \approx m/2$. Note that this does not affect $x^{(p)}$ of $\phi(x) \to (x^{(p)}, x^{(q)})$ and since $p \mid (q-1)/2$, we have $\phi(x + p(q-1)/2) = (x^{(p)}, x^{(q)} + p(q-1)/2)$. Formally, we define a new operator $\textsc{NoisyTrunc}'_U$ by

$$\textsc{NoisyTrunc}'_U(x) = \textsc{NoisyTrunc}_U(x + ap) - a$$

where $a = \lfloor q/2 \rfloor \approx m/2$. Recall that $\textsc{NoisyTrunc}_U(x') = \lfloor x'/p \rfloor + \varepsilon$ for $0 \le \varepsilon \le U$ if x' satisfied the upper bound in Algorithm 5, hence the following holds:

$$\begin{aligned}
\textsc{NoisyTrunc}'_U(x) &= \textsc{NoisyTrunc}_U(x + ap) - a \\
&= \textsc{NoisyTrunc}_U(x + \lfloor q/2 \rfloor p) - \lfloor q/2 \rfloor \\
&= \lfloor x/p \rfloor + \varepsilon + (q-1)/2 - (q-1)/2 = \lfloor x/p \rfloor + \varepsilon.
\end{aligned}$$

4.4 Error Reduction

Below we show an approach for reducing the error that can occur in the approximate truncation algorithm 5 above. The error can be removed completely by using Lemma 8 in appendix B with $M > p \cdot U$, this, however, requires evaluating a very large polynomial of degree $\approx pU^2$ in MPC, after computing algorithm 5. Which would completely remove any advantage of our algorithm. Instead, we propose an algorithm that reduces the error down to a single additive bit, and involves evaluating algorithm 5 twice, along with evaluating a polynomial of degree $\approx U^2$ in MPC; something that requires the online computation of $O(U^2)$ secure multiplications. We describe this in algorithm 6 and prove it in lemma 8 in appendix B.

Intuitively by first doing the truncation with an error up to U, it is possible to multiply the result with p to isolate the noise and then apply the truncation again. Hence, the result can be adjusted to have noise that is at most 1.

While a lower error is objectively desirable we believe this algorithm is more of theoretical interest than practical interest, as it requires running algorithm 5 twice *along* with $O(n^2)$ multiplications. This is a very significant overhead, while the payoff is minimal when running with a small number of parties such as 2 or 3.

Algorithm 6 Compute $\textsc{NoisyTrunc}_1([x]_m) = [\lfloor x/p \rfloor + \varepsilon']_m$ with $0 \le \varepsilon' \le 1$. Let P be a polynomial of degree $U^2 + 3U - 1$ as defined in Remark 6 such that $P(x) = \lfloor x/(U+1) \rfloor$.

```
Require: $0 \le x < m - (U+1)(2^{\bar{s}}+1)p^2 - Up$, $[x]_m = ([x_1]_p, [x_2]_q)$
    $[y]_m \leftarrow \textsc{NoisyTrunc}_U([x]_m)$                      ▷ $y = \lfloor \frac{x}{p} \rfloor + \epsilon$
    $[x']_m = (U+1)([x]_m - p[y]_m + pU)$                   ▷ $x' = (U+1)(p(U-\epsilon) + (x \bmod p))$
    $[w]_m \leftarrow \textsc{NoisyTrunc}_U([x']_m)$            ▷ $w = \lfloor \frac{(U+1)(x \bmod p)}{p} \rfloor + (U+1)(U-\epsilon) + \varepsilon'$
    $[y']_m = [y]_m - U - P([w]_m)$              ▷ $y' = \lfloor x/p \rfloor + \varepsilon'$. See the proof of lemma 8 for details.
    return $[y']_m$
```

Remark 2. The variable ε' in the output of algorithm 6 is distributed as follows when executing with at least one honest party:

$$\varepsilon' \sim \mathbf{Bernoulli}((x \bmod p)/p) + \mathbf{N}\left(\frac{1}{2}, \frac{1}{12U}\right)$$

Still, one more detail to consider is that the input to the second application of truncation has to satisfy the upper bound constraint of algorithm 5. More concretely:

Remark 3. Assume $U^2 + 2U \leq 2^s$ and $Q > p > 2^s$ then when $x' = (U+1)([x]_m - p[y]_m + pU)$ as per algorithm 6 then $x' < pQ - Up$.

We leave the proofs of remark 2 and 3 to appendix B. We also leave the full specification of polynomial P (in algorithm 6) to lemma 8 and the proof of correctness of algorithm 6 to proposition 1 both found in appendix B.

Finally, we observe that evaluating a polynomial of degree $O(U^2)$ in MPC would traditionally require $O(\log(U^2)) = O(\log(U))$ multiplicative depth due to the dependency of the factors when computing exponentiation. However, a technique exists that can reduce this to constant rounds while still only requiring $O(U^2)$ multiplication gates. We refer the interested reader to the work of Bar-Ilan and Beaver for details [BIB89].

5 Formal UC Construction

Based on the discussion in Section 3 and 4, we now formally show how to construct an augmented MPC functionality, $\mathcal{F}_{\mathsf{aABB}}$, allowing truncation modulo p with an additive error of at most 1. In fig. 5.1 the ideal functionality is shown (i.e. implementation of algorithm 6). In fig. 5.2 and 5.3 we show how to realize this functionality in the $\mathcal{F}_{\mathsf{ABB}}$-hybrid model. We choose to show this version of the protocol (algorithm 6) and prove its simulatability because it is a theoretically interesting case and is simply an extension, building on top of algorithm 5, and hence the protocol and the associated security proof of algorithm 5 follows from the presented protocol and security proof when adjusting the error in the ideal functionality. As described in section 4.1 we use a standard commit-and-open approach to ensure that the adversary can not avoid detection by aborting.

Since it is possible to implement truncation using $\mathcal{F}_{\mathsf{ABB}}$, our construction might intuitively seem superfluous, however our goal is to show how to realize approximate truncation *more efficiently* by leveraging an RNS, rather than realizing it using an MPC scheme in itself. Furthermore, we note that our $\mathcal{F}_{\mathsf{aABB}}$ realization also reduces the problem of doing MPC over $\mathbb{Z}_m$ to the problem of doing MPC over *smaller* domains $\mathbb{Z}_p$ and $\mathbb{Z}_q$, which could potentially lead to optimizations when working over very large domains, e.g. when using MPC to construct RSA moduli [ACS02]. For this reason, we realize $\mathcal{F}_{\mathsf{aABB}}$ using two instances of $\mathcal{F}_{\mathsf{ABB}}$; one modulo p and one modulo q. We *note* that the security model of $\mathcal{F}_{\mathsf{aABB}}$ builds on the security of $\mathcal{F}_{\mathsf{ABB}}$ (in fig. 2.2), which is *malicious*. The only exception is that the optimized truncation command only affords *covert* robustness, in correspondence with our current realization of generation of correlated randomness using the NoisePair (algorithm 1). Furthermore, we notice that the *distribution* of the truncation error is necessarily closely linked to the underlying protocol for NoisePair.

$\mathcal{F}_{\mathsf{aABB}}(n, p, q, \gamma)$

This functionality is parametrized by the number of parties participating, n, and the modulo which the computation is over, $m = p \cdot q$, and a deterrence factor γ.

Rand: Upon receiving $(\texttt{random}, \texttt{ssid})$ from all parties for a fresh $\texttt{ssid}$, sample $x \leftarrow_R \mathbb{Z}_m$ and store $(\texttt{ssid}, x)$.

Input: Upon receiving $(\texttt{input}, i, \texttt{ssid}, x)$ for a fresh $\texttt{ssid}$ from some party $i \in [n]$ and $(\texttt{input}, i, \texttt{ssid}, ?)$ from all other parties $j \in [n] \backslash \{i\}$, then store $(\texttt{ssid}, x \mod m)$.

Linear: Upon receiving $(\texttt{linear}, \texttt{ssid}_1, \texttt{ssid}_2, \alpha, \beta)$ for $\alpha, \beta \in \mathbb{Z}_m$ and a fresh $\texttt{ssid}_2$ and existing $\texttt{ssid}_1$ from all parties $i \in [n]$, retrieve $(\texttt{ssid}_1, x)$ and store $(\texttt{ssid}_2, \alpha \cdot x + \beta)$.

Add: Upon receiving $(\texttt{add}, \texttt{ssid}_1, \texttt{ssid}_2, \texttt{ssid}_3)$ for a fresh $\texttt{ssid}_3$ and existing $\texttt{ssid}_1$ and $\texttt{ssid}_2$ from all parties $i \in [n]$, retrieve $(\texttt{ssid}_1, x)$, $(\texttt{ssid}_2, y)$ then store $(\texttt{ssid}_3, x + y)$.

Mult: Upon receiving $(\texttt{mult}, \texttt{ssid}_1, \texttt{ssid}_2, \texttt{ssid}_3)$ for a fresh $\texttt{ssid}_3$ and existing $\texttt{ssid}_1$ and $\texttt{ssid}_2$ from all parties $i \in [n]$, retrieve $(\texttt{ssid}_1, x)$, $(\texttt{ssid}_2, y)$ and store $(\texttt{ssid}_3, x \cdot y)$.

Truncation: Upon receiving $(\texttt{truncation}, \texttt{ssid}_1, \texttt{ssid}_2)$ for an existing $\texttt{ssid}_1$ and fresh $\texttt{ssid}_2$ from all parties $i \in [n]$, sample ϵ and compute $\varepsilon' \in \{0, 1\}$ as described in Remark 2, and send $(\texttt{truncation}, \epsilon, \varepsilon')$ to Adv. Once Adv returns $(\texttt{cheat}, j, g, f)$ where j is a corrupt party, $g \in \{0, 1\}$ and $f : \mathbb{Z}_m \to \mathbb{Z}_m$, proceed as follows:
- If $g = 0$ retrieve $(\texttt{ssid}_1, x)$ and compute $\bar{x} = \lfloor \frac{x}{p} \rfloor + \varepsilon'$ and store $(\texttt{ssid}_2, \bar{x})$.
- If $g = 1$ sample a random bit b which is 1 with probability γ and proceed as follows:
  - If $b = 1$ return $(\texttt{cheat}, j)$ to all parties.
  - If $b = 0$ retrieve $(\texttt{ssid}_1, x)$ and compute $\bar{x} = \lfloor \frac{f(x)}{p} \rfloor + \varepsilon'$, store $(\texttt{ssid}_2, \bar{x})$ and finally return (*accept*) to Adv.

Output: Upon receiving $(\texttt{output}, \texttt{ssid}, \mathcal{P})$ for an existing $\texttt{ssid}$ with $\mathcal{P} \subseteq [n]$ from all parties $i \in [n]$, retrieve $(\texttt{ssid}, x)$ and send x to the adversary Adv. Wait for Adv to return either $\texttt{deliver}$ or $\texttt{abort}$. If the adversary returns $\texttt{deliver}$, output x to all parties in $\mathcal{P}$, otherwise output $\texttt{abort}$.

Abort: Adv may at any point input $\texttt{abort}$ at which point the functionality returns $\texttt{abort}$ to all parties and aborts by not accepting any more calls for the current $\texttt{sid}$.

Fig. 5.1. Ideal functionality for the augmented arithmetic black box

Lemma 4. *Protocol Π_{aABB} in Fig. 5.2 and 5.3 UC-securely implements the ideal functionality $\mathcal{F}_{aABB}$ of Fig. 5.1 in the $\mathcal{F}_{ABB}$-, $\mathcal{F}_{CT}$-hybrid*[5] *model with* robustness *against a* static *and* strong, covert *adversary with deterrence γ and* privacy *against a* static *and* malicious *adversary, corrupting up to $n-1$ parties, with statistical security parameter s for a simulator running polynomially in 2^n time. Under the assumption that the input parameter, x fulfills that: $0 \leq x < pQ - Up$ for $2^s < p < Q \leq \frac{q-(U+1)p}{n(U+1)(2^{\tilde{s}}+1)}$ and $\tilde{s} = s+1$, $0 \leq U \leq n$, $q > p$ with* $\gcd(p, q) = 1$.

To prove lemma 4 we first build a simulator that uses the ideal functionality $\mathcal{F}_{\mathsf{aABB}}$ of Fig. 5.1. Then using a series of game-hops we show that an adversary controlled by the environment interacting between the simulation and the real protocol cannot distinguish except with a statistically small probability.

Remark 4. In our simulation proof, we need to do a game-hop with statistical indistinguishability, which involves arguing that 2 distinct variables are statistically indistinguishable. For this reason, we require $\tilde{s} = s+1$ to represent the statistical distance of s between *each* of the variables. Hence we can use the union bound to argue that the aggregated statistical distance between the distributions is 2^{-s}.

Remark 5. Notice that the ideal functionality chooses both ϵ and ε' from the same distribution as the real protocol in an honest run.

For simplicity, we only prove security for the maximum number of corrupted parties and assume without loss of generality that the honest party is party n. We also for simplicity choose to assume that the adversary never tries to cheat in more than one iteration. It is trivial, though cumbersome, to enhance the simulation to handle an adversary that might cheat in any number of iterations. Let $\mathcal{S}$ be the following simulator simulating

[5] Observe that $\mathcal{F}_{\mathsf{CT}}$ can be realized using $\mathcal{F}_{\mathsf{ABB}}.(\texttt{random}, \texttt{ssid})$.

the honest party, through interaction with the ideal functionality $\mathcal{F}_{\mathsf{aABB}}$. Furthermore, we implicitly assume the simulator keeps track of all messages it receives, in particular, the `ssid`s and we say a `ssid` is fresh if it has not been associated to a value in $\mathcal{F}_{\mathsf{ABB}}(n, p)$ or $\mathcal{F}_{\mathsf{ABB}}(n, q)$. We then define the simulator $\mathcal{S}$ as follows:

$\Pi_{\mathsf{aABB}}(n, p, q, \gamma, s)$

This protocol implements $\mathcal{F}_{\mathsf{aABB}}$ in the $\mathcal{F}_{\mathsf{ABB}}$-, $\mathcal{F}_{\mathsf{CT}}$-hybrid model. It is parametrized by the number of parties participating, n, and the modulo $m = p \cdot q$ for which the computation is over, where $\gcd(p, q) = 1$ and $q > (n+1)(2^{\tilde{s}} + 1)p + n$. Assume implicit access to the instances $\mathcal{F}_{\mathsf{ABB}}(n, p)$ and $\mathcal{F}_{\mathsf{ABB}}(n, q)$.

Rand: Upon receiving $(\texttt{random}, \texttt{ssid})$ for a fresh `ssid` proceed as follows:
1. $[r^{(p)}]_p \leftarrow_R \mathbb{Z}_p$ under derived $\texttt{ssid}_p$.
2. $[r^{(q)}]_q \leftarrow_R \mathbb{Z}_q$ under derived $\texttt{ssid}_q$.

Input: Upon receiving $(\texttt{input}, i, \texttt{ssid}, x)$ for some party $i \in [n]$ and $(\texttt{input}, i, \texttt{ssid}, ?)$ from all other parties $j \in [n] \backslash \{i\}$ for a fresh `ssid`:
1. $[x^{(p)}]_p \leftarrow \textsc{Share}_p(x \mod p)$ under derived $\texttt{ssid}_p$.
2. $[x^{(q)}]_q \leftarrow \textsc{Share}_q(x \mod q)$ under derived $\texttt{ssid}_q$.

Linear: Upon receiving $(\texttt{linear}, \alpha, \beta, \texttt{ssid}_1, \texttt{ssid}_2)$ where $\alpha, \beta \in \mathbb{Z}_m$, $\texttt{ssid}_2$ is fresh and $[x^{(p)}]_p$ and $[x^{(q)}]_q$ exists under derived $\texttt{ssid}_{1,p}$ and $\texttt{ssid}_{1,q}$ respectively, proceed as follows:
1. $[y^{(p)}]_p \leftarrow (\alpha \mod p) \cdot [x^{(p)}]_p + (\beta \mod p)$ under derived $\texttt{ssid}_{2,p}$.
2. $[y^{(q)}]_q \leftarrow (\alpha \mod q) \cdot [x^{(q)}]_q + (\beta \mod q)$ under derived $\texttt{ssid}_{2,q}$.

Add: Upon receiving $(\texttt{add}, \texttt{ssid}_1, \texttt{ssid}_2, \texttt{ssid}_3)$ where $\texttt{ssid}_3$ is fresh and $[x^{(p)}]_p$, $[x^{(q)}]_q$, and $[y^{(p)}]_p, [y^{(q)}]_q$ exist under derived $\texttt{ssid}_{1,p}$, $\texttt{ssid}_{1,q}$ and $\texttt{ssid}_{2,p}, \texttt{ssid}_{2,q}$ respectively, proceed as follows:
1. $[z^{(p)}]_p \leftarrow [x^{(p)}]_p + [y^{(p)}]_p$ under derived $\texttt{ssid}_{3,p}$.
2. $[z^{(q)}]_q \leftarrow [x^{(q)}]_q + [y^{(q)}]_q$ under derived $\texttt{ssid}_{3,q}$.

Mult: Upon receiving $(\texttt{mult}, \texttt{ssid}_1, \texttt{ssid}_2, \texttt{ssid}_3)$ where $\texttt{ssid}_3$ is fresh and $[x^{(p)}]_p$, $[x^{(q)}]_q$, and $[y^{(p)}]_p, [y^{(q)}]_q$ exist under derived $\texttt{ssid}_{1,p}$, $\texttt{ssid}_{1,q}$ and $\texttt{ssid}_{2,p}, \texttt{ssid}_{2,q}$ respectively, proceed as follows:
1. $[z^{(p)}]_p \leftarrow [x^{(p)}]_p \cdot [y^{(p)}]_p$ under derived $\texttt{ssid}_{3,p}$.
2. $[z^{(q)}]_q \leftarrow [x^{(q)}]_q \cdot [y^{(q)}]_q$ under derived $\texttt{ssid}_{3,q}$.

Output:
Upon receiving $(\texttt{output}, \texttt{ssid}, \mathcal{P})$ where $[x^{(p)}]_p$ and $[x^{(q)}]_q$ exist under derived $\texttt{ssid}_p$ and $\texttt{ssid}_q$ and $\mathcal{P} \subseteq [n]$, proceed as follows:
1. $x^{(p)} \leftarrow \textsc{Open}_p([x^{(p)}]_p, \mathcal{P})$ and $x^{(q)} \leftarrow \textsc{Open}_q([x^{(q)}]_q, \mathcal{P})$.
2. Compute and output $x = q \cdot (q^{-1} x^{(p)} \mod p) + p \cdot (p^{-1} \cdot x^{(q)} \mod q) \mod p \cdot q$.

Abort: If at any point $\mathcal{F}_{\mathsf{ABB}}(n, p)$ or $\mathcal{F}_{\mathsf{ABB}}(n, q)$ outputs `abort`, then output `abort` as well.

Fig. 5.2. Protocol realizing $\mathcal{F}_{\mathsf{aABB}}(n, m, s)$

$\Pi_{\mathsf{aABB}}(n,p,q,\gamma,s)$ - ***Continued***

Truncation: Upon receiving $(\mathtt{truncation}, \mathtt{ssid}_1, \mathtt{ssid}_2)$ for a fresh $\mathtt{ssid}_1$ and existing derived $\mathtt{ssid}_{2,p}$ and $\mathtt{ssid}_{2,q}$ from all parties $i \in [n]$, retrieve $[x^{(p)}]_p$ and $[x^{(q)}]_q$ if it can be determined through the previous computations that $0 \le x < pQ - Up$, then preform $\textsc{NoisyTrunc}_1([x]_m) = [\lfloor x/p \rfloor + \varepsilon']_m = ([y]_m)$ through the following steps, letting $\lambda = \lceil 1/(1-\gamma) \rceil$:

1. Do the following input-independent covert preprocessing steps:
   - (a) Each party i samples $\rho_{i,k}, \psi_{i,k} \leftarrow \mathbb{Z}_{(U+1)2^{\bar{s}}Q/p}$ for $k \in [\lambda]$. ▷ Sampling for noise drowning in algorithm 4.
   - (b) $[\rho_{i,k}]_q \leftarrow \textsc{Share}_q(\rho_{i,k})$ and $[\psi_{i,k}]_q \leftarrow \textsc{Share}_q(\psi_{i,k})$ for each $i \in [n]$ and $k \in [\lambda]$.
   - (c) Each party i samples $r_{i,j,k,l} \leftarrow \mathbb{Z}_p$ for $j \in [n]$, $k \in [\lambda]$, $l \in [4]$. ▷ Sampling for needed noise pairs, algorithm 1.
   - (d) $\textsc{Share}_p(r_{i,j,k,l}) \rightarrow [r_{i,j,k,l}]_p$ for each $i,j \in [n]$, $k \in [\lambda]$, $l \in [4]$.
   - (e) Compute $[r_{i,k,l}]_p = \sum_{j\in[n]} [r_{i,j,k,l}]_p$ for $i \in [n]$.
   - (f) $\textsc{Open}_p([r_{i,k,l}]_p, \{i\})$, so party i learns $r_{i,k,l}$, for $i \in [n]$.
   - (g) $\textsc{Share}_q(r_{i,k,l}) \rightarrow [\bar{r}_{i,k,l}]_q$, for $i \in [n]$.
   - (h) $\mathcal{F}_{\mathsf{CT}}.\mathtt{sample}(\mathtt{ssid}, \lambda) \rightarrow c$. ▷ Sample the challenge
   - (i) $\rho_{i,k} \leftarrow \textsc{Open}_q([\rho_{i,k}]_q)$ and $\psi_{i,k} \leftarrow \textsc{Open}_q([\psi_{i,k}]_q)$ for $i \in [n]$ and $k \in [\lambda]\backslash\{c\}$. ▷ Compute and validate the noise drowning randomness.
   - (j) If any $\rho_{i,k}, \psi_{i,k} \ge (U+1)2^{\bar{s}}Q/p$ for $k \in [\lambda]\backslash\{c\}$ then output $(\mathtt{cheat}, i)$ and abort. Otherwise set $[\rho]_q = \sum_{i\in[n]} [\rho_{i,c}]_q$ and $[\psi]_q = \sum_{i\in[n]} [\psi_{i,c}]_q$.
   - (k) For each party $i \in [n]$ compute $r_{i,k,l} \leftarrow \textsc{Open}([r_{i,k,l}]_p)$ and $\bar{r}_{i,k,l} \leftarrow \textsc{Open}([\bar{r}_{i,k,l}]_q)$ for each $k \in [\lambda]\backslash\{c\}$. ▷ Validate and compute the noise pairs
   - (l) If $r_{i,k,l} \neq \bar{r}_{i,k,l}$ output $(\mathtt{cheat}, i)$ and then abort.
   - (m) Compute $[r_{c,l}]_p = \sum_{i\in[n]} [r_{i,c,l}]_p \mod p$ and $[\bar{r}_{c,l}]_q = [r_{c,l} + \epsilon_{c,l} \cdot p]_q = \sum_{i\in[n]} [\bar{r}_{i,c,l}]_q \mod q$, for $i \in [n]$.
   - (n) Define pairs $([r_l]_p, [\bar{r}_l]_q) = ([r_{c,l}]_p, [\bar{r}_{c,l}]_q)$ for $l \in [\lambda]$.
2. $\tilde{x}^{(p)} \leftarrow \textsc{Open}([x^{(p)}]_p + [r_1]_p)$ ▷ The following steps executes algorithm 6
3. $[\bar{x}^{(p)}]_q \leftarrow \tilde{x}^{(p)} - [\bar{r}_1]_q$ ▷ First execution of algorithm 5
4. $[y^{(q)}]_q \leftarrow (p^{-1} \mod q)([x^{(q)}]_q - [\bar{x}^{(p)}]_q)$
5. $\tilde{y}^{(q)} \leftarrow \textsc{Open}_q([y^{(q)}]_q + [\bar{r}_2]_q + [\rho]_q \cdot p)$.
6. $[y^{(p)}]_p \leftarrow (\tilde{y}^{(q)} \mod p) - [r_2]_p$.
7. $[v^{(q)}]_q = (U+1)([x^{(q)}]_q - p[y^{(q)}]_q + pU)$ and $[v^{(p)}]_p = (U+1)([x^{(q)}]_q - p[y^{(q)}]_q + pU)$.
8. $\tilde{v}^{(p)} \leftarrow \textsc{Open}([v^{(p)}]_p + [r_3]_p)$. ▷ Second execution of algorithm 5
9. $[\bar{v}^{(p)}]_q \leftarrow \tilde{v}^{(p)} - [\bar{r}_3]_q$
10. $[w^{(q)}]_q \leftarrow (p^{-1} \mod q)([v^{(q)}]_q - [\bar{v}^{(p)}]_q)$
11. $\tilde{w}^{(q)} \leftarrow \textsc{Open}_q([w^{(q)}]_q + [\bar{r}_4]_q + [\psi]_q \cdot p)$.
12. $[w^{(p)}]_p \leftarrow (\tilde{w}^{(q)} \mod p) - [r_4]_p$.
13. $[\bar{x}^{(q)}]_q = [v^{(q)}]_q - U - P([w^{(q)}]_q)$ and $[\bar{x}^{(p)}]_p = [v^{(p)}]_p - U - P([w^{(p)}]_p)$ ▷ See remark 6
14. Store $\bar{x}^{(p)}$ and $\bar{x}^{(q)}$ under derived $\mathtt{ssid}_2$.

Fig. 5.3. Protocol realizing **Truncation** of $\mathcal{F}_{\mathsf{aABB}}(n,m,s)$

5.1 The simulator, $\mathcal{S}$:

When the ideal functionality is initialized, it needs to be initialized with a deterrence factor γ, where we assume, without loss of generality that $1/(1-\gamma)$ is an integer.

In the description of the protocol, we have been explicit in the distinction between $\mathtt{ssid}$ used in $\mathcal{F}_{\mathsf{aABB}}$ and the $\mathtt{ssid}$'s used in the two different instances of $\mathcal{F}_{\mathsf{ABB}}$. For simplicity, we will abstract that away from now on.

Case 1: Rand – On input $(\mathtt{random}, \mathtt{ssid})$ on $\mathcal{F}_{\mathsf{ABB}}(n,p)$ and $\mathcal{F}_{\mathsf{ABB}}(n,q)$ from any party, relay the message $(\mathtt{random}, \mathtt{ssid})$ to $\mathcal{F}_{\mathsf{aABB}}$. Then internally emulate $\mathcal{F}_{\mathsf{ABB}}(n,p)$ and $\mathcal{F}_{\mathsf{ABB}}(n,q)$ by sampling $x^{(p)} \leftarrow_R \mathbb{F}_p$ and $x^{(q)} \leftarrow_R \mathbb{F}_q$.

Case 2: Input – On input $(\mathtt{input}, i, \mathtt{ssid}, x^{(p)})$ and $(\mathtt{input}, i, \mathtt{ssid}, x^{(q)})$ from corrupt party i with a fresh $\mathtt{ssid}$ on $\mathcal{F}_{\mathsf{ABB}}(n,p)$ and $\mathcal{F}_{\mathsf{ABB}}(n,q)$ respectively, then define $x_i = \phi^{-1}(x_i^{(p)}, x_i^{(q)})$ for ϕ^{-1} as defined in Section 3.1

and relay $(\texttt{input}, i, \texttt{ssid}, x_i)$ to $\mathcal{F}_{\mathsf{aABB}}$. Similarly, on input $(\texttt{input}, i, \texttt{ssid}, ?)$ for all other corrupt parties j, relay the message to $\mathcal{F}_{\mathsf{aABB}}$ and internally store adversarial inputs.

Case 3: Linear – On input $(\texttt{linear}, \texttt{ssid}_1, \texttt{ssid}_2, \alpha^{(p)}, \beta^{(p)})$ for $\alpha^{(p)}, \beta^{(p)} \in \mathbb{F}_p$ and $(\texttt{linear}, \texttt{ssid}_1, \texttt{ssid}_2, \alpha^{(q)}, \beta^{(q)})$ for $\alpha^{(q)}, \beta^{(q)} \in \mathbb{F}_q$ for a fresh $\texttt{ssid}_2$ and existing $\texttt{ssid}_1$ on $\mathcal{F}_{\mathsf{ABB}}(n,p)$ respectively $\mathcal{F}_{\mathsf{ABB}}(n,q)$ from a corrupt party, compute $\alpha = \phi^{-1}(\alpha_i^{(p)}, \alpha_i^{(q)})$ and $\beta = \phi^{-1}(\beta_i^{(p)}, \beta_i^{(q)})$ for ϕ^{-1} as defined in Section 3.1 and relay the message $(\texttt{linear}, \texttt{ssid}_1, \texttt{ssid}_2, \alpha, \beta)$ to $\mathcal{F}_{\mathsf{aABB}}$.

Case 4: Mult – On input $(\texttt{mult}, \texttt{ssid}_1, \texttt{ssid}_2, \texttt{ssid}_3)$ for a fresh $\texttt{ssid}_3$ and existing $\texttt{ssid}_1, \texttt{ssid}_2$ on $\mathcal{F}_{\mathsf{ABB}}(n,p)$, respectively $\mathcal{F}_{\mathsf{ABB}}(n,q)$ from a corrupt party, relay the message $(\texttt{mult}, \texttt{ssid}_1, \texttt{ssid}_2, \texttt{ssid}_3)$ to $\mathcal{F}_{\mathsf{aABB}}$.

Case 5: Truncation – Upon receiving $(\texttt{truncation}, \epsilon, \varepsilon')$ from $\mathcal{F}_{\mathsf{aABB}}$ simulate the calls to $\mathcal{F}_{\mathsf{ABB}}(n,p)$ and $\mathcal{F}_{\mathsf{ABB}}(n,q)$ as described in the simulator for **Rand**, **Input**, **Linear**, **Mult**. $\mathcal{F}_{\mathsf{CT}}.\texttt{sample}$ is simulated to fit the implementation. The remaining non-trivial steps are simulated as follows:

1f Based on the extracted adversarial shares, along with ϵ the simulator samples random shares for the honest party s.t. $\epsilon p = (\sum_{i\in[n]} r_{i,k,l} \mod q) - (\sum_{i\in[n]} r_{i,k,l} \mod p)$ using rejection sampling. Finally, simulate $\textsc{Open}_p([r_{i,k,l}]_p, \{i\})$ to corrupt party i based on the extracted shares and the honest shares.

1g Based on the extracted inputs of the adversary, check if $\bar{r}_{i,k,l} \neq r_{i,k,l}$ for all shares shared by the adversary for all $k \in [\lambda], l \in [4]$ and check if any $\rho_{i,k}, \psi_{i,k} \geq (U+1)2^{\bar{s}}Q/p$. In either case input $(\texttt{cheat}, i, 1, f)$ to the ideal functionality $\mathcal{F}_{\mathsf{aABB}}$ for any such i, save the iteration number as $\ell_l = k$ and whether the cheating was on the noise pairs or the padding. Here f is defined from the value $a = (\sum_{i\in[n-1]} \bar{r}_{i,k,l} \mod q) - (\sum_{i\in[n-1]} r_{i,k,l} \mod p)$ and $\rho_{i,k}, \psi_{i,k}$ s.t. $\bar{x} = \lfloor \frac{f(x)}{p} + \epsilon \rfloor$ [6] Delay the delivery of the response from $\mathcal{F}_{\mathsf{aABB}}$.

1h If $(\texttt{cheat}, i, 1, f)$ was not given as input to $\mathcal{F}_{\mathsf{aABB}}$ in the simulation of step 1g, then input $(\texttt{cheat}, \perp, 0, \perp)$ to $\mathcal{F}_{\mathsf{aABB}}$ and delay the delivery of the response from $\mathcal{F}_{\mathsf{aABB}}$ and emulate $\mathcal{F}_{\mathsf{CT}}.\texttt{sample} \rightarrow c$ as the real functionality $\mathcal{F}_{\mathsf{CT}}$. If cheating occurred in step 1g then wait for $\mathcal{F}_{\mathsf{aABB}}$ to either output $(\texttt{cheat}, i)$ or $(\texttt{accept})$, if $\mathcal{F}_{\mathsf{aABB}}$ outputs $(\texttt{cheat}, i)$ emulate $\mathcal{F}_{\mathsf{CT}}.\texttt{sample}$ to output a random c as in the real functionality but under then constraint that $c \neq \ell_l$. If $\mathcal{F}_{\mathsf{aABB}}$ outputs $(\texttt{accept})$ emulate $\mathcal{F}_{\mathsf{CT}}.\texttt{sample}$ to output a random $c = \ell_l$.

1i Simulate $\textsc{Open}_q(\rho_{i,k})$ and $\textsc{Open}_q(\psi_{i,k})$ based on the extracted values from corrupt parties, and simulate the honest party according the protocol by sampling $\rho_{n,k}$ and $\psi_{n,k}$ randomly from $\mathbb{Z}_{(U+1)2^{\bar{s}}Q/p}$.

1j If $(\texttt{cheat}, i, 1, f)$ was given as input to $\mathit{\Pi}_{\mathsf{aABB}}$ in step 1g and $\mathit{\Pi}_{\mathsf{aABB}}$ gave as output $(\texttt{cheat}, i)$ and the cheating occurred on the padding, then output $(\texttt{cheat}, i)$ and abort.

1k Simulate the opening of values $r_{i,k,l}$ and $\bar{r}_{i,k,l}$ using the extracted and simulated shares.

1l If $(\texttt{cheat}, i, 1, f)$ was given as input to $\mathit{\Pi}_{\mathsf{aABB}}$ in step 1g and $\mathit{\Pi}_{\mathsf{aABB}}$ gave as output $(\texttt{cheat}, i)$ and the cheating occurred on the noise pair, then output $(\texttt{cheat}, i)$ and abort.

2, 8 Simulate a value for x denoted x' and use it to simulate the opening of $\tilde{x}^{(p)}$ using $x'^{(p)}$ and r_1 that $\mathcal{S}$ picked earlier. Similarly for $v^{(p)}$.

5, 11 Simulate the opening of $\tilde{y}^{(q)}$ by using the simulated values previously picked. Similarly for $\tilde{w}^{(q)}$.

Case 6: Output – On input $(\texttt{output}, \texttt{ssid}, \mathcal{P})$ for an existing $\texttt{ssid}$ to $\mathcal{F}_{\mathsf{ABB}}(n,p)$, respectively $\mathcal{F}_{\mathsf{ABB}}(n,q)$ from a corrupt party, relay the message $(\texttt{output}, \texttt{ssid}, \mathcal{P})$ to $\mathcal{F}_{\mathsf{ABB}}(n,p)$ and $\mathcal{F}_{\mathsf{ABB}}(n,q)$. If $i \in \mathcal{P}$ then define $x^{(p)}$, $x^{(q)}$ to be the messages received back from $\mathcal{F}_{\mathsf{ABB}}(n,p)$, respectively $\mathcal{F}_{\mathsf{ABB}}(n,q)$. Then define $x = \phi^{-1}(x^{(p)}, x^{(q)})$ for ϕ^{-1} as defined in Section 3.1 and output $(\texttt{ssid}, x)$.

Case 7: Abort – On input $(\texttt{abort})$ from a corrupt party to either $\mathcal{F}_{\mathsf{ABB}}(n,p)$ or $\mathcal{F}_{\mathsf{ABB}}(n,q)$ relay the message to $\mathcal{F}_{\mathsf{aABB}}$ and all other corrupt parties.

[6] Note that f is always uniquely defined from $a, \rho_{i,k}, \psi_{i,k}, p, q, \epsilon$ since RNSs are linear and a and $\rho_{i,k}, \psi_{i,k}$ cause additive errors in part of the computation.

Indistinguishability: We now argue indistinguishably through a series of game-hops from the simulation above down to the real execution and argue that any adversary controlled by the environment interacting between two consecutive games cannot distinguish except with statistically small probability. During the game-hops, we will assume extraordinary powers in the simulator to simplify the proof process. However, we will ultimately still end up with a sequence of games going from the simulation (*without* any special powers) to the real execution though statistically or perfectly indistinguishable game hops.

Game 0 – Simulation as in 5.1: In this game, the adversary is acting on the simulation described in Section 5.1.

Game 1 – Run basic commands as in the real protocol: In this game, the adversary is talking to a simulator that simulates all commands except **Truncation** as in the real protocol, and **Truncation** as in 5.1. We assume that the simulator has access to all the honest parties' input.

Game 2 – Simulated with the real x: This game runs as the previous one, except the real x is used instead of x'.

Game 3 – A real execution of the protocol: In this game, the adversary is acting on a real execution of the protocol, as described in protocol 5.2 and 5.3.

Lemma 5. *Games 0 and 1 are perfectly indistinguishable*

Proof. That the execution of the basic commands is indistinguishable follows from the fact that $\mathcal{F}_{\mathsf{ABB}}(n,m)$ can be computed securely, that ϕ is bijective, and the fact that the simulator knows all the honest inputs. That $\mathcal{F}_{\mathsf{ABB}}(n,m)$ can be computed securely means that all that is relayed to $\mathcal{F}_{\mathsf{ABB}}(n,p)$ and $\mathcal{F}_{\mathsf{ABB}}(n,q)$, could be relayed to $\prod_{\mathsf{ABB}}(n,p)$ and $\prod_{\mathsf{ABB}}(n,q)$ instead. That ϕ is bijective ensures that the tour around $\mathcal{F}_{\mathsf{aABB}}$ in game 0 makes no difference for the results. That the simulator in game 1 knows all the honest parties' inputs ensures, that it can still simulate the truncation, with the same input values as in game 0. □

Lemma 6. *Game 1 and 2 are statistically indistinguishable with a statistical distance of at most* 2^{-s}

Proof. Notice that the first time x' is used in the simulation is in step 2, so the whole prepossessing is unchanged between these two games. In step 2 and 8 x and v are perfectly masked by the values r_1 and r_3 respectively which are perfectly random and unknown to the distinguisher as long as at least one party is honest. In step 5 and 11 the values $\tilde{y}$ and $\tilde{w}$ are masked by the values $\bar{r}_2$ and ρ, or $\bar{r}_4$ and ψ, respectively. Both $\bar{r}_2$ and $\bar{r}_4$ are perfectly random modulo p, and both ρ and ψ are perfectly random modulo $(U+1)2^{\tilde{s}}Q/p$. Since $y, w < Q$ as shown in Alg. 5 we get that $\tilde{y}$ and $\tilde{w}$ statistically hides y and w respectively with distance $2^{-\tilde{s}}$. This gives a collective statistical distance of at most $2^{-\tilde{s}+1} = 2^{-s}$ as wanted and discussed in Remark 4. □

Lemma 7. *Games 2 and 3 are perfectly indistinguishable under the assumption that the simulator may run in $O(2^n)$time.*

Proof. First notice that the ϵ and ε' used in the simulated preprocessing (game 2) is distributed in the same way as in the real preprocessing (game 3). This is argued to be the case in honest computations in remark 5. In the case of a cheating adversary (succeeding in cheating) any alterations to ε' present in the real execution of the protocol may be absorbed into $f(x)$ in the simulation. Therefore ε' effectively has the same distribution in both cases.

Now notice that if the adversary attempts to cheat then the ideal functionality aborts with probability γ. Since the simulated preprocessing aborts if the ideal functionality aborts, game 1 aborts with probability γ. In the real preprocessing (game 2) the probability of abort (when the adversary attempts to cheat) is $1 - 1/\lambda = 1 - 1/(\lceil 1/(1-\gamma)\rceil) = 1 - 1/(1/(1-\gamma)) = \gamma$, (remember we assumed the ideal functionality is initialized with a deterrence factor where $1/(1-\gamma)$ is an integer).

Most of the simulation runs just as in the real protocol, the only values that are computed differently are c, the honest parties' shares of $r_{i,k,l}$, and $\bar{x}$; the resulting truncated value. From proposition 1 and the assumption that the input parameter x fulfills: $0 \leq x < pQ - Up$ we see that the truncated value has the same distribution in both games. The honest parties shares of $r_{i,k,l}$ are chosen randomly (as in the real protocol) but with the constraint of producing the right ϵ, as ϵ is distributed in the same way in the simulation and real preprocessing. However, to make the shares for the honest party, the simulator does rejection sampling. In the rare case where ϵ is close to either its maximum or minimum value, doing an ordinary rejection sampling is expected to take exponential time in the number of participants. But if the number of participants is limited by some small constant, then the expected running time of the rejection sampling is also upper bounded by a constant.

If the adversary is not attempting to cheat, c is chosen from the same distribution in the real protocol and the simulation. If the adversary does however attempt to cheat (remember we assumed that the adversary only ever attempts to cheat in one iteration), c is chosen under the constraint, that the simulation aborts, if the ideal functionality calls out the cheating. As already argued the ideal functionality calls out the cheating with the right probability and c has the right distribution.

Notice that if the adversary does not attempt to cheat, then all the guarantees of the subprotocols are upheld, and the result is correct in both games. If the adversary attempts to cheat but is caught it results in the process being aborted in both games. If the adversary succeeds in cheating (altering the value of $\bar{r}$, ϵ, ε' or pushing the value of ρ or ψ outside the allowed range[7]) it influences the result of the process in the same way in both games, but in neither game, it reveals additional data. Attempting to cheat can therefore not help the distinguisher to distinguish between the two games.

Using the real x in game 2, x is trivially distributed in the same way as in game 3. Which insures that all values derived from x, ϵ,r_l, $\bar{r}_l$, ρ and ψ has the right distributions. This includes $\tilde{y}$, $\tilde{v}$ and $\tilde{w}$ □

We have now reduced the simulation to the execution of the real protocol. Now lemma 4 follows.

Proof (of lemma 4).

Protocol Π_{aABB} (Fig. 5.2 and 5.3) UC-securely implements the ideal functionality $\mathcal{F}_{\mathsf{aABB}}$ (Fig. 5.1) in the $\mathcal{F}_{\mathsf{ABB}}$-, $\mathcal{F}_{\mathsf{CT}}$-hybrid model with a statistical security parameter of s, for a simulator running polynomially in 2^n time. Under the assumption that the input parameter, x fulfills that: $0 \leq x < pQ - Up$ for $2^s < p < Q \leq \frac{q-(U+1)p}{n(U+1)(2^{\tilde{s}}+1)}$ and $\tilde{s} = s+1$, $0 \leq U \leq n$, $q > p$ with $\gcd(p,q) = 1$. This follows from the statistical indistinguishability between game 0 and game 3 shown in lemmas 5, 6 and 7.
The robustness against a *static* and *strong covert* adversary with deterrence factor γ, corrupting up to $n-1$ parties with deterrence factor γ follows from the ideal functionality, as the protocol is not shown to have greater security properties than the ideal functionality.
The privacy against a *static* and *malicious* adversary corrupting up to $n-1$ parties with statistical security parameter s stems from the fact that the ideal functionality only allows the cheating adversary to alter the result, it does not leak more information then the always allowed ε'. □

6 Efficiency

6.1 Implementation

Note:*After publication we realized that it seems possible to use algorithm 2 in conjunction with the scheme of Catrina and Saxena in order to avoid use of multiplications, with the added cost of an additive error of at most n. Thus the comparison with Catrina and Saxena's algorithm below for fixed-point multiplications is not fair, but we have left it for posterity. Furthermore, we highlight that our scheme is still advantageous for general computation over large domains, has the ability to reduce approximate truncation errors to 1 bit, and finally our approach utilizes a different paradigm for truncation that could prove more efficient in the malicious setting, given more efficient malicious preprocessing algorithms.*

We developed a proof-of-concept implementation[8] of our RNS based MPC scheme and NoisyTrunc$_U$ of algorithm 5 and compare it against SPDZ [DPSZ12] for fixed-point computation when using the algorithm of

[7] Here we do not consider it cheating if the value is not moved outside the allowed range.
[8] Our FRESCO fork is freely available at `https://github.com/jonas-lj/fresco` and our benchmark setup can be found at `https://github.com/jonas-lj/FFTDemo`.

Catrina and Saxena [CS10] for probabilistic truncation. We specifically chose to compare with the probabilistic truncation of Catrina and Saxena because, to the best of our knowledge, it is the most efficient scheme that only requires black-box access to $\mathbb{F}_{\mathsf{ABB}}(n,\cdot)$, with a sufficiently large prime modulo, *and* which is also secure in the dishonest majority setting. Furthermore, this scheme is already implemented in our framework of choice, FRESCO, and hence makes it to make a more fair, apples-to-apples comparison. Concretely their scheme realizes *probabilistic approximate* truncation using a random value, bounded by a certain 2-power, with a known bit-decomposition, which is used to pad the value to truncate. This value is then opened and truncated in plain. The public, truncated value is then input to MPC again and the padding is subtracted and the decomposed random bits are used to account for any overflow that might happen from the random padding. We highlight that both their and our construction can be executed in constant rounds both during the online and preprocessing phases and that both constructions do not require any secure multiplications during the online phase. However, we also forgo the need for secure multiplications in the preprocessing phase.

Our benchmarks consist of micro-benchmarks in multiple network settings and with different-sized computation domains, but also through the real-world application of Fast Fourier Transform (FFT) using the Cooley–Tukey algorithm. All phases of the Catrina and Saxena protocol we benchmark are *maliciously* secure. While our online and triple preprocessing phases are also maliciously secure (see Damgård *et al.* [DEF+19]), our generation of correlated randomness (NoisePairs and Pads) is only secure in the *strong covert* security model for robustness.

We chose to benchmark our protocol with the larger error $\epsilon \leq n$, instead of $\varepsilon' \leq 1$ as practically *efficient* MPC computations are generally only desirable for a small number of parties, such as 2 or 3. Thus, the improvement in error by running algorithm 5 over algorithm 6 is minimal. While the computational punishment is significant as our base algorithms must be run *twice* along with multiple multiplication gates. Thus, we believe this is the desirable practical version, as the fixed-point domain size can be increased a few bits to make any error insignificant in practice. Furthermore, for our chosen real-world application of FFT, we empirically validated that the error in the accuracy of the result when using NoisyTrunc$_U$ was at most $5.81 \cdot 10^{-15}$ for any of our benchmarked setups.

Since our protocol offers security based on a malicious arithmetic secure MPC protocol and supports security for a dishonest majority, we found SPDZ [DPSZ12, CDE+18] to be the most natural competitor and MPC scheme which we can base our underlying $\mathcal{F}_{\mathsf{ABB}}$ on. For this reason, we chose to implement our scheme in the FRESCO [Ale] framework, which is an open-source Java framework for MPC that natively has support for SPDZ. It is designed to allow developers to implement their own MPC back-end and then take advantage of an extensive library of functions such as sorting, searching, and statistics, which can be used to design real-world MPC applications. Furthermore, FRESCO has been used extensively in other academic works [DDN+16, ABB+17, DEF+19, BDF21].

Table 2. Domain sizes used in the benchmarks, assuming 3 parties and at least 39 bits of statistical security. Column "Usable" expresses the usable amount of bits.

$\log(m)$	Ours			[CS10]
	$\log(p)$	$\log(q)$	Usable	Usable
136	40	96	91	95
192	56	136	147	151
256	88	168	211	215
512	216	296	467	471
1536	728	808	1491	1495

Code design. We wrote our code as a new MPC back-end for FRESCO, aggregating two SPDZ instances, of appropriate moduli and using these to implement the basic arithmetic operations required by an MPC scheme (input, output, addition, linear operations, and multiplications). We then wrote our efficient, approximate truncation function and integrated this with the existing FRESCO code for performing fixed-point arithmetic. (see section 3.2 for details on fixed-point computation in MPC and section 4.3 for a discussion of representing negative numbers in our scheme). Our code uses no multi-threading on top of what is implicitly done in FRESCO, and we observe that FRESCO only takes advantage of multi-threading insofar as to allow asynchronous networking and in certain select locations implicitly through the Java class `ParallelStream`. As our code is a proof-of-concept, there are still plenty of places it can be improved; in particular, using common seeds of randomness between each pair of parties could be used to limit communication when computing NoisePairs and Pads.

Table 3. Timing in seconds for **1024** *regular* multiplications and triple preprocessing for both our scheme and SPDZ for 2 parties with domains with various *bits* available for computation. $s \geq 39$. The best numbers are marked in bold.

	Triple preprocessing		Integer multiplication	
$\log(m)$	Ours (RNS SPDZ)	SPDZ	Ours (RNS SPDZ)	SPDZ
	LAN, latency 0.43ms			
136	11.6 ± 0.21	$\mathbf{10.7 \pm 0.18}$	0.197 ± 0.014	$\mathbf{0.190 \pm 0.029}$
192	$\mathbf{14.8 \pm 0.29}$	16.6 ± 0.48	0.205 ± 0.012	$\mathbf{0.180 \pm 0.008}$
256	$\mathbf{19.1 \pm 0.32}$	20.7 ± 0.18	0.206 ± 0.006	$\mathbf{0.184 \pm 0.004}$
512	$\mathbf{42.3 \pm 0.70}$	54.9 ± 1.3	0.275 ± 0.015	$\mathbf{0.214 \pm 0.008}$
1536	$\mathbf{233 \pm 1.8}$	462	0.467 ± 0.017	$\mathbf{0.337 \pm 0.003}$
	WAN, latency 10 ms			
136	12.8 ± 1.1	$\mathbf{11.2 \pm 0.36}$	0.392 ± 0.136	$\mathbf{0.247 \pm 0.046}$
192	$\mathbf{15.8 \pm 0.37}$	17.3 ± 0.43	0.311 ± 0.079	$\mathbf{0.248 \pm 0.012}$
256	$\mathbf{21.2 \pm 1.3}$	$\mathbf{21.2 \pm 0.28}$	0.262 ± 0.028	$\mathbf{0.246 \pm 0.015}$
512	$\mathbf{44.5 \pm 1.4}$	56.1 ± 1.2	0.326 ± 0.010	$\mathbf{0.286 \pm 0.012}$
1536	$\mathbf{243 \pm 3.7}$	494	0.535 ± 0.012	$\mathbf{0.420 \pm 0.008}$

Table 4. Timing in seconds for **1024** *fixed-point* multiplications and preparation of the *correlated randomness* required for this (NoisePair and Pad for our scheme with $\gamma = 2$ and random bit decomposition for SPDZ) for 2 parties with domains with various *bits* available for computation. Column #Triples express how many preprocessed multiplication triples are required for **1024** fixed-point multiplications. $s \geq 39$ The best numbers are marked in bold.

	#Triples		Correlated randomness		Fixed-point multiplication	
$\log(m)$	Ours	[CS10]	Ours	[CS10]	Ours	[CS10]
	LAN, latency 0.43ms					
136	**1024**	$41,984$	0.355 ± 0.018	$\mathbf{0.263 \pm 0.006}$	$\mathbf{0.363 \pm 0.010}$	1.17 ± 0.015
192	**1024**	$58,368$	0.368 ± 0.011	$\mathbf{0.264 \pm 0.005}$	$\mathbf{0.365 \pm 0.007}$	1.42 ± 0.029
256	**1024**	$91,136$	0.365 ± 0.014	$\mathbf{0.312 \pm 0.007}$	$\mathbf{0.395 \pm 0.007}$	1.68 ± 0.027
512	**1024**	$222,208$	$\mathbf{0.401 \pm 0.062}$	0.669 ± 0.004	$\mathbf{0.438 \pm 0.016}$	2.82 ± 0.044
1536	**1024**	$746,496$	$\mathbf{0.505 \pm 0.023}$	8.86 ± 0.077	$\mathbf{0.787 \pm 0.010}$	11.0 ± 0.042
	WAN, latency 10 ms					
136	**1024**	$41,984$	0.438 ± 0.008	$\mathbf{0.307 \pm 0.005}$	$\mathbf{0.402 \pm 0.014}$	4.99 ± 0.32
192	**1024**	$58,368$	0.453 ± 0.014	$\mathbf{0.314 \pm 0.006}$	$\mathbf{0.413 \pm 0.014}$	6.37 ± 0.10
256	**1024**	$91,136$	0.455 ± 0.014	$\mathbf{0.369 \pm 0.016}$	$\mathbf{0.441 \pm 0.013}$	8.15 ± 0.12
512	**1024**	$222,208$	$\mathbf{0.476 \pm 0.020}$	0.766 ± 0.008	$\mathbf{0.487 \pm 0.013}$	15.1 ± 0.35
1536	**1024**	$746,496$	$\mathbf{0.608 \pm 0.024}$	8.84 ± 0.042	$\mathbf{0.849 \pm 0.021}$	35.6 ± 0.42

Experimental setup. We ran all our experiments on AWS EC2 t2.xlarge servers located in Paris, Frankfurt, and London and observed that the average latency between any two servers is between 10-15 ms and an average of 0.43 ms when in the same data center. We observe that each of these machines has 4 virtual cores on Intel Xeon CPUs and 16 GiB of RAM. All machines were running Amazon Linux Coretto and OpenJDL 17. Network communication (even on same-machine tests) is done using a standard TCP/IP socket, *without* adding TLS or securing layers on top. Numbers are based on the average of *at least* 10 iterations and errors are the standard deviation. The only exception is triple preprocessing for SPDZ for a domain of 1536 bits, which is only done once.

In all the benchmarks, we have used the same overall choice of domain size, m. However, some slack in the computation space is needed to be able to carry out the probabilistic truncation correctly, both in our scheme and the one by Catrina and Saxena. For Catrina and Saxena this reduction is $n + s$ bits, and for our scheme, it is approximately $2n + \tilde{s}$ as discussed in Sec. 4.3. We show the concrete effect of this for our benchmarks in table 2. Finally, observe that for all benchmarks we have ensured that $s \geq 39$.

Micro benchmarks. We took a micro-benchmark approach to our implementation, letting it consist of several interchangeable components for different levels of preprocessing. We did so, to allow us to isolate bottlenecks, in our benchmarks. Hence, whatever is not benchmarked in a given test is emulated with a trivial dummy implementation. Concretely we obtain the following micro-benchmarks

Triple preprocessing Preprocessing of multiplication triples for both Catrina and Saxena's and our scheme. This can be done before the input or function to compute is known. We based it on MASCOT [KOS16], as this is currently the only multiplication triple preprocessing supported by FRESCO. However, more efficient approaches have been presented since MASCOT [KPR18], so these numbers should be considered an upper bound.

Correlated randomness For our scheme this involves preprocessing of NoisePairs and Pads; the process of algorithm 1 and algorithm 2. For Catrina and Saxena this involves bit-decomposition of a random number of $\lceil \log_2(p) \rceil$ bits, (*excluding* the preprocessing of $\lceil \log_2(p) \rceil$ triples which is needed for the sampling of random bits) needed for the approximate truncation [CS10]. This phase can be done offline at the same time as triple preprocessing.

Fixed-point multiplication Online time of fixed-point multiplication with base p and hence $\log(p)$ bits precision. Thus $\log(p) + 1$ preprocessed multiplication triples are required for Catrina and Saxena's protocol and 1 for our protocol (along with a correlated randomness element). Both protocols only require 1 multiplication to be executed during the online phase.

Integer multiplication Online time of pure *integer* multiplications in MPC. This requires a preprocessed multiplication triple for both protocols.

We express these micro-benchmarks in table 3 and 4. From table 3, we can conclude that triple preprocessing becomes cheaper for our RNS scheme compared to SPDZ, the larger m gets. Whereas for the online time for multiplications, our RNS scheme is slightly worse than SPDZ. While the first observation is expected, the second is surprising as we intuitively would expect our scheme to perform comparatively better for larger domains. This is because computation over Z_p and $\mathbb{Z}_q$ and generally more efficient than computation over $\mathbb{Z}_{pq}$, assuming pq does not fit within a word. This has been the motivation for several previous usages of RNSs [Qui82, AHK17, JMR22]. From table 4, when it comes to the correlated randomness, we again observe that our scheme is more efficient for larger m, whereas Catrina and Saxena's scheme is more efficient for smaller m. Concerning the online *fixed-point* multiplication time we see that our scheme is significantly faster than the one by Catrina and Saxena, *and* that it scales more gracefully for larger m. It is not unexpected that our scheme performs better for fixed-point computation (even excluding triple preprocessing) first; for the reason of more efficient computation over the smaller $\mathbb{Z}_p$ and $\mathbb{Z}_q$ domains, but also since our online truncation computation does not need to perform $O(\log(p))$ bit-fiddling operations, like Catrina and Saxena's scheme. Concerning the generation of correlated randomness, it is hard to predict how our scheme would fare against the other scheme since the approaches are so different. Although for similar reasons as above, we did expect it to scale better relatively, compared to Catrina and Saxena's scheme, which the benchmarks confirm. We give more detail about the time it takes to preprocess our correlated randomness in Fig. 2. This figure shows how the generation of correlated randomness scales with different choices of deterrence factors, both on LAN and WAN. More specifically it shows that network latency has a minimal effect on the time, as expected due to the protocol being constant round. As expected this loosely mimics a cost function $\lambda(\gamma) = \lceil 1/(1-\gamma) \rceil$, which reflects the number of times the heaviest parts of the preprocessing must be carried out as γ increases. More surprisingly it also shows that there is barely any performance penalty for larger computational domains, up to $\log(m) = 512$.

Concerning the choice of domain, we picked the smallest possible (with reasonable statistical security, i.e. $s \geq 39$) that is supported by our scheme, along with certain larger sizes and the *largest* supported by FRESCO (domain size 1536 bits). While on the short side, the largest parameter also shows that using our scheme for general integer computation over larger domains is advantageous when counting total execution time, i.e. both online and preprocessing time together. Large domain computations are for example needed in distributed RSA key generation [FLOP18, CDK+22]. We would also expect the online time to be more efficient over large domains, this does not occur in our benchmarks. We believe the reason is that the overhead

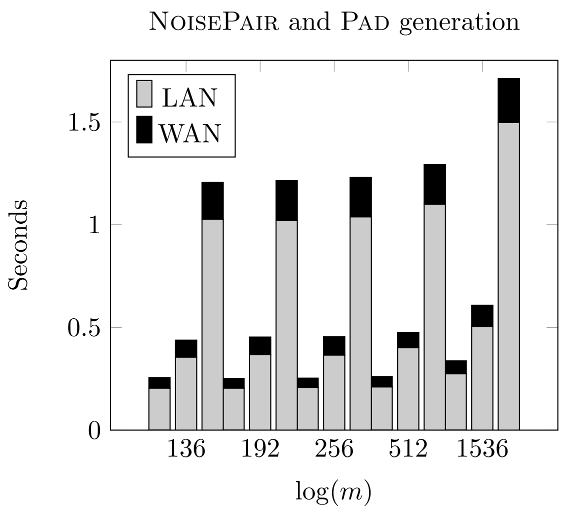

Fig. 2. Computation time in seconds for noise pair and padding preprocessing needed for **1024** truncations for 2 parties. $s \geq 39$. Left-side bars represent a semi-honest preprocessing ($\gamma = 0$), middle bars represent $\gamma = 0.5$ and right-side bars represent $\gamma = 0.875$.

required by using 2 MPC instances outshines the computational advantage in doing multiplication over smaller domains.

We observe that online time for fixed-point multiplications is between 3.2-42x faster using our scheme depending on the computation domain (as seen in the two right-most columns of table 4). However, if we include the preprocessing time, our scheme is between 36-1,400x faster[9]. The reason for such a significant difference is that our scheme only requires 1 triple per fixed-point multiplication and Catrina and Saxena's protocol requires $1 + \ell$ triples where 2^ℓ is the domain size of the fractional digits, where $\ell = \lceil \log_2(p) \rceil$ in our benchmarks to allow a direct comparison.

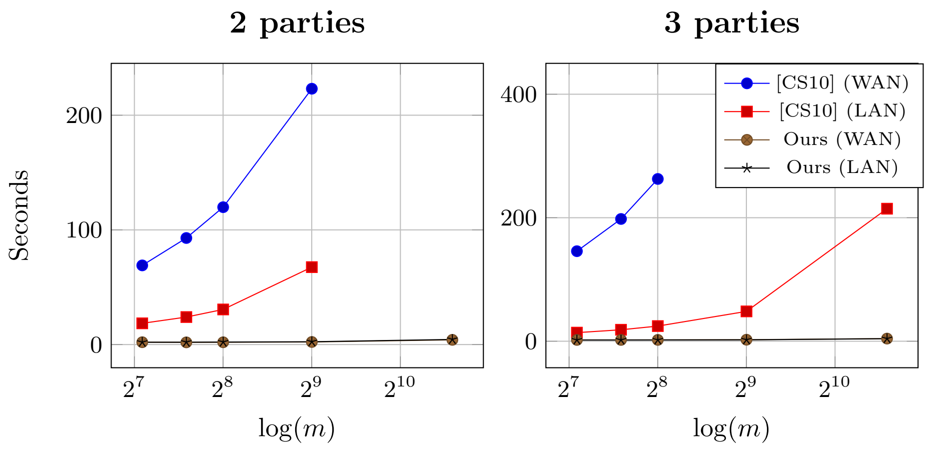

Fig. 3. Online performance of computing an FFT of **1024** inputs for different amount of parties in different network settings. $s \geq 39$. Latency for 2 servers is 10 ms and 10-14 ms for 3 servers.

[9] The factors are for $\gamma = 0.5$ and depend on the size of the domain and whether execution is over WAN/LAN. Concretely the factors are computed by taking the number of triples required for truncation from table 4 and multiplying with the preprocessing time from table 3 and adding the online time (again from table 4).

FFT benchmark. To consider our scheme in a realistic setting, we benchmarked the Fast Fourier Transform for various input sizes using the Cooley-Tukey Algorithm. This use-case is primarily chosen because of its pervasive appearance in computation such as signal processing or convolution neural networks, the latter of which has also been studied in the setting of secure computation [LXZ+20]. Furthermore, FFT computations are well-suited for residue number systems because they only use multiplication and addition. The input to the computation is a vector of complex numbers, each of which is in our implementation represented by two fixed-point numbers, one for the real part and one for the imaginary part. We show the online time for evaluating Cooley-Tukey on 1024 inputs using our scheme in Fig. 3. More specifically the figure shows the increase in time as the computation domain increases, for 2 and 3 parties. The base for the fixed-point computation is p and hence depends on the domain m, as defined in table 2. The computation requires $O(|\text{input}| \log(|\text{input}|))$ multiplications executed over $O(\log(|\text{input}|))$ rounds. We here observe that while each of these multiplications consists of a secret value and public value, we still require a truncation after each multiplication since the input is a fixed-point number. Thus, our scheme does not require multiplication triples whereas the one by Catrina and Saxena does. From the figure we observe that our scheme is significantly faster for FFT than the one by Catrina and Saxena as we would expect because of the reasons and findings from the "Micro benchmark" section, but exacerbated by the fact that our scheme does not need to multiply two secret numbers in MPC.

6.2 Comparison with related techniques

Several schemes for efficient probabilistic truncation exist that perform better than the work of Catrina and Saxena [CS10], but also require access to a functionality $\mathbb{F}_{\mathsf{ABB}}(n, 2)$. While it is possible to emulate operations modulo 2 in large fields, addition (XOR) requires a multiplication. The same is true for sampling of a random bit. Furthermore, schemes working over modulo 2 are typically significantly more efficient than schemes working over a large modulo p [LOS14, FKOS15, HSS17]. Hence the possibility of sampling random bits using $\mathbb{F}_{\mathsf{ABB}}(n, 2)$ and moving these to $\mathbb{F}_{\mathsf{ABB}}(n, p)$ could lead to a more efficient version of the Catrina and Saxena protocol. The line of work trying to achieve this starts with the ABY protocol [DSZ15] which is a semi-honestly secure two-party protocol allowing mixed computation over bits and large domains when garbled circuits are used for bit computation. This was later extended to the malicious security model for 3 parties [MR18]. Later Rotaru and Wood [RW19] showed an efficient protocol for generating and computing over bits in an MPC scheme working modulo a large p with the help of garbled circuits. Their protocol works for an arbitrary amount of parties, in the dishonest majority setting against a malicious adversary. They coined the term *daBits* for a pair $([b]_2, [b]_p)$ where p is large. Several works improve upon this construction [AOR+19, RST+22, BST20], culminating in the work by Escudero *et al.* [EGK+20]. They show a more efficient protocol for daBits which is maliciously secure against a dishonest majority that only requires black-box access to $\mathbb{F}_{\mathsf{ABB}}(n, 2)$ and $\mathbb{F}_{\mathsf{ABB}}(n, p)$. However, they also show how to extend the daBit notion to extended daBits (*edaBits*), which is a representation of $(\{[b_i]_2\}_i, [r]_p)$ s.t. $\sum_i 2^i \cdot [b_i]_2 = [r]_p$. That is, a full bit-decomposition of a random number modulo p, represented by its binary parts. This allows a more efficient execution of the probabilistic truncation protocol of Catrina and Saxena. For completeness, we mention that certain other computations that require bit-decomposition can be realized even faster using edaBits, than by adapting the existing protocols of Catrina, Saxena and de Hoogh. This is specifically the case for comparison when using the Rabbit protocol [MRVW21].

It is hard to make an apples-to-apples efficiency comparison between these schemes and ours (and the one by Catrina and Saxena) due to their need for $\mathcal{F}_{\mathsf{ABB}}(n, 2)$. This is because $\mathcal{F}_{\mathsf{ABB}}(n, 2)$ can typically not be realized using the same techniques as for $\mathcal{F}_{\mathsf{ABB}}(n, p)$ with $p > 2^s$. However, in table 5 we try to compare the different schemes asymptotically, based on the heavy computations (multiplications in $\mathbb{F}_{\mathsf{ABB}}$ for $\mathbb{Z}_m$ and $\mathbb{F}_2$)[10].

[10] Observe that edaBits require many different components to achieve their efficient result. This includes *faulty multiplications* in MPC which are about $O(B)$ times more efficient than a "normal" multiplication in MPC. Here $B \in \{3, 4, 5\}$ depending on an amortization parameter. In the table, we have for simplicity only counted real multiplications and assumed $O(B)$ faulty multiplications are equivalent to a real one.

Table 5. Complexity comparison between our scheme and other approaches for a single probabilistic truncation in the amortized setting when truncating k bits of values over $\mathbb{Z}_m$. Row *Trip.* expresses the amount of preprocessed triples needed, whereas *Offline rounds* expresses the rounds of communication needed which is independent of the private inputs, and finally *Online* $\mathbb{F}_2/\mathbb{Z}_m$ expresses the multiplications required to compute truncation.

		[CS10]	daBits [EGK+20]	edaBits [EGK+20]	Ours
Trip.	$\mathbb{F}_2$	0	$O(\log(n) \cdot (k+s)$	$O(\log(n)^2 + \log(n) \cdot (n+k)$	0
	$\mathbb{Z}_m$	k	0	0	0
Offline rounds		1	$O(\log(k))$	$O(k)$	5
Online	$\mathbb{F}_2$	0	0	0	0
	$\mathbb{Z}_m$	0	0	0	0
	Rounds	1	1	1	2

Furthermore, we observe that the edaBits authors find a 5-9x improvement in computing the "comparison" operation, compared to the arithmetic approach of Catrina and de Hoogh [Cd10] when *including* preprocessing. While comparison is not the same as probabilistic truncation, the main bottleneck of both protocols is the bit-decomposition of a random element in $\mathbb{F}_m$. Hence we believe a similar improvement would be found for probabilistic truncation. Thus we expect that our results will still be about a factor 4-280x more efficient than the edaBit approach when including preprocessing and using the "comparison" improvement factor verbatim.

It is worth emphasizing that both the online and preprocessing approach by Catrina and Saxena, daBits, and edaBits are maliciously secure against a dishonest majority. Our online phase is also maliciously secure, but our concrete suggestion for realizing the preprocessing phase is done in a strong covert security model for robustness. Furthermore, the possible error in the truncation of Catrina and Saxena is at most 1, whereas our error is at most $1 \leq U$. Finally, we require working in a domain of $o(p^3)$ bits (spread on two different MPC schemes working modulo p and modulo $q > p^2$), which gives us $o(p^2)$ usable bits in the secret shared value that gets truncated. Catrina and Saxena, daBits, and edaBits also require a gap in the usable bits of 2^s, hence they need a domain of $o(p^2)$ bits when $p \approx 2^s$.

We leave as future work the possibility of incorporating daBit and edaBit techniques to sample random and correlated values in different domains in a manner that works with our protocols.

References

ABB+17. José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, Benjamin Grégoire, Vincent Laporte, and Vitor Pereira. A fast and verified software stack for secure function evaluation. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 1989–2006. ACM Press, October / November 2017.

ABL+04. Mikhail J. Atallah, Marina Bykova, Jiangtao Li, Keith B. Frikken, and Mercan Topkara. Private collaborative forecasting and benchmarking. In Vijay Atluri, Paul F. Syverson, and Sabrina De Capitani di Vimercati, editors, *Proceedings of the 2004 ACM Workshop on Privacy in the Electronic Society, WPES 2004, Washington, DC, USA, October 28, 2004*, pages 103–114. ACM, 2004.

ABZS13. Mehrdad Aliasgari, Marina Blanton, Yihua Zhang, and Aaron Steele. Secure computation on floating point numbers. In *NDSS 2013*. The Internet Society, February 2013.

ACS02. Joy Algesheimer, Jan Camenisch, and Victor Shoup. Efficient computation modulo a shared secret with application to the generation of shared safe-prime products. In Moti Yung, editor, *CRYPTO 2002*, volume 2442 of *LNCS*, pages 417–432. Springer, Heidelberg, August 2002.

ADEN21. Mark Abspoel, Anders P. K. Dalskov, Daniel Escudero, and Ariel Nof. An efficient passive-to-active compiler for honest-majority MPC over rings. In Kazue Sako and Nils Ole Tippenhauer, editors, *ACNS 21, Part II*, volume 12727 of *LNCS*, pages 122–152. Springer, Heidelberg, June 2021.

AHK17. Shahzad Asif, Md. Selim Hossain, and Yinan Kong. High-throughput multi-key elliptic curve cryptosystem based on residue number system. *IET Comput. Digit. Tech.*, 11(5):165–172, 2017.

AL10. Yonatan Aumann and Yehuda Lindell. Security against covert adversaries: Efficient protocols for realistic adversaries. *Journal of Cryptology*, 23(2):281–343, April 2010.

Ale. Alexandra Institute. FRESCO - a FRamework for Efficient Secure COmputation. `https://github.com/aicis/fresco`.

AOR+19. Abdelrahaman Aly, Emmanuela Orsini, Dragos Rotaru, Nigel P. Smart, and Tim Wood. Zaphod: Efficiently combining LSSS and garbled circuits in SCALE. In Michael Brenner, Tancrède Lepoint, and Kurt Rohloff, editors, *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography, WAHC@CCS 2019, London, UK, November 11-15, 2019*, pages 33–44. ACM, 2019.

BCD+09. 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, *FC 2009*, volume 5628 of *LNCS*, pages 325–343. Springer, Heidelberg, February 2009.

BCDF22. Carsten Baum, James Hsin-yu Chiang, Bernardo David, and Tore Kasper Frederiksen. Eagle: Efficient privacy preserving smart contracts. *IACR Cryptol. ePrint Arch.*, page 1435, 2022.

BCG+21. Elette Boyle, Nishanth Chandran, Niv Gilboa, Divya Gupta, Yuval Ishai, Nishant Kumar, and Mayank Rathee. Function secret sharing for mixed-mode and fixed-point secure computation. In Anne Canteaut and François-Xavier Standaert, editors, *EUROCRYPT 2021, Part II*, volume 12697 of *LNCS*, pages 871–900. Springer, Heidelberg, October 2021.

BCT21. Aritra Banerjee, Michael Clear, and Hitesh Tewari. zkHawk: Practical private smart contracts from MPC-based hawk. Cryptology ePrint Archive, Report 2021/501, 2021. `https://eprint.iacr.org/2021/501`.

BDF21. Carsten Baum, Bernardo David, and Tore Kasper Frederiksen. P2DEX: Privacy-preserving decentralized cryptocurrency exchange. In Kazue Sako and Nils Ole Tippenhauer, editors, *ACNS 21, Part I*, volume 12726 of *LNCS*, pages 163–194. Springer, Heidelberg, June 2021.

BDOZ11. Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. Semi-homomorphic encryption and multiparty computation. In Kenneth G. Paterson, editor, *EUROCRYPT 2011*, volume 6632 of *LNCS*, pages 169–188. Springer, Heidelberg, May 2011.

BIB89. Judit Bar-Ilan and Donald Beaver. Non-cryptographic fault-tolerant computing in constant number of rounds of interaction. In Piotr Rudnicki, editor, *8th ACM PODC*, pages 201–209. ACM, August 1989.

BOS16. Carsten Baum, Emmanuela Orsini, and Peter Scholl. Efficient secure multiparty computation with identifiable abort. In Martin Hirt and Adam D. Smith, editors, *TCC 2016-B, Part I*, volume 9985 of *LNCS*, pages 461–490. Springer, Heidelberg, October / November 2016.

BST20. Charlotte Bonte, Nigel P. Smart, and Titouan Tanguy. Thresholdizing HashEdDSA: MPC to the rescue. Cryptology ePrint Archive, Report 2020/214, 2020. `https://eprint.iacr.org/2020/214`.

Can01. Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd Annual Symposium on Foundations of Computer Science, FOCS 2001, 14-17 October 2001, Las Vegas, Nevada, USA*, pages 136–145. IEEE Computer Society, 2001.

CCD+20. Megan Chen, Ran Cohen, Jack Doerner, Yashvanth Kondi, Eysa Lee, Schuyler Rosefield, and abhi shelat. Multiparty generation of an RSA modulus. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part III*, volume 12172 of *LNCS*, pages 64–93. Springer, Heidelberg, August 2020.

Cd10. Octavian Catrina and Sebastiaan de Hoogh. Improved primitives for secure multiparty integer computation. In Juan A. Garay and Roberto De Prisco, editors, *SCN 10*, volume 6280 of *LNCS*, pages 182–199. Springer, Heidelberg, September 2010.

CDE+18. Ronald Cramer, Ivan Damgård, Daniel Escudero, Peter Scholl, and Chaoping Xing. SPD $\mathbb{Z}_{2^k}$: Efficient MPC mod 2^k for dishonest majority. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part II*, volume 10992 of *LNCS*, pages 769–798. Springer, Heidelberg, August 2018.

CDK+22. Megan Chen, Jack Doerner, Yashvanth Kondi, Eysa Lee, Schuyler Rosefield, abhi shelat, and Ran Cohen. Multiparty generation of an RSA modulus. *Journal of Cryptology*, 35(2):12, April 2022.

CDPW07. Ran Canetti, Yevgeniy Dodis, Rafael Pass, and Shabsi Walfish. Universally composable security with global setup. In Salil P. Vadhan, editor, *TCC 2007*, volume 4392 of *LNCS*, pages 61–85. Springer, Heidelberg, February 2007.

CGOS21. Nishanth Chandran, Divya Gupta, Sai Lakshmi Bhavana Obbattu, and Akash Shah. SIMC: ML inference secure against malicious clients at semi-honest cost. Cryptology ePrint Archive, Report 2021/1538, 2021. `https://eprint.iacr.org/2021/1538`.

CHI+21. Megan Chen, Carmit Hazay, Yuval Ishai, Yuriy Kashnikov, Daniele Micciancio, Tarik Riviere, abhi shelat, Muthu Venkitasubramaniam, and Ruihan Wang. Diogenes: Lightweight scalable RSA modulus generation with a dishonest majority. In *2021 IEEE Symposium on Security and Privacy*, pages 590–607. IEEE Computer Society Press, May 2021.

CL17. Ran Cohen and Yehuda Lindell. Fairness versus guaranteed output delivery in secure multiparty computation. *Journal of Cryptology*, 30(4):1157–1186, October 2017.

CR07. C.C.S. Caiado and P.N. Rathie. Polynomial coefficients and distribution of the sum of discrete uniform variables. In A. M. Mathai, M. A. Pathan, K. K. Jose, and Joy Jacob, editors, *Eighth Annual Conference of the Society of Special Functions and their Applications*. Society for Special Functions & their Applications, January 2007.

CS10. Octavian Catrina and Amitabh Saxena. Secure computation with fixed-point numbers. In Radu Sion, editor, *FC 2010*, volume 6052 of *LNCS*, pages 35–50. Springer, Heidelberg, January 2010.

DA01. Wenliang Du and Mikhail J. Atallah. Privacy-preserving cooperative statistical analysis. In *17th Annual Computer Security Applications Conference (ACSAC 2001), 11-14 December 2001, New Orleans, Louisiana, USA*, pages 102–110. IEEE Computer Society, 2001.

DCT$^+$18. Maxim Deryabin, Nikolay Chervyakov, Andrei Tchernykh, Mikhail Babenko, and Mariia Shabalina. High performance parallel computing in residue number system. *International Journal of Combinatorial Optimization Problems and Informatics*, 9(1):62–67, Feb. 2018.

DDN$^+$16. Ivan Damgård, Kasper Damgård, Kurt Nielsen, Peter Sebastian Nordholt, and Tomas Toft. Confidential benchmarking based on multiparty computation. In Jens Grossklags and Bart Preneel, editors, *FC 2016*, volume 9603 of *LNCS*, pages 169–187. Springer, Heidelberg, February 2016.

DEF$^+$19. Ivan Damgård, Daniel Escudero, Tore Kasper Frederiksen, Marcel Keller, Peter Scholl, and Nikolaj Volgushev. New primitives for actively-secure MPC over rings with applications to private machine learning. In *2019 IEEE Symposium on Security and Privacy*, pages 1102–1120. IEEE Computer Society Press, May 2019.

DEN22. Anders P. K. Dalskov, Daniel Escudero, and Ariel Nof. Fast fully secure multi-party computation over any ring with two-thirds honest majority. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 653–666. ACM Press, November 2022.

DGN$^+$17. Nico Döttling, Satrajit Ghosh, Jesper Buus Nielsen, Tobias Nilges, and Roberto Trifiletti. TinyOLE: Efficient actively secure two-party computation from oblivious linear function evaluation. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 2263–2276. ACM Press, October / November 2017.

DKL$^+$13. Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In Jason Crampton, Sushil Jajodia, and Keith Mayes, editors, *ESORICS 2013*, volume 8134 of *LNCS*, pages 1–18. Springer, Heidelberg, September 2013.

DMRT21. Cyprien Delpech de Saint Guilhem, Eleftheria Makri, Dragos Rotaru, and Titouan Tanguy. The return of eratosthenes: Secure generation of RSA moduli using distributed sieving. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 594–609. ACM Press, November 2021.

DOK$^+$20. Anders P. K. Dalskov, Claudio Orlandi, Marcel Keller, Kris Shrishak, and Haya Shulman. Securing DNSSEC keys via threshold ECDSA from generic MPC. In Liqun Chen, Ninghui Li, Kaitai Liang, and Steve A. Schneider, editors, *ESORICS 2020, Part II*, volume 12309 of *LNCS*, pages 654–673. Springer, Heidelberg, September 2020.

DPSZ12. Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 643–662. Springer, Heidelberg, August 2012.

DSZ15. Daniel Demmler, Thomas Schneider, and Michael Zohner. ABY - A framework for efficient mixed-protocol secure two-party computation. In *NDSS 2015*. The Internet Society, February 2015.

EGK$^+$20. Daniel Escudero, Satrajit Ghosh, Marcel Keller, Rahul Rachuri, and Peter Scholl. Improved primitives for MPC over mixed arithmetic-binary circuits. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part II*, volume 12171 of *LNCS*, pages 823–852. Springer, Heidelberg, August 2020.

FDH$^+$10. Martin Franz, Björn Deiseroth, Kay Hamacher, Somesh Jha, Stefan Katzenbeisser, and Heike Schröder. Secure computations on non-integer values. In *2010 IEEE International Workshop on Information Forensics and Security, WIFS 2010, Seattle, WA, USA, December 12-15, 2010*, pages 1–6. IEEE, 2010.

FK11. Martin Franz and Stefan Katzenbeisser. Processing encrypted floating point signals. In Chad Heitzenrater, Scott Craver, and Jana Dittmann, editors, *Proceedings of the thirteenth ACM multimedia workshop on Multimedia and security, MM&Sec '11, Buffalo, New York, USA, September 29-30, 2011*, pages 103–108. ACM, 2011.

FKOS15. Tore Kasper Frederiksen, Marcel Keller, Emmanuela Orsini, and Peter Scholl. A unified approach to MPC with preprocessing using OT. In Tetsu Iwata and Jung Hee Cheon, editors, *ASIACRYPT 2015, Part I*, volume 9452 of *LNCS*, pages 711–735. Springer, Heidelberg, November / December 2015.

FLOP18. Tore Kasper Frederiksen, Yehuda Lindell, Valery Osheter, and Benny Pinkas. Fast distributed RSA key generation for semi-honest and malicious adversaries. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part II*, volume 10992 of *LNCS*, pages 331–361. Springer, Heidelberg, August 2018.

FPBS16. Apostolos P. Fournaris, Louiza Papachristodoulou, Lejla Batina, and Nicolas Sklavos. Residue number system as a side channel and fault injection attack countermeasure in elliptic curve cryptography. In *2016 International Conference on Design and Technology of Integrated Systems in Nanoscale Era, DTIS 2016, Istanbul, Turkey, April 12-14, 2016*, pages 1–4. IEEE, 2016.

FSW03. Pierre-Alain Fouque, Jacques Stern, and Jan-Geert Wackers. Cryptocomputing with rationals. In Matt Blaze, editor, *FC 2002*, volume 2357 of *LNCS*, pages 136–146. Springer, Heidelberg, March 2003.

GMW87. Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In Alfred Aho, editor, *19th ACM STOC*, pages 218–229. ACM Press, May 1987.

HIMV19. Carmit Hazay, Yuval Ishai, Antonio Marcedone, and Muthuramakrishnan Venkitasubramaniam. LevioSA: Lightweight secure arithmetic computation. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 327–344. ACM Press, November 2019.

HOSS18. Carmit Hazay, Emmanuela Orsini, Peter Scholl, and Eduardo Soria-Vazquez. Concretely efficient large-scale MPC with active security (or, TinyKeys for TinyOT). In Thomas Peyrin and Steven Galbraith, editors, *ASIACRYPT 2018, Part III*, volume 11274 of *LNCS*, pages 86–117. Springer, Heidelberg, December 2018.

HSS17. Carmit Hazay, Peter Scholl, and Eduardo Soria-Vazquez. Low cost constant round MPC combining BMR and oblivious transfer. In Tsuyoshi Takagi and Thomas Peyrin, editors, *ASIACRYPT 2017, Part I*, volume 10624 of *LNCS*, pages 598–628. Springer, Heidelberg, December 2017.

iee19. Ieee standard for floating-point arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, 2019.

IOZ14. Yuval Ishai, Rafail Ostrovsky, and Vassilis Zikas. Secure multi-party computation with identifiable abort. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part II*, volume 8617 of *LNCS*, pages 369–386. Springer, Heidelberg, August 2014.

JMR22. David Jacquemin, Ahmet Can Mert, and Sujoy Sinha Roy. Exploring RNS for isogeny-based cryptography. *IACR Cryptol. ePrint Arch.*, page 1289, 2022.

KLM05. Eike Kiltz, Gregor Leander, and John Malone-Lee. Secure computation of the mean and related statistics. In Joe Kilian, editor, *TCC 2005*, volume 3378 of *LNCS*, pages 283–302. Springer, Heidelberg, February 2005.

KLR16. Liisi Kerik, Peeter Laud, and Jaak Randmets. Optimizing MPC for robust and scalable integer and floating-point arithmetic. In Jeremy Clark, Sarah Meiklejohn, Peter Y. A. Ryan, Dan S. Wallach, Michael Brenner, and Kurt Rohloff, editors, *FC 2016 Workshops*, volume 9604 of *LNCS*, pages 271–287. Springer, Heidelberg, February 2016.

KOS16. Marcel Keller, Emmanuela Orsini, and Peter Scholl. MASCOT: Faster malicious arithmetic secure computation with oblivious transfer. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 830–842. ACM Press, October 2016.

KPR18. Marcel Keller, Valerio Pastro, and Dragos Rotaru. Overdrive: Making SPDZ great again. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part III*, volume 10822 of *LNCS*, pages 158–189. Springer, Heidelberg, April / May 2018.

LOS14. Enrique Larraia, Emmanuela Orsini, and Nigel P. Smart. Dishonest majority multi-party computation for binary circuits. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part II*, volume 8617 of *LNCS*, pages 495–512. Springer, Heidelberg, August 2014.

LP00. Yehuda Lindell and Benny Pinkas. Privacy preserving data mining. In Mihir Bellare, editor, *CRYPTO 2000*, volume 1880 of *LNCS*, pages 36–54. Springer, Heidelberg, August 2000.

LP12. Yehuda Lindell and Benny Pinkas. Secure two-party computation via cut-and-choose oblivious transfer. *Journal of Cryptology*, 25(4):680–722, October 2012.

LXZ$^+$20. Shaohua Li, Kaiping Xue, Bin Zhu, Chenkai Ding, Xindi Gao, David S. L. Wei, and Tao Wan. FALCON: A fourier transform based approach for fast and secure convolutional neural network predictions. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2020, Seattle, WA, USA, June 13-19, 2020*, pages 8702–8711. Computer Vision Foundation / IEEE, 2020.

MR18. Payman Mohassel and Peter Rindal. ABY^3: A mixed protocol framework for machine learning. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 35–52. ACM Press, October 2018.

MRVW21. Eleftheria Makri, Dragos Rotaru, Frederik Vercauteren, and Sameer Wagh. Rabbit: Efficient comparison for secure multi-party computation. In Nikita Borisov and Claudia Díaz, editors, *FC 2021, Part I*, volume 12674 of *LNCS*, pages 249–270. Springer, Heidelberg, March 2021.

MZ17. Payman Mohassel and Yupeng Zhang. SecureML: A system for scalable privacy-preserving machine learning. In *2017 IEEE Symposium on Security and Privacy*, pages 19–38. IEEE Computer Society Press, May 2017.

NO09. Jesper Buus Nielsen and Claudio Orlandi. LEGO for two-party secure computation. In Omer Reingold, editor, *TCC 2009*, volume 5444 of *LNCS*, pages 368–386. Springer, Heidelberg, March 2009.

Qui82. J.-J. Quisquater. Fast decipherment algorithm for rsa public-key cryptosystem. *Electronics Letters*, 18:905–907(2), October 1982.

RST$^+$22. Dragos Rotaru, Nigel P. Smart, Titouan Tanguy, Frederik Vercauteren, and Tim Wood. Actively secure setup for SPDZ. *Journal of Cryptology*, 35(1):5, January 2022.

RW19. Dragos Rotaru and Tim Wood. MArBled circuits: Mixing arithmetic and Boolean circuits with active security. In Feng Hao, Sushmita Ruj, and Sourav Sen Gupta, editors, *INDOCRYPT 2019*, volume 11898 of *LNCS*, pages 227–249. Springer, Heidelberg, December 2019.

SBIT22. Stella Simić, Alberto Bemporad, Omar Inverso, and Mirco Tribastone. Tight error analysis in fixed-point arithmetic. *Form. Asp. Comput.*, 34(1), sep 2022.

ST67. Nicholas S Szabo and Richard I Tanaka. *Residue arithmetic and its applications to computer technology / Nicholas S. Szabo, Richard I. Tanaka.* McGraw-Hill series in information processing and computers. McGraw-Hill, New York, 1967.

VSG$^+$19. Nikolaj Volgushev, Malte Schwarzkopf, Ben Getchell, Mayank Varia, Andrei Lapets, and Azer Bestavros. Conclave: secure multi-party computation on big data. In George Candea, Robbert van Renesse, and Christof Fetzer, editors, *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019*, pages 3:1–3:18. ACM, 2019.

WGC19. Sameer Wagh, Divya Gupta, and Nishanth Chandran. SecureNN: 3-party secure computation for neural network training. *PoPETs*, 2019(3):26–49, July 2019.

Yao82. Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *23rd FOCS*, pages 160–164. IEEE Computer Society Press, November 1982.

YSMN21. Sen Yuan, Milan Shen, Ilya Mironov, and Anderson C. A. Nascimento. Practical, label private deep learning training based on secure multiparty computation and differential privacy. Cryptology ePrint Archive, Report 2021/835, 2021. `https://eprint.iacr.org/2021/835`.

A Background on MPC and its Security Models

Many different flavors of MPC exist today concerning the security they offer. This is typically classified by several different metrics depending on such as whether the adversary corrupts parties *before* the start of the protocol (*static* security) or after (*adaptive* security), and the number of parties they corrupt (dishonest or honest majority). In this paper, we consider the setting of static corruption and a dishonest majority. Another essential metric of corruption is what powers the adversary have over the corrupt parties. Concerning the latter, the most common models are the *semi-honest* and *malicious* models. In the semi-honest model, corrupt parties are assumed to follow the protocol and are thus only allowed to break privacy based on analysis of the transcript. In the malicious model on the other hand, corrupt parties might deviate from the prescribed protocol, but except with negligible probability they cannot influence the result of the computation in any way, and they don't learn anything other than they would have, had they behaved honestly. The malicious model is very strong and is generally much more computationally expensive than the semi-honest model. Furthermore, multiple theoretical limitations of the malicious model exist; if at most $t < n/2$ parties are corrupt then it is possible to guarantee the completion of the computation with output to all parties in case of malicious corruption, under the assumption of a broadcast channel [GMW87]. This is known as *guaranteed output delivery.* A weaker version of this is *fairness* [CL17] where *all* parties are guaranteed to learn the output if a single party does. Hence, ensuring that a *rushing* adversary[11] cannot selectively abort if they don't like the result of the computation. Thus, for a malicious corruption threshold $t \geq n/2$ we can still have malicious security [NO09, BDOZ11, DPSZ12, LP12, LOS14, KOS16, HOSS18], but with the possibility of the protocol aborting in case of malicious behavior. This leaves two cases; do we *identify* a corrupt party when aborting, or will it be unknown to the honest parties who else is honest and who misbehaves? The identification of malicious parties becomes highly relevant in real-world executions, where we would like to

[11] A rushing adversary is allowed to wait for a reply from all honest parties before communicating.

boot, and perhaps publicly shame, a malicious server. It turns out that *identifiable abort* becomes cumbersome and computationally expensive [IOZ14, BOS16].

To achieve greater granularity of the security need versus the computational requirements Aumann and Lindell [AL10] formally introduced the model of *covert* security which captures the situation where parties can act maliciously, but they will get caught (identified) if they do so with a certain (non-negligible) probability. Concretely based on a public parameter γ, known as the *deterrence factor*, the adversary can succeed in cheating with probability at most $1-\gamma$. More specifically Aumann and Lindell define 3 different covert models: *Fail-safe* where the simulator is allowed to fail/be distinguishable in case the adversary succeeds in cheating. *Weak*; where the adversary receives the honest parties' input when cheating *regardless* of whether they get caught or not. *Strong*; where the adversary only learns the honest parties' inputs in case they cheat and *don't* get caught.

The features relating to abort and identifiability are closely connected to the *synchronicity* of the communication model. In the synchronous model, parties are expected to communicate at the same time and hence a party can abort by refusing to send a required message at the required points in the protocol. Thus, their identity is implicitly identified, through this inaction. We call this a *stop-abort*. Note that the synchronous model does not preclude a *rushing* adversary, as we can simply imagine that each party (synchronously) gets to decide whether they want to be the next one to provide input. Thus, the adversary can simply decide to skip each possible option they have to provide input until the malicious parties are the only ones that have not provided input. In practice, a stop-abort is discovered through a time-out on the network layer in the protocol. A harder case happens in the situation where an adversary behaves maliciously and sends a mathematically incorrect message, which will only cause the protocol to fail at a later time. We call this a *fault-abort*. In such situations, it can be impossible to identify which party sent the wrong message; at least without requiring all parties to send their transcript of the entire protocol, and hence break privacy. We note that our functionality $\mathcal{F}_{\mathsf{ABB}}$ in fig. 2.2 allows the adversary to identifiably stop-abort implicitly at any step by not querying the box with a message that makes it operate. Fault aborts are explicitly captured in the *output* phase. While some protocols may allow the discovery of fault-aborts earlier, they only really become relevant in the output phase as the functionality $\mathcal{F}_{\mathsf{ABB}}$ does not afford *fairness* (where *all* parties are guaranteed to learn the output if a single party does), and hence an abort only really becomes relevant when the adversary learns the output of the computation. The reason is that a rushing adversary could learn the output and choose to abort if they do not like it. In such situations, it becomes crucial to be able to identify the malicious party as the abort is now dependent on the honest parties' private input (through the computation).

B Helper proofs

Proof (of remark 1). Assume without loss of generality that the honest party is n. Observe that party n will always pick values $r_{1,n,c},\dots,r_{n,n,c}$ uniformly at random. Next see that the value r_k will be a sum of values modulo p, including $r_{n,n,c}$. Hence, the value is uniformly random distributed. This in turn, also means that each $r_{i,c}$ will be uniformly random distributed modulo p. Thus, we see that the distribution of $\bar{r}_c = \sum_{i\in[n]} r_{i,c}$ and hence ϵ can be described exactly, see for example [CR07]. However, this is quite cumbersome. Instead, we consider a continuous distribution as an approximation. Recall that the Irwin-Hall distribution is the probability distribution for a random variable; computed as the sum of independent random and uniform variables. This if we assume that the $r_{i,c}$'s are continuously, uniformly distributed, $r_{i,c} \sim p\cdot\mathbf{U}(0,1)$ for all $i\in[n]$, then $\bar{r}_c = \sum_{i\in[n]} r_{i,c} \sim p\cdot\mathbf{IrwinHall}(n)$. Now since $\epsilon = \lfloor \bar{r}_c/p \rfloor$ we have that ϵ is approximately Irwin-Hall distributed on n; $\epsilon \sim \mathbf{IrwinHall}(n)$. $\square$

Proof (of remark 2). First observe that ε' can be calculated based on ϵ and x in the following way:

$$\varepsilon' = \left\lfloor \frac{\lfloor \frac{(U+1)(x \bmod p)}{p} \rfloor + \epsilon}{U+1} \right\rfloor \tag{1}$$

which is argued in lemma 8, now recall from remark 1 that $\epsilon \sim \mathbf{IrwinHalls}(n)$. Next, recalling that the Irwin-Hall distribution converges to the normal distribution,we have $\epsilon \sim \mathbf{N}(U/2, U/12)$ for a sufficiently large number of parties. Similarly, consider an indicator variable b which is 1 if $x + r > p$ in algorithm 3, similarly to the variable in the proof of lemma 1. Observe that b can be approximated by a Bernoulli distribution with probability parameter $(x \bmod p)/p$. I.e. $b \sim \mathbf{Bernoulli}(x \bmod p)/p)$. Now see that the distribution of b affects ϵ in lemma 6 by at most 1. Thus, the error, ϵ, resulting from algorithm 1 when used in algorithm 3 is approximately distributed by $\mathbf{N}(U/2, U/12) + \mathbf{Bernoulli}((x \bmod p)/p)$. Thus, from equation 1 we get

$$\begin{aligned}
\varepsilon' &\sim \left\lfloor \frac{\lfloor \frac{(U+1)(x \bmod p)}{p} \rfloor + \mathbf{N}(U/2, U/12) + \mathbf{Bernoulli}((x \bmod p)/p)}{U+1} \right\rfloor \\
&\sim \left\lfloor \frac{(U+1)\mathbf{Bernoulli}((x \bmod p)/p) + \mathbf{N}(U/2, U/12) + \mathbf{Bernoulli}((x \bmod p)/p)}{U+1} \right\rfloor \\
&\sim \left\lfloor \frac{(U+2)\mathbf{Bernoulli}((x \bmod p)/p) + \mathbf{N}(U/2, U/12)}{U+1} \right\rfloor \\
&= \mathbf{Bernoulli}((x \bmod p)/p) + \mathbf{N}(U/2, U/12)
\end{aligned}$$

□

Proof (of remark 3). First, observe the following:

$$\begin{aligned}
x - py + pU &\leq pQ - Up - p\left(\left\lfloor \frac{pQ - Up}{p} \right\rfloor + \epsilon\right) + Up \\
&= pQ - Up - (pQ - Up + p\epsilon) + Up \\
&= p\epsilon + Up \leq Up
\end{aligned}$$

Thus we have that $x' \leq (U+1)Up$, so all there is left to show is that:

$$\begin{aligned}
(U+1)Up &\leq pQ - Up \\
(U+1)U + U &\leq Q \\
U^2 + 2U &\leq Q
\end{aligned}$$

Now since $Q > p$ and $p > 2^s$, we thus have for $U^2 + 2U \leq 2^s$ as by assumption.

The following lemma and its constructive proof show how to construct the polynomial P required by algorithm 6 and that the computation of this algorithm indeed reduces an error $\epsilon \leq n$ to $\varepsilon' \leq 1$. The security proof of algorithm 6 is found in the monolithic proof of Lemma 4 in Section 5.

Proposition 1. *Algorithm 6 computes $[\lfloor x/p \rfloor + \varepsilon']_m$ with $\varepsilon' \in \{0, 1\}$, under the assumption that $0 \leq x < pQ - Up$.*

Proof. This follows from lemma 8 using algorithm 5 as the function T with $L = U$ and $M = U + 1$.

Lemma 8. *Let $p > 1$ and assume that $T = T_{p,L} : \mathbb{N} \to \mathbb{N}$ computes the truncation by p with a bounded error, e.g.*

$$T(x) = \left\lfloor \frac{x}{p} \right\rfloor + \epsilon$$

where ϵ is some random number with $0 \leq \epsilon \leq L$ and $L < p$. If we let M be an integer with $M \geq 2$ and define

$$T'(x) = T(x) - L + P(T(M(x - pT(x) + pL)))$$

where $P = P_M$ is a polynomial such that $P(x) = \lfloor x/M \rfloor$ for $x = 0, \ldots, M + ML + L - 1$ (see remark 6), then

$$T'(x) = \left\lfloor \frac{x}{p} \right\rfloor + \varepsilon'$$

for a non-negative integer ε' where

$$0 \leq \varepsilon' \leq \begin{cases} 0 & \textit{if } M > pL, \\ 1 & \textit{if } L < M \leq pL, \\ 1 + \left\lfloor \frac{L}{M} \right\rfloor & \textit{if } M \leq L. \end{cases}$$

Proof. Write $x = kp + r$ with $k, r \in \mathbb{N}$ and $0 \leq r < p$. Then $T(x) = k + \epsilon$ where $0 \leq \epsilon \leq L$ so

$$x - pT(x) + pL = x - kp - p\epsilon + pL = r + p(L - \epsilon).$$

Now a second application of T gives us

$$T(M(r + p(L - \epsilon))) = \left\lfloor \frac{Mr + Mp(L - \epsilon)}{p} \right\rfloor + \tilde{\epsilon} = \left\lfloor \frac{Mr}{p} \right\rfloor + M(L - \epsilon) + \tilde{\epsilon}.$$

for some $\tilde{\epsilon}$ with $0 \leq \tilde{\epsilon} \leq L$. Now note that $\left\lfloor \frac{Mr}{p} \right\rfloor + M(L - \epsilon) + \tilde{\epsilon} < M + ML + L$ since $r < p$ so $Mr/p < M$ and hence $\lfloor Mr/p \rfloor < M$, $\epsilon \geq 0$ and $\tilde{\epsilon} \leq L$, so we may apply P and get

$$P\left(\left\lfloor \frac{Mr}{p} \right\rfloor + M(L - \epsilon) + \tilde{\epsilon} \right) = \left\lfloor \frac{\left\lfloor \frac{Mr}{p} \right\rfloor + M(L - \epsilon) + \tilde{\epsilon}}{M} \right\rfloor = L - \epsilon + \varepsilon'$$

where

$$\varepsilon' = \left\lfloor \frac{\left\lfloor \frac{Mr}{p} \right\rfloor + \tilde{\epsilon}}{M} \right\rfloor \leq \left\lfloor \frac{\left\lfloor \frac{M(p-1)}{p} \right\rfloor + \tilde{\epsilon}}{M} \right\rfloor = \left\lfloor \frac{M - \left\lfloor \frac{M}{p} \right\rfloor + \tilde{\epsilon}}{M} \right\rfloor.$$

Now, if $M \leq L$ then $\varepsilon' \leq 1 + \left\lfloor \frac{L}{M} \right\rfloor$ because $\tilde{\epsilon} \leq L$. If $M > L$ then ignoring the $\lfloor M/p \rfloor$ term, the numerator is strictly smaller than $2M$ so $\varepsilon' \leq 1$. If $M > pL$ then $\varepsilon' = 0$ because $\lfloor M/p \rfloor > L$ so the numerator is strictly smaller than M. Inserting this into the definition of T' gives

$$T'(x) = T(x) - L + P(T(M(x - pT(x) + pL))) = \left\lfloor \frac{x}{p} \right\rfloor + \epsilon - L + L - \epsilon + \varepsilon' = \left\lfloor \frac{x}{p} \right\rfloor + \varepsilon'$$

which finishes the proof. □

Remark 6. The polynomial P_M used in lemma 8 may be constructed using Lagrange interpolation,

$$P_M(x) = \sum_{j=0}^{M+ML+L-1} \left\lfloor \frac{j}{M} \right\rfloor \left(\prod_{\substack{i=0 \\ i \neq j}}^{M+ML+L-1} \frac{x - i}{j - i} \right).$$

Note that P_M has degree $M + ML + L - 2$ and may be computed in advance for given M and L.

In our case, L will be a lot smaller than p, so we will in practice choose $M = L + 1$ and accept the small error term because using $M > pL$ will make the degree of P_M very high, so the evaluation of this will become a bottleneck.