Improved Primitives for MPC over Mixed Arithmetic-Binary Circuits

Daniel Escudero[1], Satrajit Ghosh[1], Marcel Keller[2], Rahul Rachuri[1], Peter Scholl[1]

[1] Aarhus University, {escudero, satrajit, rachuri, peter.scholl}@cs.au.dk
[2] CSIRO's Data61, mks.keller@gmail.com

Abstract. This work introduces novel techniques to improve the translation between arithmetic and binary data types in secure multi-party computation. We introduce a new approach to performing these conversions using what we call *extended doubly-authenticated bits* (edaBits), which correspond to shared integers in the arithmetic domain whose bit decomposition is shared in the binary domain. These can be used to considerably increase the efficiency of non-linear operations such as truncation, secure comparison and bit-decomposition.
Our edaBits are similar to the *daBits* technique introduced by Rotaru et al. (Indocrypt 2019). However, we show that edaBits can be directly produced much more efficiently than daBits, with active security, while enabling the same benefits in higher-level applications. Our method for generating edaBits involves a novel cut-and-choose technique that may be of independent interest, and improves efficiency by exploiting natural, tamper-resilient properties of binary circuits that occur in our construction. We also show how edaBits can be applied to efficiently implement various non-linear protocols of interest, and we thoroughly analyze their correctness for both signed and unsigned integers.
The results of this work can be applied to any corruption threshold, although they seem best suited to dishonest majority protocols such as SPDZ. We implement and benchmark our constructions, and experimentally verify that our technique yields a substantial increase in efficiency. EdaBits save in communication by a factor that lies between 2 and 60 for secure comparisons with respect to a purely arithmetic approach, and between 2 and 25 with respect to using daBits. Improvements in throughput per second are slightly lower but still as high as a factor of 47. We also apply our novel machinery to the tasks of biometric matching and convolutional neural networks, obtaining a noticeable improvement as well.

1 Introduction

Secure multi-party computation, or MPC, allows a set of parties to compute some function f on private data, in such a way that the parties do not learn anything about the actual inputs to f, beyond what could be computed given the result. MPC can be used in a wide range of applications, such as private statistical analysis, machine learning, secure auctions and more.

MPC protocols can vary widely depending on the adversary model that is considered. For example, protocols in the *honest majority* setting are only secure as long as fewer than half of the parties are corrupt and colluding, whilst protocols secure against a *dishonest majority* allow all-but-one of the parties to be corrupt. Another important distinction is whether the adversary is assumed to be *semi-honest*, that is, they will always follow the instructions of the protocol, or *malicious*, and can deviate arbitrarily.

The mathematical structure underpinning secure computation usually requires to fix what we call a computation domain. The most common examples of such domains are computation modulo a large number (prime or power of two) or binary circuits (computation modulo two). In terms of cost, the former is more favorable to integer computation such as addition and multiplication while the latter is preferable for highly non-linear functions such as comparisons.

Applications often feature both linear and non-linear functionality. For example, convolution layers in deep learning consist of dot products followed by a non-linear activation function. It is therefore desirable to convert between an arithmetic computation domain and binary circuits. This has led to a line of works exploring this possibility, starting with the ABY framework [DSZ15] (Arithmetic-Boolean-Yao) in the two-party setting with semi-honest security. Other works have extended this to the setting of three parties with an honest majority [MR18, ABF$^+$18], dishonest majority with malicious security [RW19], as well as creating compilers that automatically decide which parts of a program should done in the binary or arithmetic domain [BDK$^+$18, IMZ19, CGR$^+$19].

A particular technique that is relevant for us is so-called *daBits* [RW19] (doubly-authenticated bits), which are random secret bits that are generated simultaneously in both the arithmetic and binary domains. These can be used for binary/arithmetic conversions in MPC protocols with any corruption setting, and have in particular been used with the SPDZ protocol [DPSZ12], which provides malicious security in the dishonest majority setting. Later works have given more efficient ways of generating daBits [AOR$^+$19, RST$^+$19, BST20], both with SPDZ and in the honest majority setting.

Another recent work uses function secret sharing [BGI15] for binary/arithmetic conversions and other operations such as comparison [BGI19]. This approach leads to a fast online phase with just one round of interaction and optimal communication complexity. However, it requires either a trusted setup, or an expensive preprocessing phase which has not been shown to be practical for malicious adversaries.

Limitations of daBits. Using daBits, it is relatively straightforward to convert between two computation domains. However, we found that in application-oriented settings the benefit of daBits alone is relatively limited. More concretely, if daBits are used to compute a comparison between two numbers that are secret-shared in $\mathbb{Z}_M$, for large arithmetic modulus M, the improvement is a factor of three at best. The reason for this is that the cost of creating the required daBits comes quite close to computing the comparison entirely in $\mathbb{Z}_M$. This limitation seems to be inherent with any approach based on daBits, since a daBit requires generating a random shared bit in $\mathbb{Z}_M$. The only known way of doing this with malicious security require first performing a multiplication (or squaring) in $\mathbb{Z}_M$ on a secret value [DFK+06, DEF+19]. However, secret multiplication is an expensive operation in MPC, and doing this for every daBit gets costly.

1.1 Our Contributions

In this paper, we present a new approach to converting between binary and arithmetic representations in MPC. Our method is general, and can be applied to a wide range of corruption settings, but seems particularly well-suited to the case of dishonest majority with malicious security such as SPDZ [DPSZ12, DKL+13], over the arithmetic domain $\mathbb{Z}_p$ for large prime p, or the ring $\mathbb{Z}_{2^k}$ [CDE+18]. Unlike previous works, we do not generate daBits, but instead create what we call *extended daBits* (edaBits), which avoid the limitations above. These allow conversions between arithmetic and binary domains, but can also be used directly for certain non-linear functions such as truncations and comparisons. We found that, for two- and three-party computation, edaBits allow to reduce the communication cost by up to two orders of magnitude and the wall clock time by up to a factor of 50 while both the inputs as well as the output are secret-shared in an arithmetic domain.

Below we highlight some more details of our contribution.

Extended daBits. An edaBit consists of a set of m random bits $(r_{m-1}, \dots, r_0)$, secret-shared in the binary domain, together with the value $r = \sum_{i=0}^{m-1} r_i 2^i$ shared in the arithmetic domain. We denote these sharings by $[r_{m-1}]_2, \dots, [r_0]_2$ and $[r]_M$, for arithmetic modulus M. Note that a daBit is simply an edaBit of length $m = 1$, and m daBits can be easily converted into an edaBit with a linear combination of the arithmetic shares. We show that this is wasteful, however, and edaBits can in general be produced much more efficiently than m daBits, for values of m used in practice.

Efficient malicious generation of edaBits. Let us first consider a simple approach with semi-honest security. If there are n parties, we have each party locally sample a value $r^i \in \mathbb{Z}_M$, then secret-shares r^i in the arithmetic domain, and the bits of r^i in the binary domain. We refer to these sharings as a *private* edaBit known to P_i. The parties can combine these by computing $\sum_i r^i$ in the arithmetic domain, and executing $n - 1$ protocols for addition in the binary domain, with a cost $O(nm)$ AND gates. Compared with using daBits, which costs $O(m)$ secret multiplications in $\mathbb{Z}_M$, this is much cheaper if n is not too large, by the simple fact that AND is a cheaper operation than multiplication in MPC.

To extend this naive approach to the malicious setting, we need a way to somehow verify that a set of edaBits was generated correctly. Firstly, we extend the underlying secret-sharing scheme to one that enforces correct computations on the underlying shares. This can be done, for instance, using authenticated secret-sharing with information-theoretic MACs as in SPDZ [DPSZ12]. Secondly, we use a cut-and-choose procedure to check that a large batch of edaBits are correct. This method is inspired by previous techniques for checking multiplication triples in MPC [BLN+15, FKOS15, FLNW17]. However, the case of edaBits is much more challenging to do efficiently, due to the highly non-linear relation between sharings in different domains, compared with the simple multiplicative property of triples (shares of (a, b, c) where $c = ab$).

Cut-and-choose approach. Our cut-and-choose procedure begins as in the semi-honest case, with each party P_i sampling and inputting a large batch of private edaBits of the form $(r^i_{m-1}, \ldots, r^i_0), r^i$. We then run a verification step on P_i's private edaBits, which begins by randomly picking a small subset of the edaBits to be opened and checked for correctness. Then, the remaining edaBits are shuffled and put into buckets of fixed size B. The first edaBit in each bucket is paired off with every other edaBit in the bucket, and we run a checking procedure on each of these pairs. To check a pair of edaBits r, s, the parties can compute $r + s$ in both the arithmetic and binary domains, and check these open to the same value. If all checks pass, then the parties take the first private edaBit from every bucket, and add this to all the other parties' private edaBits, created in the same way, to obtain secret-shared edaBits. Note that to pass a single check, the adversary must have corrupted both r and s so that they cancel each other out; therefore, the only way to successfully cheat is if every bucket with a corrupted edaBit contains *only* corrupted edaBits. By carefully choosing parameters, we can ensure that it is very unlikely the adversary manages to do this. For example, with 40-bit statistical security, from the analysis of [FLNW17], we could use bucket size $B = 3$ when generating more than a million sets of edaBits.

While the above method works, it incurs considerable overhead compared with similar cut-and-choose techniques used for multiplication triples. This is because in every pairwise check within a bucket, the parties have to perform an addition of binary-shared values, which requires a circuit with $O(m)$ AND gates. Each of these AND gates consumes an authenticated multiplication triple over $\mathbb{Z}_2$, and generating these triples themselves requires additional layers of cut-and-choose and verification machinery, when using efficient protocols based on oblivious transfer [NNOB12, FKOS15, WRK17b].

To reduce this cost, our first optimization is as follows. Recall that the check procedure within each bucket is done on a pair of *private* values known to one party, and not secret-shares. This means that when evaluating the addition circuit, it suffices to use *private* multiplication triples, which are authenticated triples where the secret values are known to party P_i. These are much cheaper to generate than fully-fledged secret-shared triples, although still require a verification procedure based on cut-and-choose. To further reduce costs, we propose a second, more significant optimization.

Cut-and-choose with faulty check circuits. Instead of using private triples that have been checked separately, we propose to use *faulty private triples*, that is, authenticated triples that are not guaranteed to be correct. This immediately raises the question, how can the checking procedure be useful, if the verification mechanism itself is faulty? The hope is that if we randomly shuffle the set of triples, it may still be hard for an adversary who corrupts them to ensure that any incorrect edaBits are canceled out in the right way by the faulty check circuit, whilst any correct edaBits still pass unscathed. Proving this, however, is challenging. In fact, it seems to inherently rely on the *structure* of the binary circuit that computes the check function. For instance, if a faulty circuit can cause a check between a good and a bad edaBit to pass, and the same circuit also causes a check between two good edaBits to pass, for some carefully chosen inputs, then this type of cheating can help the adversary.

To rule this out, we consider circuits with a property we call *weak additive tamper-resilience*, meaning that for any tampering that flips some subset of AND gate outputs, the tampered circuit is either incorrect for every possible input, or it is correct for all inputs. This notion essentially rules out input-dependent failures from faulty multiplication triples, which avoids the above attack and allows us to simplify the analysis.

Weak additive tamper-resilience is implied by previous notions of circuits secure against additive attacks [GIP+14], however, these constructions are not practical over $\mathbb{F}_2$. Fortunately, we show that the standard ripple-carry adder circuit satisfies our notion, and suffices for creating edaBits in $\mathbb{Z}_{2^k}$. However, the circuit for binary addition modulo a prime, which requires an extra conditional subtraction, does not satisfy this. Instead, we adapt the circuit over the integers to use in our protocol modulo p, which allows us to generate length-m edaBits for any $m < \log p$; this turns out to be sufficient for most applications.

With this property, we can show that introducing faulty triples does not help an adversary to pass the check, so we can choose the same cut-and-choose parameters as previous works on triple generation, while saving significantly in the cost of generating our triples used in verification. The bulk of our technical contribution is in analysing this cut-and-choose technique.

Silent OT-friendly. Another benefit of our approach is that we can take advantage of recent advances in oblivious transfer (OT) extension techniques, which allow to create a large number of random, or correlated, OTs, with very little interaction [BCG+19b]. In practice, the communication cost when using this "silent OT" method can be more than 100x less than OT extension based on previous techniques [IKNP03], with a modest increase in computation [BCG+19a]. In settings where bandwidth is expensive, this suits our protocol well, since we mainly use MPC operations in $\mathbb{F}_2$ to create edaBits, and these are best done with OT-based techniques. This reduces the communication of our edaBits protocol by an $O(\lambda)$ factor, in practice cutting communication by 50–100x, although we have not yet implemented this optimization.

Note that it does not seem possible to exploit silent OT with previous daBit generation methods such as by Aly et al. [AOR+19]. This is due to the limitation mentioned previously that these require a large number of random bits shared in $\mathbb{Z}_p$, which we do not know how to create efficiently using OT.

Applications: improved conversions and primitives. edaBits can be used in a natural way to convert between binary and arithmetic domains, where each conversion of an m-bit value uses one edaBit of length m, and a single m-bit addition circuit. (In the mod-p case, we also need one "classic" daBit per conversion, to handle a carry computation.) However, for many primitives such as secure comparison, equality test and truncation, a better approach is to exploit the edaBits to perform the operation without doing an explicit conversion. In the $\mathbb{Z}_{2^k}$ case, a similar approach was used previously when combining the SPDZ2k protocol with daBit-style conversions [DEF+19]. We adapt these techniques to work with edaBits, in both $\mathbb{Z}_{2^k}$ and $\mathbb{Z}_p$. As an additional contribution, more at the engineering level, we take great care in all our constructions to ensure they work for both signed and unsigned data types. This was not done by previous truncation protocols in $\mathbb{Z}_{2^k}$ based on SPDZ [DEF+19, DEK19], which only perform a *logical shift*, as opposed to the *arithmetic shift* that is needed to ensure correctness on signed inputs.

Handling garbled circuits. Our conversion method can also be extended to convert binary shares to garbled circuits, putting the 'Y' into 'ABY' and allowing constant round binary computations. In this paper, we do not focus on this, since the technique is exactly the same as described in [AOR+19]; when using binary shares based on TinyOT MACs, conversions between binary and garbled circuit representation comes for free, based on the observation from Hazay et al. [HSS17] that TinyOT sharings can be locally converted into shares of a multi-party garbled circuit.

Performance evaluation. We have implemented our protocol in all relevant security models and computation domains as provided by MP-SPDZ [Kel20], and we found it reduces communication both in microbenchmarks and application benchmarks when comparing to a purely arithmetic or a daBit-based implementation. More concretely, for secure comparisons the reduction in communication lies between a factor of 2 and 60 going from purely arithmetic to edaBits, and between 2 and 25 from daBits to edaBits. Improvements in throughput per second are slightly lower but still as high as a factor of 47. Generally, the improvements are higher for dishonest-majority computation and semi-honest security when considering black-box approaches such as purely arithmetic computation or using daBits. However, semi-honest computation allows for non-black-box approaches [MR18, DSZ15] that are as fast as ours.

We have also compared our implementation with the most established software for mixed circuits [BDK+18] and found that it still improves up to a factor of two for a basic benchmark in semi-honest two-party computation. However, they maintain an advantage if the parties are far apart (100 ms RTT) due to the usage of garbled circuits.

Finally, a comparison with a purely arithmetic implementation of deep-learning inference shows an improvement of up to a factor six in terms of both communication and wall clock time.

1.2 Paper Outline

We begin in Section 2 with some preliminaries. In Section 3, we introduce edaBits and show how to instantiate them, given a source of private edaBits. We then present our protocol for creating private edaBits in Section 4, based on the new cut-and-choose procedure. Then, in Sections 4.2–4.4 we describe abstract cut-and-choose games that model the protocol, and carry out a formal analysis. Then in Section 5 we show how to use edaBits for higher-level primitives like comparison and truncation. Finally, in Section 6, we analyze the efficiency of our constructions and present performance numbers from our implementation.

2 Preliminaries

In this work we consider three main algebraic structures: $\mathbb{Z}_M$ for $M = p$ where p is a large prime, $M = 2^k$ where k is a large integer, and $\mathbb{Z}_2$.

2.1 Arithmetic Black-Box

We model MPC via the arithmetic black box model (ABB), which is an ideal functionality in the universal composability framework [Can01]. This functionality allows a set of n parties $P_1, \ldots, P_n$ to input values, operate on them, and receive outputs after the operations have been performed. Typically (see for example Rotaru and Wood [RW19]), this functionality is parameterized by a positive integer M, and the values that can be processed by the functionality are in $\mathbb{Z}_M$, with the native operations being addition and multiplication modulo M.

In this work, we build on the basic ABB to construct edaBits, which are used in our higher-level applications. We therefore consider an extended version of the arithmetic black box model that handles values in both binary and arithmetic domains. First, within one single instance of the functionality we can have both binary and arithmetic computations, where the latter can be either modulo p or modulo 2^k. Furthermore, the functionality allows the parties to convert a single binary share into an arithmetic share of the same bit (but not the other way round). We will use this limited conversion capability to bootstrap to our fully-fledged edaBits, which can convert larger ring elements in both directions, and with much greater efficiency. The details of the functionality are presented in Fig. 1.

Notation. As shorthand, we write $[x]_2$ to refer to a secret bit x that has been stored by the functionality $\mathcal{F}_{\mathsf{ABB}}$, and similarly $[x]_M$ for a value $x \in \mathbb{Z}_M$ with $M \in \{p, 2^k\}$. We overload the operators $+$ and $\cdot$, writing for instance, $[y]_M = [x]_M \cdot [y]_M + c$ to denote that the secret values x and y are first multiplied using the Mult command, and then the public constant c is added using LinComb.

Functionality $\mathcal{F}_{\mathsf{ABB}}$

Input: On input $(\mathsf{Input}, P_i, \mathsf{type}, \mathsf{id}, x)$ from P_i and $(\mathsf{Input}, P_i, \mathsf{type}, \mathsf{id})$ from all other parties, with id a fresh identifier, $\mathsf{type} \in \{\mathsf{binary}, \mathsf{arithmetic}\}$ and $x \in \mathbb{Z}_2$ or $x \in \mathbb{Z}_M$ (depending on type), store $(\mathsf{type}, \mathsf{id}, x)$.

Linear Combination: On input $(\mathsf{LinComb}, \mathsf{type}, \mathsf{id}, (\mathsf{id}_j)_{j=1}^m, \mathsf{type}, c, (c_j)_{j=1}^m)$, where each id_j is stored in memory and $c, c_j \in \mathbb{Z}_2$ if $\mathsf{type} = \mathsf{binary}$ or $c, c_j \in \mathbb{Z}_M$ if $\mathsf{type} = \mathsf{arithmetic}$, retrieve $((\mathsf{type}, \mathsf{id}_1, x_1), \ldots, (\mathsf{type}, \mathsf{id}_m, x_m))$, compute $y = c + \sum_j x_j \cdot c_j$ modulo 2 if $\mathsf{type} = \mathsf{binary}$ and modulo M if $\mathsf{type} = \mathsf{arithmetic}$, and store $(\mathsf{type}, \mathsf{id}, y)$.

Multiply: On input $(\mathsf{Mult}, \mathsf{type}, \mathsf{id}, \mathsf{id}_1, \mathsf{id}_2)$ from all parties (where $\mathsf{id}_1, \mathsf{id}_2$ are present in memory), retrieve $(\mathsf{type}, \mathsf{id}_1, x)$, $(\mathsf{type}, \mathsf{id}_2, y)$, compute $z = x \cdot y$ modulo 2 if $\mathsf{type} = \mathsf{binary}$ and modulo M if $\mathsf{type} = \mathsf{arithmetic}$, and store (id, z).

From Binary to Arithmetic: On input $(\mathsf{ConvertB2A}, \mathsf{id}, \mathsf{id}')$ from all parties, retrieve $(\mathsf{binary}, \mathsf{id}', x)$ and store $(\mathsf{arithmetic}, \mathsf{id}, x)$.

Output: On input $(\mathsf{Output}, \mathsf{type}, \mathsf{id})$ from all honest parties (where id is present in memory), retrieve $(\mathsf{type}, \mathsf{id}, y)$ and output it to the adversary. Wait for an input from the adversary; if this is $\mathsf{Deliver}$ then output y to all parties, otherwise output Abort.

Fig. 1. Ideal functionality for the MPC arithmetic black box modulo 2 and modulo M, where M is either 2^k or p.

Instantiations. There are several ways to instantiate the basic commands of this functionality, depending on the adversarial setting. In the honest majority setting one can use for example Shamir secret-sharing or replicated-secret sharing [DN07, BLW08], which can be either passively or actively secure [FLNW17]. In the dishonest majority setting, additive secret-sharing is typically used. For the case of active security, we can combine this with information-theoretic MACs to enforce correct opening of shared values [DPSZ12, DKL+13, CDE+18, WRK17b]. Furthermore, the conversions between the arithmetic bits and binary sharings can be implemented via daBits, as shown in [AOR+19, RW19, RST+19]. We present a short summary of this daBit generation method in Section A in the appendix.

Since all of these are linear secret-sharing schemes, when secret values inside $\mathcal{F}_{\mathsf{ABB}}$ represent sharings under such a scheme, the $\mathsf{LinComb}$ command of $\mathcal{F}_{\mathsf{ABB}}$ can be implemented by simply computing the same linear combination on the shares. The Mult command is usually realized by preprocessing multiplication triples, that is, shared values $[a]_M, [b]_M, [c]_M$ where a, b are uniformly random in $\mathbb{Z}_M$ and $c = a \cdot b$. Given such a triple, two secret values $[x]_M, [y]_M$ can be multiplied by first opening $x + a$ and $y + b$, and then computing

$$[z]_M = (x + a)(y + b) - (x + a)[b]_M - (y + b)[a]_M + [c]_M$$

which can be computed as a linear operation in the secret values, producing $z = xy$.

We remark that preprocessing triples is often the most expensive part of the entire MPC protocol, especially in the dishonest majority setting. In the arithmetic case, these can be produced using linearly or somewhat homomorphic encryption [DPSZ12, KPR18], oblivious linear function evaluation [DGN$^+$17] or, with a higher communication cost, oblivious transfer [KOS16, CDE$^+$18]. In the binary case with $M = 2$, techniques based on oblivious transfer are usually fastest, and these are known as the TinyOT family of protocols [NNOB12, FKOS15, WRK17a, WRK17b].

3 Extended daBits

The main primitive of our work is the concept of extended daBits, or *edaBits*. Unlike a daBit, which is a random bit b shared as $([b]_M, [b]_2)$, an edaBit is a collection of bits $(r_{m-1}, \dots, r_0)$ such that (1) each bit is secret-shared as $[r_i]_2$ and (2) the integer $r = \sum_{i=0}^{m} r_i 2^i$ is secret-shared as $[r]_M$.

One edaBit of length m can be generated from m daBits, and in fact, this is typically the first step when applying daBits to several non-linear primitives like truncation. Instead of following this approach, we choose to generate the $\mathsf{edaBits}$—which is what is needed for most applications where daBits are used—directly, which leads to a much more efficient method and ultimately leads to more efficient primitives for MPC protocols.

At a high level, our protocol for generating $\mathsf{edaBits}$ proceeds as follows. Let us think initially of the passively secure setting. Each party P_i samples m random bits $r^i_{i,0}, \dots, r^i_{i,m-1}$, and secret-shares these bits towards the parties over $\mathbb{Z}_2$, as well as the integer $r_i = \sum_{j=0}^{m-1} r_{i,j} 2^j$ over $\mathbb{Z}_M$. Since each edaBit is known by one party, these $\mathsf{edaBits}$ must be combined to get $\mathsf{edaBits}$ where no party knows the underlying values. We refer to the former as *private* $\mathsf{edaBits}$, and to the latter as *global* $\mathsf{edaBits}$. The parties combine the private $\mathsf{edaBits}$ by adding them together: the arithmetic shares can be simply added locally as $[r]_M = \sum_{i=1}^{n} [r_i]_M$, and the binary shares can be added via an n-input binary adder. Some complications arise, coming from the fact that the r_i values may overflow mod p. Dealing with this is highly non-trivial, and we will discuss this in detail in the description of our protocol in Section 3.2. However, before we dive into our construction, we will first present the functionality we aim at instantiating. This functionality is presented in Fig. 2.

3.1 Functionality for Private Extended daBits

We also use a functionality $\mathcal{F}_{\mathsf{edaBitsPriv}}$, which models a *private* set of $\mathsf{edaBits}$ that is known to one party. This functionality is defined exactly as $\mathcal{F}_{\mathsf{edaBits}}$, except that the bits $r_0, \dots, r_{m-1}$ are given as output to one party; additionally, if that party is corrupt, the adversary may instead choose these bits.

The heaviest part of our contribution lies on the instantiation of this functionality, which we postpone to Section 4.

Functionality $\mathcal{F}_{\mathsf{edaBits}}$

The functionality is parametrized by $M \in \{2^k, p\}$ and $m \leq \log M$. It has the same features as $\mathcal{F}_{\mathsf{ABB}}$, together with the following command:

Create $\mathsf{edaBits}$**:** On input $(\mathsf{edabit}, \mathsf{id}_M, \mathsf{id}_2)$ from all parties, sample $(r_0, \ldots, r_{m-1}) \in \mathbb{Z}_2^m$ uniformly at random and store $(\mathsf{binary}, \mathsf{id}_2, r_j)$ for $j = 0, \ldots, m-1$, together with $(\mathsf{arithmetic}, \mathsf{id}_M, r)$, where $r = \sum_{j=0}^{m-1} r_j 2^j$.

Fig. 2. Ideal functionality for extended daBits.

3.2 From Private to Global Extended daBits

As we discussed already at the beginning of this section, one can instantiate $\mathcal{F}_{\mathsf{edaBits}}$ using $\mathcal{F}_{\mathsf{edaBitsPriv}}$, by combining the different private $\mathsf{edaBits}$ to ensure no individual party knows the underlying values. Small variations are required depending on whether $M = 2^k$ or $M = p$, for reasons that will become clear in a moment.

Now, to provide an intuition on our protocol, assume that the ABB is storing $([r_i]_M, [r_{i,0}]_2, \ldots, [r_{i,m-1}]_2)$ for $i = 1, \ldots, n$, where party P_i knows $(r_{i,0}, \ldots, r_{i,m-1})$ and $r_i = \sum_{j=1}^{m-1} r_{i,j} 2^j$. The parties can add their arithmetic shares to get shares of $r' = \sum_{i=1}^{n} r_i \mod M$, and they can also add their binary shares using a binary n-input adder, which results in shares of the bits of r', only without modular reduction.

Since we want to output a random m-bit integer, the parties need to remove the bits of r' beyond the m-th bit from the arithmetic shares. We have binary shares of these carry bits as part of the output from the binary adder, so using $\log(n)$ calls to $\mathsf{ConvertB2A}$ of $\mathcal{F}_{\mathsf{ABB}}$, each of which costs a (regular) daBit, we can convert these to the arithmetic world and perform the correction. Notice that for the case of $M = 2^k$, $m = k$, we can omit this conversion since the arithmetic shares are already reduced.

Even without the correction above, the least significant m bits of r' still correspond to $r_0, \ldots, r_{m-1}$. This turns out to be enough for some applications because it is easy to "delete" the most significant bit in $\mathbb{Z}_{2^k}$ by multiplying with two. We call such an edaBit loose as apposed to a strict one as defined in Fig. 2.

One must be careful with potential overflows modulo M. If $M = 2^k$, then any overflow bits beyond the k-th position can simply be discarded. On the other hand, if $M = p$, as long as $m < \log p$ then we can still subtract the $\log n$ converted carries from the arithmetic shares to correct for any overflow modulo p. The protocol is given in Fig. 3, and the security stated in Theorem 1 below, whose proof follows in a straightforward manner from the correctness of the additions in the protocol. In the protocol, $\mathsf{nBitADD}$ denotes an n-input binary adder on m-bit inputs. This can be implemented naively in a circuit with $< (m + \log n) \cdot (n - 1)$ AND gates.

Protocol Π_{edaBits}

Pre:

- Access to $\mathcal{F}_{\mathsf{edaBitsPriv}}$.
- If $M = p$, then $0 < m < \log(p)$.

Post: The parties get $([r]_M, [r_i]_2, \ldots, [r_i]_2)$ where $r = \sum_{j=1}^{m-1} r_i 2^j$ and the bits are uniform to the adversary.

1. The parties call the functionality $\mathcal{F}_{\mathsf{edaBitsPriv}}$ to get random shares $([r_i]_M, [r_{i,0}]_2, \ldots, [r_{i,m-1}]_2)$, for $i = 1, \ldots, n$. Party P_i additionally learns $r_{i,j}$ and $r_i = \sum_{j=1}^{m-1} r_{i,j} 2^j$.
2. The parties invoke $\mathcal{F}_{\mathsf{ABB}}$ to compute $[r']_M = \sum_{i=1}^{n} [r_i]_M$.
3. The parties invoke $\mathcal{F}_{\mathsf{ABB}}$ to compute $\mathsf{nBitADD}\,(([r_{1,j}]_2)_j, \ldots, ([r_{n,j}]_2)_j)$, obtaining $m + \log n$ bits $([b_0]_2, \ldots, [b_{m+\log(n)-1}]_2)$.
4. Call $\mathsf{ConvertB2A}$ from $\mathcal{F}_{\mathsf{ABB}}$ to convert $[b_j]_2 \mapsto [b_j]_M$ for $j = m, \ldots, m + \log(n) - 1$. If $M = 2^k$, values b_j for $j > k$ do not need to be converted, and for the sake of notation, we denote $[b_j]_{2^k} := 0$ for $j > k$.
5. Use $\mathcal{F}_{\mathsf{ABB}}$ to compute $[r]_M = [r']_M - 2^m \sum_{j=0}^{\log(n)-1} [b_{j+m}]_M 2^j$.
6. Output $([r]_M, [b_0]_2, \ldots, [b_{m-1}]_2)$.

Fig. 3. Protocol for generating global $\mathsf{edaBits}$ from private $\mathsf{edaBits}$.

Theorem 1. *Protocol Π_{edaBits} UC-realizes functionality $\mathcal{F}_{\mathsf{edaBits}}$ in the $(\mathcal{F}_{\mathsf{edaBitsPriv}}, \mathcal{F}_{\mathsf{B2A}})$-hybrid model.*

4 Instantiating Private Extended daBits

Our protocol for producing private edaBits is fairly intuitive. The protocol begins with each party inputting a set of edaBits to the ABB functionality. However, since a corrupt party may input inconsistent edaBits (that is, the binary part may not correspond to the bit representation of the arithmetic part), some extra checks must be set in place to ensure correctness. To this end, the parties engage in a consistency check, where each party must prove that their private edaBits were created correctly. We do this with a cut-and-choose procedure, where first a random subset of a certain size of edaBits is opened, their correctness is checked, and then the remaining edaBits are randomly placed into buckets. Within each bucket, all edaBits but the first one are checked against the first edaBit by adding the two in both the binary and arithmetic domains, and opening the result. With high probability, the first edaBit will be correct if all the checks pass.

This method is based on a standard cut-and-choose technique for verifying multiplication triples, used in several other works [FKOS15, FLNW17]. However, the main difference in our case is that the checking procedure for verifying two edaBits within a bucket is much more expensive: checking two multiplication triples consists of a simple linear combination and openeing, whereas to check

$\mathsf{edaBits}$, we need to run a binary addition circuit on secret-shared values. This binary addition itself requires $O(m)$ multiplication triples to verify, and the protocol for producing these triples typically requires further cut-and-choose steps to ensure correctness and security.

In this work, we take a different approach to reduce this overhead. First, we allow some of the triples used to perform the check within each bucket to be incorrect, which saves in resources as a triple verification step can be omitted. Furthermore, we observe that these multiplication triples are intended to be used on inputs that are known to the party proposing the $\mathsf{edaBits}$, and thus it is acceptable if this party knows the bits of the underlying triples as well. As a result, we can simplify the triple generation by letting this party sample the triples together with the $\mathsf{edaBits}$, which is much cheaper than letting the parties jointly sample (even incorrect) triples. Note that even though the triples may be incorrect, they must still be authenticated (in practice, with MACs) by the party who proposes them so that the errors cannot be changed after generating the triples.

To model this, we extend the arithmetic black box model with the following commands, for generating a private triple, and for faulty multiplication, which uses a previously stored triple to do a multiplication.

Input Triple. On input $(\mathsf{Triple}, \mathsf{id}, a, b, c)$ from P_i, where id is a fresh binary identifier and $a, b, c \in \{0, 1\}$, store $(\mathsf{Triple}, i, \mathsf{id}, a, b, c)$.

Faulty Multiplication. On input $(\mathsf{FaultyMult}, \mathsf{id}, \mathsf{id}_1, \mathsf{id}_2, \mathsf{id}_T, i)$ from all parties (where $\mathsf{id}_1, \mathsf{id}_2$ are present in memory), retrieve $(\mathsf{binary}, \mathsf{id}_1, x)$, $(\mathsf{binary}, \mathsf{id}_2, y)$, $(\mathsf{Triple}, i, \mathsf{id}_T, a, b, c)$, compute $z = x \cdot y \oplus (c \oplus a \cdot b)$, and store (id, z).

The triple command can be directly instantiated using Input from $\mathcal{F}_{\mathsf{ABB}}$, while $\mathsf{FaultyMult}$ uses Beaver's multiplication technique with one of these triples. Note that in Beaver-based binary multiplication, it is easy to see that any additive error in a triple leads to exactly the same error in the product.

Now we are ready to present our protocol to preprocess private $\mathsf{edaBits}$, described in Fig. 4. The party P_i locally samples a batch of $\mathsf{edaBits}$ and multiplication triples, then inputs these into $\mathcal{F}_{\mathsf{ABB}}$. The parties then run the $\mathsf{CutNChoose}$ subprotocol, given in Fig. 5, to check that the $\mathsf{edaBits}$ provided by P_i are consistent. The protocol outputs a batch of N $\mathsf{edaBits}$, and is parametrized by a bucket size B, and values C, C' which determine how many $\mathsf{edaBits}$ and triples are opened, respectively. $\mathsf{BitADDCarry}$ denotes a two-input binary addition circuit with a carry bit, which must satisfy the weakly additively tamper resilient property given in the next section. As we will see later, this can be computed with m AND gates and depth $m - 1$.[3]

The cut-and-choose protocol starts by using a standard coin-tossing functionality, $\mathcal{F}_{\mathsf{Rand}}$, to sample public random permutations used to shuffle the sets

[3] This circuit is rather naive, and in fact there are logarithmic depth circuits with a greater number of AND gates. However, as we will see later in the section, it is important for our security proof to use specifically these naive circuits to obtain the tamper-resilient property. Furthermore, they are only used in the preprocessing phase, so the overhead in round complexity is insignificant in practice.

Protocol $\Pi_{\mathsf{edaBitsPriv}}$

Pre: $\mathcal{F}_{\mathsf{ABB}}$ with modulus M, length parameter $m \in \mathbb{Z}$ with $m \leq \log_2 M$
Post: Batch of N shared edaBits $\{([r_j]_M, [r_{j,0}]_2, \ldots, [r_{j,m-1}]_2)\}_{j=1}^{N}$, where party P_i knows the underlying bits.

1. P_i samples $r_{j,0}, \ldots, r_{j,m-1} \in \mathbb{Z}_2$, for $j = 1, \ldots, NB + C$, and inputs these to $\mathcal{F}_{\mathsf{ABB}}$ in $\mathbb{Z}_2$.
2. P_i computes $r_j = \sum_{i=0}^{m-1} r_{j,i} 2^i$ and inputs $r_j \in \mathbb{Z}_M$ to $\mathcal{F}_{\mathsf{ABB}}$.
3. P_i samples $(N(B-1)+C')m$ random bit triples and inputs these to $\mathcal{F}_{\mathsf{ABB}}$.
4. The parties run the CutNChoose procedure to check the consistency of these edaBits. If the check passes, then the parties obtain N edaBits. Otherwise, they abort.

Fig. 4. Protocol for producing private extended daBits.

of edaBits and triples. The coin-tossing can be implemented, for example, with hash-based commitments in the random oracle model. Then the first C edaBits and $C'm$ triples are opened and tested for correctness; this is to ensure that not too large a fraction of the remaining edaBits and triples are incorrect. Then the edaBits are divided into buckets of size B, together with $B-1$ sets of m triples in each bucket. Then, the top edaBit from each bucket is checked with every other edaBit in the bucket by evaluating a binary addition circuit using the triples, and comparing the result with the same addition done in the arithmetic domain. Each individual check in the CutNChoose procedure takes two edaBits of m bits each, and consumes m triples as well as a single regular daBit, needed to convert the carry bit from the addition into the arithmetic domain. Note that when working with modulus $M = 2^k$, if $m = k$ then this conversion step is not needed.

4.1 Weakly Tamper-Resilient Binary Addition Circuit

To implement the BitADDCarry circuit we use a ripple-carry adder, which computes the carry bit at every position with the following equation:

$$c_{i+1} = c_i \oplus ((x_i \oplus c_i) \wedge (y_i \oplus c_i)), \forall i \in \{0, m-1\} \tag{1}$$

where $c_0 = 0$, and x_i, y_i are the i-th bits of the two binary inputs. It then outputs $z_i = x_i \oplus y_i \oplus c_i$, for $i = 0, \ldots, m-1$, and the last carry bit c_m. Note that this requires m AND gates and has linear depth.

Below we define the tamper-resilient property of the circuit that we require. We consider an adversary who can additively tamper with a binary circuit by inducing bit-flips in the output wires of any AND gate.

Definition 1. *A binary circuit $\mathcal{C} : \mathbb{F}_2^{2m} \to \mathbb{F}_2^{m+1}$ is* weakly additively tamper resilient*, if given any tampered circuit $\mathcal{C}^*$, obtained by additively tampering $\mathcal{C}$, one of the following holds:*

Procedure CutNChoose

Pre: A batch of $(NB+C)$ shared edaBits $\{([r]_M, [r_0]_2, \dots, [r_{m-1}]_2)\}_{j=1}^{NB+C}$ and a batch of $(N \cdot (B-1) \cdot m + C' \cdot m)$ triples, all stored in $\mathcal{F}_{\mathsf{ABB}}$, where party P_i knows the underlying bits of the edaBits and the triples.
Post: N verified edaBits
The parties do the following:

1. Using $\mathcal{F}_{\mathsf{Rand}}$, sample two public random permutations and use these to shuffle the edaBits and the triples.
2. Open the first C of the shuffled edaBits in both worlds, and the first $C' \cdot m$ triples. Abort if any of the edaBits or the triples are inconsistent.
3. Place the remaining edaBits into buckets of size B and the triples into buckets of size $(B-1) \cdot m$.
4. For each bucket, select the first edaBit $([r]_M, [r_0]_2, \dots, [r_{m-1}]_2)$, and for every other edaBit $([s]_M, [s_0]_2, \dots, [s_{m-1}]_2)$ in the same bucket, perform the following check:
   (a) Let $[r+s]_M = [r]_M + [s]_M$.
   (b) Let $([c_0]_2, \dots, [c_m]_2) = \mathsf{BitADDCarry}([r_0]_2, \dots, [r_{m-1}]_2, [s_0]_2, \dots, [s_{m-1}]_2)$, using the FaultyMult command to evaluate each AND gate.
   (c) Convert $[c_m]_2 \mapsto [c_m]_M$ with ConvertB2A.
   (d) Let $[c']_M = [r+s]_M - 2^m \cdot [c_m]_M$. Open c' and the corresponding bits $c_0, \dots, c_{m-1}$ from the binary world, and check that $c' = \sum_{i=0}^{m-1} c_i 2^i$.
5. If all the checks pass, output the first edaBit from each of the N buckets.

Fig. 5. Cut-and-choose procedure to check correctness of input edaBits.

1. $\forall (x,y) \in \mathbb{F}_2^m : \ \mathcal{C}(x,y) = \mathcal{C}^*(x,y)$.
2. $\forall (x,y) \in \mathbb{F}_2^m : \ \mathcal{C}(x,y) \neq \mathcal{C}^*(x,y)$.

Intuitively, this says that the tampered circuit is either incorrect on every possible input, or functionally equivalent to the original circuit. In our protocol, this property restricts the adversary from being able to pass the check with a tampered circuit with bad edaBits as well as the same circuit with good edaBits. It ensures that if any multiplication triple is incorrect, then the check at that position would only pass with either a good edaBit, or a bad edaBit (but not both).

We now show that this property is satisfied by the ripple-carry adder circuit above, which we use.

Lemma 1. *The ripple carry adder circuit above is weakly additively tamper-resilient (Definition 1).*

Proof. Consider a tampered circuit $\mathcal{C}^*$, and let i be the smallest index where the AND gate in equation 1 has been tampered. Since c_i was computed correctly, we have $\mathcal{C}^*(x,y)[i+1] = \mathcal{C}(x,y)[i+1] \oplus 1$. Therefore, any tampering leads to incorrect output, so the circuit is weakly additively tamper resilient. $\square$

As a side note, the naive binary circuit which requires 2 AND gates per carry computation also has the property of being weakly additively tamper resilient. Because it has 2 AND gates, it can either be the case that $\mathcal{C}(x, y) = \mathcal{C}^*(x, y)$ or $\mathcal{C}(x, y) = \mathcal{C}^*(x, y) \oplus 1$, depending on whether the carry computation was tampered with 1 triple or 2 triples. In either case, this is independent of x and y.

In the case of generating edaBits over $\mathbb{Z}_p$, we still use the ripple-carry adder circuit, and our protocol works as long as the length of the edaBits satisfies $m < \log(p)$. If we wanted edaBits with $m = \lceil \log p \rceil$, for instance to be able to represent arbitrary elements of the field, it seems we would need to use an addition circuit modulo p. Unfortunately, the natural circuit consisting of a binary addition followed by a conditional subtraction is *not* weakly additively tamper resilient. One possible workaround is to use Algebraic Manipulation Detection (AMD) [GIP+14, GIW16] circuits, which satisfy much stronger requirements than being weakly additively tamper resilient, however this gives a very large overhead in practice.

4.2 Overview of Cut-and-Choose Analysis

The remainder of this section is devoted to proving that the cut-and-choose method used in our protocol is sound, as stated in the following theorem.

Theorem 2. *Let $N \geq 2^{s/(B-1)}$ and $C = C' = B$, for some bucket size $B \in \{3, 4, 5\}$. Then the probability that the* CutNChoose *procedure in protocol $\Pi_{\mathsf{edaBitsPriv}}$ outputs at least one incorrect* edaBit *is no more than 2^{-s}.*

Assuming the theorem above, we can prove that our protocol instantiates the desired functionality, as stated in the following theorem. The only interesting aspect to note about security is that we need $m \leq \log M$ to ensure that the value c' computed in step 4d of CutNChoose does not overflow modulo p when $M = p$ is prime. This guarantees that the check values are computed the same way in the binary and arithmetic domains.

Theorem 3. *Protocol $\Pi_{\mathsf{edaBitsPriv}}$ securely instantiates the functionality $\mathcal{F}_{\mathsf{edaBitsPriv}}$ in the $\mathcal{F}_{\mathsf{ABB}}$-hybrid model.*

To give some idea of parameters, in Table 1 we give the required bucket sizes and number N of edaBits that must be produced to ensure 2^{-s} failure probability according to Theorem 2. Note that these are exactly the same bounds as the standard cut-and-choose procedure without any faulty verification steps from [FLNW17]. Our current proof relies on case-by-case analyses for each bucket size, which is why Theorem 2 is not fully general. We leave it as an open problem to obtain a general result for any bucket size.

Overview of Analysis. We analyse the protocol by looking at two abstract games, which model the cut-and-choose procedure. The first game, RealGame,

Table 1. Number of edaBits produced by CutNChoose for statistical security 2^{-s} and bucket size B, with $C = C' = B$.

s	B	# of edaBits
40	3	≥ 1048576
40	4	≥ 10322
40	5	≥ 1024
80	5	≥ 1048576

RealGame

1. $\mathcal{A}$ prepares $NB + C$ shared edaBits $\{([r_j]_M, [r_{j,0}]_2, \ldots, [r_{j,m-1}]_2)\}_{j=1}^{NB+C}$, and batch of $N(B-1) + C'$ potentially tampered circuits $\{\mathcal{C}^*{}_j\}_{j=1}^{N(B-1)}$ to send to the challenger.
2. The challenger shuffles the edaBits and the circuits using 2 permutations.
3. The challenger opens C edaBits in both worlds and C' circuits randomly. If any of the edaBits are inconsistent, or the circuits have been tampered, Abort.
4. Within each bucket, for every pair of edaBits $(r, (r_i)_i)$ and $(s, (s_i)_i)$, take the next circuit C^* and compute $(c_0, \ldots, c_m) = \mathcal{C}^*(r_0, \ldots, r_{m-1}, s_0, \ldots, s_{m-1})$. Compute $c = \sum_{i=0}^{m-1} c_i 2^i$ and check that $r + s - 2^m \cdot c_m$ equals c.

The adversary wins if all the checks pass and there is at least one corrupted edaBit in the output.

Fig. 6. Abstract game modelling the actual cut-and-choose procedure

models the protocol fairly closely, but is difficult to directly analyze. We then make some simplifying assumptions about the game to get SimpleGame, and show that any adversary who wins in the real protocol can be translated into an adversary in the SimpleGame. This is the final game we actually analyze.

4.3 Abstracting the Cut-and-Choose Game

We first look more closely at the cut-and-choose procedure by defining an abstract game, RealGame, shown in Figure 6, that models this process. Note that in this game, the only difference compared with the original protocol is that the adversary directly chooses additively tampered binary circuits, instead of multiplication triples. The check procedure is carried out exactly as before, so it is clear that this faithfully models the original protocol.

Complexities of analyzing the game. In this game, the adversary can pass the check with a bad edaBit in two different ways. The first is to corrupt edaBits in multiples of the bucket size B, and hope that they all end up in the same bucket so that the errors cancel each other out. The second way is to corrupt a set of

SimpleGame

1. $\mathcal{A}$ prepares $NB + C$ balls, corrupts b of them and sends them to the challenger.
2. The challenger opens C of them randomly and checks whether all of them are good. If any one of them is not good, Abort.
3. The challenger permutes and throws NB balls into N buckets each of size B uniformly at random. Then sends the order of arrangement to $\mathcal{A}$.
4. $\mathcal{A}$ prepares $N(B-1) + C'$ triangles, corrupts t of them and sends them to the challenger.
5. The challenger opens C' of them randomly and checks whether all of them are good. If any one of them is not good, Abort.
6. The challenger permutes and throws $N(B-1)$ triangles into N buckets uniformly at random and runs the **Simple** BucketCheck subroutine.
7. If **Simple** BucketCheck returns 1, the challenger outputs first ball from each bucket. Else, Abort.

$\mathcal{A}$ wins if there is no Abort and at least one bad ball is in the output.

Fig. 7. Simplified CutNChoose game

edaBits and guess the permutation in which they are most likely to end up. Once a permutation is guessed, the adversary will know how many triples it needs to corrupt in order to cancel out the errors, and must also hope that the triples end up in the right place.

To compute the exact probability of all these events, we will also have to consider the number of ways in which the bad edaBits can be corrupted. For edaBits which are m bits, there are up to $2^m - 1$ different ways in which they may be corrupted. On top of that, we have to consider the number of different ways in which these bad edaBits may be paired in the check. In order to avoid enumerating the cases and the complex calculation involved, we simplify the game in a few ways which can only give the adversary a better chance of winning. However, we show that these simplifications are sufficient for our purpose.

4.4 The SimpleGame

In this section we analyze a simplified game and bound the success probability of any adversary in that game by 2^{-s}. Before explaining the simple game, we will leave the complicated world of edaBits and triples. We define a TRIP to be a set of triples that is used to check two edaBits. In our simple world edaBits transform into balls, *GOOD* edaBits into white balls (○) and *BAD* edaBits into gray balls (●). An edaBit is *BAD* when at least one of the underlying bits are not correct. TRIPs transform themselves into triangles, *GOOD* TRIPs into white triangles (△) and *BAD* TRIPs into gray triangles (▲). We define a TRIP to be *BAD* when it helps the adversary to win the game, in other words if it can alter the result of addition of two edaBits. Figure 7 illustrates the simple game.

Simple BucketCheck

Input: N buckets and a function f. Each bucket contains B balls $\{x_1, \dots, x_B\}$ and $(B-1)$ triangles $\{y_1, \dots, y_{B-1}\}$.
Output: 0 or 1.
Runs this check in each bucket:

1. Check the configuration of $[x_1, x_i|y_{i-1}]$ $\forall i \in [2, B]$.
   - If $[x_1, x_i|y_{i-1}] \in \{$[○, ○|▲], [○, ●|△], [●, ○|△]$\}$ return Reject.
   - If $[x_1, x_i|y_{i-1}] \in$ [●, ●|△] and $f($●, ●, △$) = 0$ return Reject.
   - If $[x_1, x_i|y_{i-1}] \in$ [●, ●|▲] and $f($●, ●, ▲$) = 0$ return Reject.
2. Otherwise return Accept.

If check returns Accept for all the buckets, then output 1; Otherwise output 0.

Fig. 8. A simple bucket check procedure

In the SimpleGame $\mathcal{A}$ wins if there is no Abort (means $\mathcal{A}$ passes all the checks) and there is at least one bad ball in the final output. The **simple** BucketCheck checks all the buckets. Precisely, in each bucket two balls are being checked using one triangle. For example, let us consider the size of the buckets $B = 3$. Now one bucket contains three balls $[B1, B2, B3]$ and two triangles $[T1, T2]$. Then BucketCheck checks if the configurations $[B1, B2|T1]$ and $[B1, B3|T2]$ matches any one of these configurations {[○, ○|▲], [○, ●|△], [●, ○|△]}. If that is the case then BucketCheck Aborts. When there are two bad balls and one triangle the abort condition depends on the type of bad balls. That means we are considering all bad balls to be distinct, say with different color shades. As a result, in some cases challenger aborts if the checking configuration matches [●, ●|▲] and in other cases it aborts due to [●, ●|△] configuration.

In the simple world everyone has access to a public function f, which takes two bad balls and a triangle as input and outputs 0 or 1. If the output is zero, that means it is a bad configuration, otherwise it is good. This function is isomorphic to the check from step 4 of RealGame, which takes 2 edaBits and a circuit as inputs and outputs the result of the check. The BucketCheck procedure uses f to check all the buckets. Figure 8 illustrates the check in detail. $\mathcal{A}$ passes BucketCheck if all the check configurations are favorable to the adversary. These favorable check configurations are illustrated in Table 2.

Table 2. Favorable combination of balls and triangles for the adversary.

Balls		Triangles
○	○	△
○	●	▲
●	○	▲
●	●	△/▲

After throwing triangles, in each bucket, if the check configuration of balls and triangles are from the first three entries of Table 2, then BucketCheck will not Abort. For the last entry BucketCheck will not Abort if the output of f is 1. Notice that if BucketCheck passes only due to the first configuration of Table 2 in all buckets, then the output from each bucket is going to be a good ball and $\mathcal{A}$ loses. So ideally we should take that into account while computing the winning probability of the adversary. However, for most of the cases it is sufficient to show that for large enough N the $\Pr[\mathcal{A}$ passes BucketCheck$]$ is negligible in the statistical security parameter s, as that will bound the winning probability of $\mathcal{A}$ in the simple game.

Before analyzing the SimpleGame, we show that security of RealGame follows directly from security of SimpleGame. Intuitively, that is indeed the case, as in the SimpleGame an adversary chooses number of bad triangles adaptively; Whereas in the RealGame it has to fix the tampered circuits before seeing the permuted edaBits. Thus, if an adversary cannot win the SimpleGame then it must be more difficult for it to succeed in the RealGame.

Lemma 2. *Security against all adversaries in* SimpleGame *implies security against all adversaries in* RealGame.

Proof. (Sketch.) We prove that by showing if there exist an efficient adversary $\mathcal{B}$ that wins RealGame with non-negligible probability, then there exist an efficient adversary $\mathcal{A}$ against the SimpleGame challenger that wins the game with non-negligible probability. $\mathcal{A}$ simulates the challenger of the RealGame and uses $\mathcal{B}$ to win the SimpleGame. $\mathcal{B}$ sends a batch of edaBits and a set of circuits to $\mathcal{A}$. $\mathcal{A}$ transforms the edaBits into circles. It randomly permutes the circuits, and transforms them into triangles. Clearly, a ball (or triangle) is good or bad depends on whether that was a good or bad edaBit (or a circuit).

$\mathcal{A}$ sends the set of balls to the SimpleGame challenger. The challenger throws them randomly in buckets, sends the arrangement to $\mathcal{A}$. Then $\mathcal{A}$ sends the set of triangles to the challenger. The challenger throws them randomly in buckets, and sends the arrangement to $\mathcal{A}$. In the RealGame $\mathcal{A}$ throws edaBits and the circuits according to the arrangement of balls and triangles in the SimpleGame. Clearly, the simulation is indistinguishable from a RealGame challenger. Thus from the final distribution of triangles, $\mathcal{B}$ cannot distinguish whether it is in the RealGame or in the simulation. Also in the SimpleGame the BucketCheck uses the public function f, which is isomorphic to check function that takes as input two edaBits and a circuit and outputs the result of the check, from step 4 of the RealGame. Consequently, if $\mathcal{B}$ wins with non-negligible probability then $\mathcal{A}$ wins the SimpleGame with a non-negligible probability. $\square$

Throughout the analysis, we use b to denote the number of bad balls and t to denote the number of bad triangles. Now in order to win the SimpleGame the adversary has to pass all the three checks, so let us try to bound the success probability of $\mathcal{A}$ for each of them. Throughout the analysis we consider $N \geq 2^{\frac{s}{B-1}}$, that is for $B \geq 3$, $N(B-1) \geq 2^{\frac{s}{B-1}+1}$ and we are opening $B(\geq 3)$ balls and B triangles in the first two checks.

Opening C balls: In the first check the challenger opens C balls and check whether they are good. So,

$$\Pr[C \text{ balls are good}] = \frac{\binom{NB+C-b}{C}}{\binom{NB+C}{C}} \approx (1 - b/(NB+C))^C.$$

Now for $b = (NB+C)\alpha$, where $1/(NB+C) \leq \alpha \leq 1$, the probability can be written as $(1-\alpha)^C$. In order to bound the success probability of the adversary with the statistical security parameter s, let us consider the case when $\alpha \geq \frac{2^{s/B}-1}{2^{s/B}}$ and $C = B$. Thus,

$$\Pr[C \text{ balls are good}] \approx (1-\alpha)^C = (2^{-s/B})^B = 2^{-s}.$$

So if the challenger opens B balls to check then in order to pass the first check $\mathcal{A}$ must corrupt less than α fraction of the balls, where $\alpha = \frac{2^{s/B}-1}{2^{s/B}}$. Lemma 3 follows from the above analysis.

Lemma 3. *The probability of $\mathcal{A}$ passing the first check in* SimpleGame *is less than 2^{-s}, if the adversary corrupts more than α fraction of balls for $\alpha = \frac{2^{s/B}-1}{2^{s/B}}$ and the challenger opens B balls.*

Opening C' triangles: In this case we'll consider the probability of $\mathcal{A}$ passing the second check. This is similar to the previous check, the only difference is that here the challenger opens C' triangles and checks whether they are good. Consequently,

$$\Pr[C' \text{ triangles are good}] = \frac{\binom{N(B-1)+C'-t}{C'}}{\binom{N(B-1)+C'}{C'}} \approx (1 - t/(N(B-1)+C'))^{C'}.$$

As in the previous case, if t is more than β fraction of the total number of triangles for $\beta = \frac{2^{s/B}-1}{2^{s/B}}$, we can upper bound the success probability of $\mathcal{A}$ by $(2^{-s/B})^{C'}$. Thus for $C' = B$ the success probability of $\mathcal{A}$ in the second check can be bounded by 2^{-s}. Lemma 4 follows from the above analysis.

Lemma 4. *The probability of $\mathcal{A}$ passing the second check in* SimpleGame *is less than 2^{-s}, if the adversary corrupts more than β fraction of triangles for $\beta = \frac{2^{s/B}-1}{2^{s/B}}$ and the challenger opens B triangles.*

Lemmas 3–4 show that it suffices to only look at the first two checks to prove security when the fraction of bad balls or bad triangles is sufficiently large. However, when one of these is small, we also need to analyze the checks within each bucket in the game.

BucketCheck procedure: In this case we consider that the adversary passes first two checks and reaches the last level of the game. However, in order to win the game the adversary has to pass the BucketCheck. Note that now we are dealing with NB balls and the challenger already fixes the arrangement of NB balls in N buckets. Once the ball permutation is fixed that imposes a restriction on the number of favorable (for $\mathcal{A}$) triangle permutations. For example, let us consider that the challenger throws 12 balls into 4 buckets of size 3 and fixes this permutation:

{[●, ○, ○] [○, ○, ●] [●, ●, ○] [○, ○, ○]}

Then there are only two possible favorable permutations of triangles:

{[▲, ▲] [△, ▲] [△, ▲] [△, △]}
{[▲, ▲] [△, ▲] [▲, ▲] [△, △]}

Two favorable permutations come from the fact that the third bucket contains two bad balls. From Table 2 we can see that whenever there are two bad balls in a bucket the adversary can pass the check in that bucket either with a good triangle or with a bad triangle. That means both configurations [●, ●|△] and [●, ●|▲] might be favorable to the adversary. Now $\mathcal{A}$ can use the public function f to determine the value of f(●, ●, △) and f(●, ●, ▲). In this example, let us consider the value of f(●, ●, △) to be 1; Then the first permutation of triangles is favorable to the adversary. As a result the probability of passing the BucketCheck essentially depends on the probability of hitting that specific permutation of triangles among all possible arrangements of triangles. Then the probability of the adversary passing the last check given a specific arrangement of balls L_i is given by:

$$\Pr[\mathcal{A} \text{ passes } \mathsf{BucketCheck}|L_i] = 1/\binom{N(B-1)}{t}$$

where $t = N(B-1)\beta$. Thus,

$$\Pr[\mathcal{A} \text{ passes } \mathsf{BucketCheck}|L_i] = \frac{(N(B-1)\beta)!(N(B-1)(1-\beta))!}{N(B-1)!}$$

In order to upper bound $\Pr[\mathcal{A}$ passes BucketCheck$]$ we will upper bound the probability for different ranges of α and β. Note that the total probability is given by:

$$\Pr[\mathcal{A} \text{ passes } \mathsf{BucketCheck}] = \sum_i \Pr[\mathcal{A} \text{ passes } \mathsf{BucketCheck}|L_i] \cdot \Pr[L_i]$$

If we can argue that for all possible $(2^{s/B}-1)/2^{s/B} \geq \alpha \geq 1/NB$, the maximum probability for $\Pr[\mathcal{A}$ passes BucketCheck$|L_i]$, for some configuration L_i, can be bounded by 2^{-s}, then:

$$\Pr[\mathcal{A} \text{ passes } \mathsf{BucketCheck}] \leq \sum_i 2^{-s} \cdot \Pr[L_i]$$

Note that the maximum possible value of α is 1, however as the challenger opens C balls and C' triangles, the adversary cannot set α to be 1. To pass the first check $\mathcal{A}$ must set α to be less than $(2^{s/B}-1)/2^{s/B}$ if the challenger opens B balls and B triangles.

Now let us try to bound $\Pr[\mathcal{A}$ passes $\mathsf{BucketCheck}|L_i]$. The value of $\binom{N(B-1)}{t}$ maximizes at $t \approx N(B-1)/2$. Starting from the case when there is no bad triangle, the probability monotonically decreases from 1 to its minimum at $\beta \approx 1/2$, and then it monotonically increases to 1 when all triangles are bad. We analyze the success probability of $\mathcal{A}$ in three cases.

Case I $(B-1 \le t \le N(B-1)-(B-1))$**:** Here we are considering the cases when $\mathcal{A}$ chooses number of bad triangles t from the range $[B-1, N(B-1)-(B-1)]$ to maximize its success probability. Now,

$$\Pr[\mathcal{A} \text{ passes } \mathsf{BucketCheck}|L_i] = 1/\binom{N(B-1)}{t}.$$

Clearly, the probability is maximum when t is equal to $(B-1)$ or $N(B-1)-(B-1)$, which is given by:

$$\Pr[\mathcal{A} \text{ passes } \mathsf{BucketCheck}|L_i] = \frac{(B-1)! \cdot (N(B-1)-(B-1))!}{N(B-1)!}$$
$$= \left(\frac{B-1}{N(B-1)}\right) \cdot \left(\frac{B-2}{N(B-1)-1}\right) \cdots \left(\frac{1}{N(B-1)-(B-2)}\right)$$

Now given $N \ge 2^{\frac{s}{B-1}}$ we have,

$$\Pr[\mathcal{A} \text{ passes } \mathsf{BucketCheck}|L_i] \le \left(\frac{1}{2^{\frac{s}{B-1}}}\right)^{B-1} = 2^{-s}.$$

Thus for a given b if the adversary chooses number of bad triangles $t \in [B-1, N(B-1)-(B-1)]$, then:

$$\Pr[\mathcal{A} \text{ passes } \mathsf{BucketCheck}] \le \sum_i 2^{-s} \cdot \Pr[L_i].$$

Given b bad balls and $(NB-b)$ good balls one can arrange them in $NB!/(NB-b)!$ ways. So the probability of hitting a specific arrangement L_i is $(NB-b)!/NB!$. Thus:

$$\Pr[\mathcal{A} \text{ passes } \mathsf{BucketCheck}] \le \frac{NB!}{(NB-b)!} \cdot 2^{-s} \cdot \frac{(NB-b)!}{NB!} = 2^{-s}.$$

Case II $(t > (N(B-1)-(B-1)))$**:** If t is greater than $(N(B-1)-(B-1))$ then the adversary will not be able to pass the second check as the challenger opens $C' = B$ triangles. Thus,

$$\Pr[C' \text{ triangles are good}] = \frac{\binom{N(B-1)+C'-t}{C'}}{\binom{N(B-1)+C'}{C'}} \le \left(1 - \frac{t}{N(B-1)+B}\right)^B$$
$$\le \left(\frac{2B-2}{N(B-1)+B}\right)^B = \left(\frac{2}{N}\right)^B \cdot \left(\frac{B-1}{B-1+\frac{B}{N}}\right)^B,$$

which is less than 2^{-s} given $N \geq 2^{\frac{s}{B-1}}$ and $\frac{s}{B-1} > B$.

Case III ($t < B-1$)**:** Here we try to find the best strategy for the adversary and then show that the success probability can be bounded by 2^{-s} if $N \geq 2^{s/B-1}$. We analyze the probability for three sub-cases, specifically for bucket size 3, 4 and 5, as that allows us to use our cut and choose technique for a wide range of practical parameters.

Bucket size 3*:* For bucket size 3 we have to consider two cases, namely $t = 0$ and $t = 1$. Let us first consider the case when $t = 0$. Clearly, if $\mathcal{A}$ corrupt all the $NB + C$ balls in a way such that $f(\bullet, \bullet, \triangle)$ always returns 1, then the adversary trivially passes BucketCheck. However in that case $\mathcal{A}$ fails with probability 1 as the challenger opens B balls in the first check. If $\mathcal{A}$ corrupts α fraction of NB balls, where $\alpha \geq \frac{2^{s/B}-1}{2^{s/B}}$; Then the success probability of the $\mathcal{A}$ can be bounded by 2^{-s}, if the challenger opens $C = B$ balls in the first check, given $N \geq 2^{\frac{s}{B-1}}$. To pass the first check $\mathcal{A}$ can only corrupt less than α fraction of NB balls. However, in that case the total number of good balls are more than one. Notice that if there is even one good ball out of the NB balls, then in the BucketCheck $[\bullet, \circ | \triangle]$ or $[\circ, \bullet | \triangle]$ check configuration occurs for most of L_is, and $\mathcal{A}$ fails. More precisely, whenever the number of bad balls are not multiple of B, then there exist a bucket with a good ball and a bad ball, thus probability of $\mathcal{A}$ passing BucketCheck becomes zero. When number of bad balls are multiple of B then there exist very few configurations for which the probability of $\mathcal{A}$ passing the BucketCheck is one; For all other possible combinations it become zero. As an example, for $(B = 3, N = 3, K = 2, t = 0)$ only one type of configuration is favorable for the adversary when K is fixed, where K is the number of bad balls to be outputted at the end of the BucketCheck, thus $1 \leq K \leq N - 1$:

$$\{[\bullet, \bullet, \bullet][\bullet, \bullet, \bullet][\circ, \circ, \circ]\}$$

Since $f(\bullet, \bullet, \triangle)$ returns 1, we can assume that all the bad balls are corrupted in the same way. Let us consider $b = KB$, then:

$$\Pr[\mathcal{A} \text{ passes } \mathsf{BucketCheck}] \leq \frac{\binom{N}{K}}{\binom{NB}{KB}}$$

At $K \approx N/2$ this probability reaches its minimum value $2^{-(NB-1)} \ll 2^{-s}$. At $K = 1$ and $K = (N-1)$ the probability reaches its maximum value which is less than $(B-1)!/(NB-(B-1))^{B-1} \leq 2^{-s}$ for $B \geq 3$ as $NB > 2^{s/2}$. Given that, the best strategy of the adversary would be to corrupt one bucket, so that it can pass the first check and hope to hit a favorable configuration in the BucketCheck. However, in that case the probability is still negligible in s. Note that the analysis for this case is same as the one from [FLNW17].

For $t = 1$ the analysis is very much similar to the previous case. Only difference is that now the adversary has to compensate for that one bad triangle. In this case the adversary can win only when the number of bad balls b are KB, $KB - 1$ or $KB + 1$ for $1 \leq K \leq (N-1)$. We are not considering the case when K is N, as in that $\mathcal{A}$ passing the first check is $\mathsf{negl}(s)$. For example for $(B = 3, N = 4, K = 2, t = 1)$, these are three possible type of favorable configurations for the adversary:

{[◍, ◍, ◍][◍, ●, ◍][○, ○, ○][○, ○, ○]}
{[◍, ◍, ◍][◍, ◍, ○][○, ○, ○][○, ○, ○]}
{[◍, ◍, ◍][◍, ◍, ◍][○, ◍, ○][○, ○, ○]}

In the first case there must exist exactly one bad ball pair in one corrupted bucket such that f(◍, ●, ▲) returns 1, thus for that pair the adversary can use the bad triangle. In the second case the adversary uses the bad triangle to check one {bad ball, good ball} pair in the second bucket. In a similar way in the third case $\mathcal{A}$ uses the bad triangle to check one {good ball, bad ball} pair in the third bucket. Note that in the second case the good ball in the second bucket can be placed in four possible positions to generate other favorable permutations. Similarly in the third case the bad ball in the third bucket can be placed in four possible positions to generate other favorable permutations. For all other arrangement the adversary fails BucketCheck, as it has to deal with more than one {bad ball, good ball} pair.

Now the probability of $\mathcal{A}$ passing the BucketCheck for the case when $b = KB$ and $t = 1$ is given by:

$$\Pr[\mathcal{A} \text{ passes } \mathsf{BucketCheck}] \leq \frac{\binom{N}{K}}{\binom{NB}{KB}} \cdot (B-1) \cdot K \cdot \frac{1}{N(B-1)}.$$

The probability of $\mathcal{A}$ passing the BucketCheck when $b = KB - 1$ and $t = 1$ is given by:

$$\Pr[\mathcal{A} \text{ passes } \mathsf{BucketCheck}] \leq \frac{\binom{N}{K}}{\binom{NB}{KB-1}} \cdot (B-1) \cdot K \cdot \frac{1}{N(B-1)}.$$

In the last case for $b = KB + 1$ and $t = 1$ the probability is given by:

$$\Pr[\mathcal{A} \text{ passes } \mathsf{BucketCheck}] \leq \frac{\binom{N}{K}}{\binom{NB}{KB+1}} \cdot (B-1) \cdot (N-K) \cdot \frac{1}{N(B-1)}.$$

The probability of success in the second case for $K = 1$ is higher than the probabilities in the first case for all possible K. In fact it maximizes at $K = 1$ in the second case; which is the same as the highest probability in the third case when $K = (N-1)$. Consequently the best strategy of the adversary would be to corrupt minimum number of balls, to minimize the failure probability of opening and checking C balls, and try to achieve the maximum success probability from the BucketCheck. That means the optimal strategy for the adversary would be the second case with $K = 1$. Thus,

$$\Pr[\mathcal{A} \text{ passes } \mathsf{BucketCheck}] \leq \frac{(B-1)!}{(NB-(B-2))^{B-1}} \leq 2^{-s}, \ \textit{for } B \geq 3.$$

Bucket size 4*:* The analysis for the cases $B = 4$, $t = 0$ and $t = 1$ follows directly from the analysis from $B = 3$. In other words, the configurations remain the same, the only difference being the bucket size is now 4.

For bucket size $B = 4$ and $t = 2$, there are six possible favorable configurations for the adversary when K is fixed, where K is the number of bad balls to be outputted at the end of the BucketCheck, thus $1 \leq K \leq N - 1$. For example for $(B = 4, N = 4, K = 2, t = 2)$ these are the six possible configurations for the adversary:

{[●, ●, ●, ●][●, ●, ●, ○][○, ○, ●, ○][○, ○, ○, ○]}
{[●, ●, ●, ●][●, ●, ○, ○][○, ○, ○, ○][○, ○, ○, ○]}
{[●, ●, ●, ●][●, ●, ●, ●][○, ○, ●, ○][○, ●, ○, ○]}
{[●, ●, ●, ●][●, ●, ●, ⬤][○, ○, ●, ○][○, ○, ○, ○]}
{[●, ●, ●, ●][●, ●, ⬤, ○][○, ○, ○, ○][○, ○, ○, ○]}
{[●, ●, ●, ●][●, ●, ⬤, ⬤][○, ○, ○, ○][○, ○, ○, ○]}

For all these cases the success probability of the adversary in the BucketCheck can be expressed as:

$$\begin{aligned}
&\Pr[\mathcal{A} \text{ passes } \mathsf{BucketCheck}] \\
&\leq \binom{N}{K}\binom{K(B-1)}{g_1 + b_1}\binom{(N-K)(B-1)}{b_2}\frac{1}{\binom{NB}{KB - g_1 + b_2}}\frac{1}{\binom{N(B-1)}{t}},
\end{aligned} \tag{2}$$

where g_1 is the total number of good balls in the chosen K buckets (which output bad balls at the end of BucketCheck), b_1 is the total number of different kind of bad balls(⬤), such that f(●, ⬤, ▲) returns 1 and b_2 is the total number of bad balls from other $(N - K)$ buckets. Note that number of bad triangles t is equal to $g_1 + b_1 + b_2$. As an example let us consider the first configuration among the six favorable configurations; In that case $g_1 = 1$, $b_1 = 0$ and $b_2 = 1$, thus:

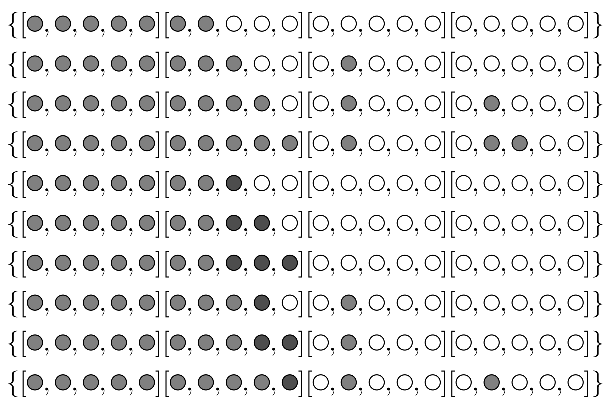
$$\Pr[\mathcal{A} \text{ passes } \mathsf{BucketCheck}] \leq \binom{N}{K}\binom{K(B-1)}{1}\binom{(N-K)(B-1)}{1}\frac{1}{\binom{NB}{KB}}\frac{1}{\binom{N(B-1)}{t}}.$$

Now for each of these configurations the probability is maximum either at $K = 1$ or at $K = N - 1$. After calculating the probabilities for each of these configurations at $K = 1$ and $K = N - 1$ it is easy to see that the success probability of the adversary is maximum in the fourth case for $K = N - 1$. Thus:

$$\begin{aligned}
&\Pr[\mathcal{A} \text{ passes } \mathsf{BucketCheck}] \leq 9N(N-1) \cdot \frac{1}{\binom{4N}{3}} \cdot \frac{1}{\binom{3N}{2}} \\
&= \left(\frac{3N-3}{3N-1}\right) \cdot \left(\frac{3}{4N}\right) \cdot \left(\frac{2}{4N-1}\right) \cdot \left(\frac{2}{4N-2}\right) \\
&\leq \left(\frac{3N-3}{3N-1}\right) \cdot 2^{-s/3} \cdot 2^{-s/3} \cdot 2^{-s/3} \leq 2^{-s}, \ \textit{given } N \geq 2^{s/B-1}.
\end{aligned}$$

Bucket size 5*:* Once again, the analysis from the previous cases carries over for $t = 0, 1, 2$, $t = 3$ being the only new case we have to analyze.

For the case when B = 5 and t = 3, there are 10 favorable configurations for the adversary when K is fixed. For N = 4 and K = 2, these are the cases:



Using eq. (2) we can calculate the probabilities for these 10 cases at $K = 1$ and $K = N - 1$ to find the best case scenario for the adversary. Doing so, we found that the first case from the figure at $K = 1$, and the fourth case at $K = N - 1$ have the best probabilities. Considering the first case, this would be,

$$
\begin{aligned}
&\Pr[\mathcal{A} \text{ passes } \mathsf{BucketCheck}] \leq \binom{N}{1} \cdot \binom{4}{3} \cdot \frac{1}{\binom{5N}{2}} \cdot \frac{1}{\binom{4N}{3}} \\
&= N \cdot 4 \cdot \frac{1}{\binom{5N}{2}} \cdot \frac{1}{\binom{4N}{3}} \\
&= \left(\frac{2}{5N}\right) \cdot \left(\frac{6}{5N-1}\right) \cdot \left(\frac{1}{4N-1}\right) \cdot \left(\frac{1}{4N-2}\right) \\
&\leq 2^{-s/4} \cdot 2^{-s/4} \cdot 2^{-s/4} \cdot 2^{-s/4} \leq 2^{-s}, \text{ } \textit{given } N \geq 2^{s/B-1}.
\end{aligned}
$$

Even though the probability is the same for the fourth case with $K = N - 1$, since the number of bad balls are much higher than the first case, the overall probability will be lower for the fourth, making the first case the best one.

We summarize the analysis as follows.

Lemma 5. *The probability of $\mathcal{A}$ passing the* BucketCheck *in* SimpleGame *is less than 2^{-s}, if $N \geq 2^{s/(B-1)}$ and the challenger opens $C = B$ balls and $C' = B$ triangles during first two checks of* SimpleGame *for $B \in \{3, 4, 5\}$ given $\frac{s}{B-1} > B$.*

Proof. This follows from the case-by-case analysis of BucketCheck procedure, together with Lemma 3 and Lemma 4. $\square$

Combining Lemma 2 and Lemma 5, this completes the proof of Theorem 2.

Remark 1. As we already mentioned the bound we obtain is not general. However, from Lemma 5 it is evident that one can produce more than 1024 edaBits efficiently with 40-bit statistical security using different bucket sizes with our CutNChoose technique, which is sufficient for the applications we are considering in this work. It also shows that if we want to achieve 80-bit statistical security for $N \geq 2^{20}$, then increasing the bucket size from 3 to 5 would be sufficient. Table 1 shows the number of edaBits we can produce with different size of buckets.

5 Primitives

This section describes the high-level protocols we build using our edaBits, both over $\mathbb{Z}_{2^k}$ and $\mathbb{Z}_p$. We focus on secure truncation (Section 5.1) and secure integer comparison (Section 5.2), although our techniques apply to a much wider set of non-linear primitives that require binary circuits for intermediate computations. For example, our techniques also allow us to compute binary-to-arithmetic and arithmetic-to-binary conversions of shared integers, by plugging in our edaBits into the conversion protocols from [Cd10] and [DEF$^+$19] for the field and ring cases, respectively.

Throughout this section our datatypes are signed integers in the interval $[-2^{\ell-1}, 2^{\ell-1})$. On the other hand, our MPC protocols operate over a modulus $M \geq 2^\ell$ which is either 2^k or a prime p. Given an integer $\alpha \in [-2^{\ell-1}, 2^{\ell-1})$, we can associate to it the corresponding ring element in $\mathbb{Z}_M$ by computing $\alpha \bmod M \in \mathbb{Z}_M$ (modular reduction returns integers in $[0, M)$). We denote this map by $\mathsf{Rep}_M(\alpha)$, and we may drop the sub-index M when it is clear from context. Finally, in the protocols below LT denotes a binary less-than circuit.

5.1 Truncation

Recall that our datatypes are signed integers in the interval $[-2^{\ell-1}, 2^{\ell-1})$, represented by integers in $\mathbb{Z}_M$ where $M \geq 2^\ell$ via $\mathsf{Rep}_M(\alpha) = \alpha \bmod M$. The goal of a truncation protocol is to obtain $[y]$ from $[a]$, where $y = \mathsf{Rep}\left(\left\lfloor \frac{\alpha}{2^m} \right\rfloor\right)$ and where $a = \mathsf{Rep}(\alpha)$. This is a crucial operation when dealing with fixed-point arithmetic, and therefore an efficient solution for it has a substantial impact in the efficiency of MPC protocols for a wide range of applications. An important observation is that, as integers, $\left\lfloor \frac{\alpha}{2^m} \right\rfloor = \frac{\alpha - (\alpha \bmod 2^m)}{2^m}$. If M is an odd prime p, this corresponds in $\mathbb{Z}_p$ to $y = (\mathsf{Rep}(\alpha) - \mathsf{Rep}(\alpha \bmod 2^m)) \cdot \mathsf{Rep}(2^m)^{-1}$. Furthermore, $\mathsf{Rep}(\alpha \bmod 2^m) = \alpha \bmod 2^m = a \bmod 2^m$ and $\mathsf{Rep}(2^m) = 2^m$, so $y = \frac{a - (a \bmod 2^m)}{(2^m)^{-1}}$.

Truncation over $\mathbb{Z}_{2^k}$. Truncation protocols over fields typically exploit the fact that one can divide by powers of 2 modulo p. This is not possible when working modulo 2^k. Instead, we take a different approach. Let $[a]_{2^k}$ be the initial shares, where $a = \mathsf{Rep}(\alpha)$ with $\alpha \in [-2^{\ell-1}, 2^{\ell-1})$ (notice that it may be the case that

$\ell < k$). First, we provide a method, $\mathsf{LogShift}$, for computing the *logical* right shift of a by m positions, assuming that $a \in [0, 2^\ell)$. That is, if a is

$$(\underbrace{0, \ldots, 0}_{k-\ell}, \underbrace{a_{\ell-1}, \ldots, a_0}_{\ell}),$$

this procedure will yield shares of

$$(\underbrace{0, \ldots, 0}_{k-\ell+m}, \underbrace{a_{\ell-1}, \ldots, a_m}_{\ell-m}).$$

Then, to compute the arithmetic shift, we use the fact that[4]

$$\left\lfloor \frac{\alpha}{2^m} \right\rfloor \equiv \mathsf{LogShift}_m(a + 2^{\ell-1}) - 2^{\ell-m-1} \bmod 2^k.$$

Now, to compute the logical shift, our protocol begins just like in the field case by computing shares of $a \bmod 2^m$ and subtracting them from a, which produces shares of $(a_{k-1}, \ldots, a_m, 0, \ldots, 0)$. The parties then open a masked version of $a - (a \bmod 2^m)$ which does not reveal the upper $k - \ell$ bits, and then shift to the right by m positions in the clear, and undo the truncated mask. One has to account for the overflow that may occur during this masking, but this can be calculated using a binary LT circuit.

The details of our logical shift protocol are provided in Fig. 9, and we analyze its correctness next. First, it is easy to see that $c = 2^{k-m}((a + r) \bmod 2^m)$, so $c/2^{k-m} = (a \bmod 2^m) + r - 2^m v$, where v is set if and only if $c/2^{k-m} < r$. From this we can see that the first part of the protocol $[a \bmod 2^m]_{2^k}$ is correctly computed. Privacy of this first part follows from the fact that $r \bmod 2^m$ completely masks $a \bmod 2^m$ when c is opened.

For the second part, let us write $b = 2^m a'$, then $d = 2^{k-\ell+m}((a' + r') \bmod 2^{\ell-m})$, so $d/2^{k-\ell+m} = a' + r' - 2^{\ell-m}u$, where u is set if and only if $d/2^{k-\ell+m} < r'$, as calculated by the protocol. We get then that $a' = d/2^{k-\ell+m} - r' + 2^{\ell-m}u$, and since a' is precisely $\mathsf{LogShift}_m(a)$, we conclude the correctness analysis.

Probabilistic Truncation. Recall that in the field case one can obtain probabilistic truncation avoiding a binary circuit, which results in a constant number of rounds. Over rings this is a much more challenging task. For example, probabilistic truncation with a constant number of rounds is achieved in ABY3 [MR18], but requires, like in the field case, a 2^s gap between the secret values and the actual modulus, which in turn implies that only small non-negative values can be truncated.

In Fig. 10, we take a different approach. Intuitively, we follow the same approach as in ABY3, which consists of masking the value to be truncated with a shared random value for which its corresponding truncation is also known,

[4] Notice that we can use the $\mathsf{LogShift}$ method on $a + 2^{\ell-1}$ since, $\alpha + 2^{\ell-1} \in [0, 2^\ell)$, which implies that $(a + 2^{\ell-1}) \bmod 2^k = \alpha + 2^{\ell-1}$ and therefore $(a + 2^{\ell-1}) \bmod 2^k$ is ℓ-bits long, as required.

Logical right shift over $\mathbb{Z}_{2^k}$

Pre:
- $\mathcal{F}_{\mathsf{ABB}}$
- Input $[a]_{2^k}$ where $a \in [0, 2^\ell)$.
- Number of bits to shift m
- edaBit $([r]_{2^k}, [r]_2)$ of length m
- edaBit $([r']_{2^k}, [r']_2)$ of length $\ell - m$

Post: $[y]_{2^k}$, where $y = \mathsf{LogShift}_m(a)$.

1. The parties compute shares of $a \bmod 2^m$ as follows:
   (a) Call $c = \mathsf{open}\left(2^{k-m} \cdot ([a]_{2^k} + [r]_{2^k})\right)$
   (b) Compute $[v]_2 = \mathsf{LT}((c_i)_{i=k-m+1}^{k}, ([r_i]_2)_{i=0}^{m-1})$
   (c) Convert $[v]_2 \mapsto [v]_{2^k}$
   (d) Let $[a \bmod 2^m]_{2^k} = 2^m\,[v]_{2^k} - [r]_{2^k} + c/2^{k-m}$.
2. The parties compute the truncation:
   (a) Compute $[b]_{2^k} = [a]_{2^k} - ([a]_{2^k} \bmod 2^m)$.
   (b) Call $d = \mathsf{open}(2^{k-\ell} \cdot ([b]_{2^k} + 2^m\,[r']_{2^k}))$.
   (c) Compute $[u]_2 = \mathsf{LT}((d_i)_{i=k-\ell+m}^{k-1}, ([r'_i]_2)_{i=0}^{\ell-m-1})$
   (d) Convert $[u]_2 \mapsto [u]_{2^k}$.[a]
   (e) Output $[y]_{2^k} = 2^{\ell-m}\,[u]_{2^k} + d/2^{k-\ell+m} - [r']_{2^k}$

[a] One can optimize this by noticing that we only need shares of u modulo $2^{k-\ell+m}$.

Fig. 9. Protocol for performing logical right-shift

opening this value, truncating it and removing the truncated mask. In ABY3 a large gap is required to ensure that the overflow that may happen by the masking process does not occur with high probability. Instead, we allow this overflow bit to be non-zero and remove it from the final expression. Doing this naively would require us to compute a LT circuit, but we avoid doing this by using the fact that, because the input is positive, the overflow bit can be obtained from the opened value by making the mask value also positive. This leaks the overflow bit, which is not secure, and to avoid this we mask this single bit with another random bit. This protocol can be seen as an extension of the probabilistic truncation protocol by Dalskov et al. [DEK19]. Below, we provide an analysis for our extension that also applies to said protocol.

Now we analyze the protocol. First we notice that $c = 2^{k-\ell-1}c'$ where $c' = (2^m r + r') + a + 2^\ell b - 2^{\ell+1} v b$, where v is set if and only if $(2^m r + r') + a$ overflows modulo 2^ℓ. It is easy to see that this implies that $c'_\ell = v \oplus b$, so we see that $v = c'_\ell \oplus b$, as calculated in the protocol.

On the other hand, we have that $(c' \bmod 2^\ell) = (2^m r + r') + a - 2^\ell v$, so $a \bmod 2^m = (c' \bmod 2^m) - r' + 2^m u$, where u is set if $(c' \bmod 2^m) < r'$. From this it can be obtained that $\left\lfloor (c' \bmod 2^\ell)/2^m \right\rfloor - r + 2^{\ell-m} = \lfloor a/2^m \rfloor + u$.

Probabilistic truncation over $\mathbb{Z}_{2^k}$

Pre:

- $\mathcal{F}_{\mathsf{ABB}}$
- Input $[a]_{2^k}$ where $a \in [0, 2^\ell)$.
- $\ell < k$
- Number of bits to truncate m
- edaBit $([r]_{2^k}, [r]_2)$ of length $(\ell - m)$
- edaBit $([r']_{2^k}, [r']_2)$ of length m
- Random bit $[b]_{2^k}$

Post: $[y]_{2^k}$ where $y = \lfloor a/2^m \rfloor + u$ with $u = 1$ with probability $(a \bmod 2^m)/2^m$.

1. Call $c = \mathsf{open}(2^{k-\ell-1} \cdot ([a]_{2^k} + 2^\ell [b]_{2^k} + 2^m [r]_{2^k} + [r']_{2^k}))$. Write $c = 2^{k-\ell-1} c'$.
2. Compute $[v]_{2^k} = [b \oplus c'_\ell]_{2^k} = [b]_{2^k} + c'_\ell - 2c'_\ell [b]_{2^k}$
3. Output $[y]_{2^k} = (c' \bmod 2^\ell)/2^m - [r]_{2^k} + 2^{\ell-m} [v]_{2^k}$

Fig. 10. Probabilistic truncation in domain modulo power of two using edaBits

Remark 2. The protocol we discussed above only works if $a \in [0, 2^\ell)$, that is, if the value α represented $\alpha \in [0, 2^{\ell-1})$. We can extend it to $\alpha \in [-2^{\ell-1}, 2^{\ell-1})$ by using the same trick as in the deterministic truncation: The truncation is called with $a + 2^{\ell-1}$ as input, and $2^{\ell-m-1}$ is subtracted from the output.

Truncation over Fields. We begin with a protocol, presented originally by Catrina and de Hoogh [Cd10], and optimize it with our edaBits. For this protocol we require a larger gap between the shares and the secret to be truncated, more precisely, it must hold that $p > 2^{\ell+s+1}$, where s is the statistical security parameter. The protocol is presented in Fig. 11.

To see the correctness of the protocol, begin by observing that because $p > 2^{\ell+s+1}$, and since $b \in [0, 2^\ell)$ the addition of b and $2^m r + r'$ does not overflow modulo p and therefore c is actually equal to $b + 2^m r + r'$, as integers. This preserves the privacy of b as $b \in [0, 2^\ell)$ and $2^m r + r'$ is uniformly random in $[0, 2^{\ell+s+1})$. Given this, it holds then that $(c \bmod 2^m) = (b \bmod 2^m) + (2^m r + r' \bmod 2^m) - v \cdot 2^m$, where $v \in \{0, 1\}$ is set if and only if $(b \bmod 2^m) + (r \bmod 2^m) \notin [0, 2^m)$. Now, observe that this condition triggers if and only if $c \bmod 2^m = \sum_{i=0}^{m-1} c_i 2^i$ is smaller than $r \bmod 2^m = \sum_{i=0}^{m-1} r_i 2^i$, so the bit v can be obtained by executing a (unsigned) binary less-than circuit as done by the protocol. We remark that for this step we use our optimized binary-shared bits, which provides an important optimization with respect to the protocol from Catrina et al.

Taking into account that $(2^m r + r') \bmod 2^m = r'$, and also that $a \equiv b \bmod 2^{\ell-1}$, $(c \bmod 2^m) - r' + v \cdot 2^m$ is the same as $a \bmod 2^m$, we obtain that the first part of the protocol in which shares of $a \bmod 2^m$ are computed is correct. Finally, the ending step computes the formula for the truncation, which concludes the correctness analysis.

Deterministic Truncation over $\mathbb{F}_p$

Pre:

- Shares $[a] = [\mathsf{Rep}(\alpha)]$, integer $0 < m < \ell$.
- edaBit $([r]_M, [r]_2)$ of length $\ell - m + s$.
- edaBit $([r']_M, [r']_2)$ of length m.

Post: Shares $[y]$ where $y = \mathsf{Rep}\left(\left\lfloor \frac{\alpha}{2^m} \right\rfloor\right)$.

1. First the parties compute shares of $a \bmod 2^m$ as follows:
   (a) Let $[b] = 2^{\ell-1} + [a]$;
   (b) Call $c = \mathsf{open}([b] + 2^m[r] + [r'])$;
   (c) The parties compute $[v]_2 = \mathsf{LT}\left((c_i)_{i=0}^{m-1}, ([r'_i]_2)_{i=0}^{m-1}\right)$;
   (d) Convert $[v]_2 \mapsto [v]$.
   (e) Let $[a \bmod 2^m] = [c \bmod 2^m] - [r'] + [v]2^m$.
2. Compute the truncated value using the formula as follows. Let $(2^m)^{-1}$ be the inverse of 2^m modulo p. Output $[y] = (2^m)^{-1} \cdot ([a] - [a \bmod 2^m])$.

Fig. 11. Deterministic truncation over fields with share gap

Probabilistic Truncation. The protocol above is not constant round, as it requires the computation of a less-than circuit on inputs of length m. It turns out that if one is willing to allow for some small error, a much more efficient protocol can be devised, as by Catrina and Saxena [CS10]. This protocol follows the same blueprint as the deterministic one, except that the computation of the overflow bit v is omitted. The description of the protocol can be found in Fig. 12. Following the analysis from the previous protocol, this implies that the value d computed in the protocol is $d = (a \bmod 2^m) - 2^m v$, so the final value computed is $(a - (a \bmod 2^m))/2^m + v$, which is the desired truncation, off by at most one bit. Furthermore, it is easy to see that the result is biased towards the nearest truncation.

5.2 Integer Comparison

Another important primitive that appears in many applications is integer comparison. In this case, two secret integers $[a]_M$ and $[b]_M$ are provided as input, and the goal is to compute shares of $\alpha \overset{?}{<} \beta$, where $a = \mathsf{Rep}(\alpha)$ and $b = \mathsf{Rep}(\beta)$.

As noticed by previous works (e.g. [Cd10, DEF+19]), this computation reduces to extracting the MSB from a shared integer as follows: If $\alpha, \beta \in [-2^{k-2}, 2^{k-2})$, then $\alpha - \beta = [-2^{k-1}, 2^{k-1})$, so $a - b = \mathsf{Rep}(\alpha - \beta)$ corresponds to the sign of $\alpha - \beta$, which is minus (i.e. the bit is 1) if and only if α is smaller than β.

To extract the MSB, we simply notice that $\mathsf{MSB}(\alpha) = -\left\lfloor \frac{\alpha}{2^{k-1}} \right\rfloor \bmod 2^k$, so this can be extracted with the protocols we have seen in the previous sections.

Probabilistic truncation over $\mathbb{F}_p$

Pre:

- $\mathcal{F}_{\mathsf{ABB}}$ with $p > 2^{k+s+1}$.
- Shares $[a] = [\mathsf{Rep}(\alpha)]$, integer $0 < m < k$.
- edaBit $([r]_M, [r]_2)$ of length $k - m + s$.
- edaBit $([r']_M, [r']_2)$ of length m.

Post: Shares $[y]$ where $y \approx \mathsf{Rep}\left(\left\lfloor \frac{\alpha}{2^m} \right\rfloor\right)$.

1. Let $[b] = 2^{k-1} + [a]$;
2. Call $c = \mathsf{open}([b] + 2^m[r] + [r'])$;
3. Let $[d] = [c \bmod 2^m] - [r']$.
4. Output $[y] = (2^m)^{-1} \cdot ([a] - [d])$.

Fig. 12. Probabilistic truncation over fields.

Table 3. Amortized costs for generating 1 Private, and 1 Global edaBit. Costs for Global edaBits do not include the cost of the n additional sets of Private edaBits that are needed.

	Private edaBits		Global edaBits	
	$\mathbb{Z}_{2^k}$	$\mathbb{F}_p$	$\mathbb{Z}_{2^k}$	$\mathbb{F}_p$
Faulty edaBits	B	B	0	$0\ (l - m + s,\ m)$
Faulty Triples	$(B-1)m$	$(B-1)m$	0	0
Secure Triples	0	0	$(\log n)(n-1)$	$(\log n)(n-1)$
daBits	0	$(B-1)$	0	$\log n$
Openings ($\mathbb{Z}_2$)	$(3m+1)(B-1)$	$(3m+1)(B-1)$	$(2m + 2\log n)(n-1)$	$(2m + 3\log n)(n-1)$
Openings ($\mathbb{Z}_M$)	$(B-1)$	$(B-1)$	0	0

6 Applications and Benchmarks

6.1 Theoretical Cost

We present the theoretical costs of the different protocols in the paper, starting with the cost for producing Private and Global edaBits in terms of the different parameters.

Table 3 shows the main amortized costs for generating a Private and Global edaBit of length m. For Global edaBits, we assume have the required correct Private edaBits to start with, which is why number of Faulty edaBits needed is 0. B is the bucket size for the cut-and-choose procedure and n is the number of parties.

Table 4 shows the cost for two of our primitives from Section 5, namely comparison of m-bit numbers and truncation of an ℓ-bit number by m binary digits. For computation modulo a prime, there is also a statistical security parameter s.

Comparison in $\mathbb{Z}_{2^k}$ is our only application where it suffices to use loose edaBits (where the relation between the sets of shares only holds modulo 2^m, c.f. Sec-

Table 4. Cost of our primitives. Numbers in brackets indicate edaBit length.

	Comparison		Truncation	
	$\mathbb{Z}_{2^k}$	$\mathbb{F}_p$	$\mathbb{Z}_{2^k}$	$\mathbb{F}_p$
Strict edaBits	0	2 $(m+1,\, s+1)$	2 $(l-m,\, m)$	2 $(l-m+s,\, m)$
Loose edaBits	1 $(m+1)$	0	0	0
classic daBits	1	1	2	1
Online ANDs	$\sim 2m$	$\sim 2m$	$\sim 2m$	$\sim 2k$

tion 3.2). This is because the arithmetic part of an edaBit is only used in the first step (the masking) but not at the end. Recall that the truncation protocols always use the arithmetic part of an edaBit twice, once before opening and once to compute an intermediate or the final result. Using a loose edaBit would clearly distort the result. With comparison on the other hand, an edaBit is only used to facilitate the conversion to binary computation, after which the result is converted back to arithmetic computation using a classic daBit.

6.2 Implementation Results

We have implemented our approach in a range of domains and security models, and we have run the generation of a million edaBits of length 64 on AWS `c5.9xlarge` with the minimal number of parties required by the security model (two for dishonest majority and three for honest majority). Table 5 shows the throughput for various security models and computation domains, and Table 6 does so for communication. In the prime field case, we use $\log p \approx 128$ to allow additional room needed for comparisons, while for arithmetic mod 2^k we use $k = 64$. For computation modulo a prime with dishonest majority, we present figures for arithmetic computation both using oblivious transfer (OT) [KOS16] and LWE-based semi-homomorphic encryption (HE) [KPR18]. Note that the binary computation is always based on oblivious transfer for dishonest majority and that all our results include all consumable preprocessing such as multiplication triples but not one-off costs such as key generation. The source code of our implementation has been added to MP-SPDZ [CSI20].

We have also implemented 63-bit[5] comparison using edaBits, only daBits, and neither, and we have run one million comparisons in parallel again on AWS `c5.9xlarge`. Table 7 shows the throughput for our various security models and computation domains, and Table 8 does so for communication. Note that the arithmetic baseline uses either the protocol of Catrina and de Hoogh [Cd10] ($\mathbb{F}_p$) or the variant by Dalskov et al. [DEK19] ($\mathbb{Z}_{2^k}$).

Our results highlight the advantage of our approach over using only daBits. The biggest improvement comes in the dishonest majority with semi-honest security model. For the dishonest majority aspect, this is most likely because there

[5] Comparison in secure computation is generally implemented by extracting the most significant bit of difference. This means that 63-bit is the highest accuracy achievable in computation modulo 2^{64}, which the natural modulus on current 64-bit platforms.

Table 5. Number of edaBits generated (in 1000s) per second in various settings

		Domain	Strict edaBits	Loose edaBits
Dishonest maj.	Malicious	2^k (OT)	4.6	7.3
		p (OT)	3.6	4.2
		p (HE)	2.7	3.4
	Semi-hon.	2^k (OT)	456.7	922.5
		p (OT)	228.0	892.6
		p (HE)	470.5	905.6
Honest maj.	Malicious	2^k	191.5	205.8
		p	156.6	162.1
	Semi-hon.	2^k	2032.1	7180.0
		p	1367.7	4934.3

Table 6. Communication per edaBit (in kbit) in various settings

		Domain	Strict edaBits	Loose edaBits
Dishonest maj.	Malicious	2^k (OT)	1335.5	480.2
		p (OT)	1936.9	1473.2
		p (HE)	940.8	779.7
	Semi-hon.	2^k (OT)	22.5	9.6
		p (OT)	43.9	9.6
		p (HE)	11.8	9.6
Honest maj.	Malicious	2^k	5.6	3.7
		p	7.6	6.4
	Semi-hon.	2^k	0.3	0.2
		p	0.5	0.2

is a great gap in the cost between multiplications and inputs (the latter is used extensively to generate edaBits). For the semi-honest security aspect, note that our approach for malicious security involves a cascade of sacrificing because the edaBit sacrifice involves binary computation, which in turn involves further sacrifice of AND triples. Finally, the improvement in communication is generally larger than the improvement in wall clock time. We estimate that this is due to the fact that switching to binary computation clearly reduces communication but increases the computational complexity.

6.3 Comparison to Previous Works

Dishonest majority. The authors of HyCC [BDK$^+$18] report figures for biometric matching with semi-honest two-party computation in ABY [DSZ15] and HyCC. The algorithm essentially computes the minimum over a list of small-dimensional Euclidean distances. The aforementioned authors report figures in LAN (1Gbps) and artificial WAN settings of two machines with four-core i7 processors. For a

Table 7. Number of comparisons (in 1000s) per second in various settings

		Domain	Arithm.	daBits	edaBits
Dishonest maj.	Malicious	2^k (OT)	0.5	1.2	4.4
		p (OT)	0.3	0.3	1.6
		p (HE)	0.6	0.7	2.0
	Semi-hon.	2^k (OT)	5.2	14.4	275.6
		p (OT)	1.6	3.3	79.7
		p (HE)	5.9	12.8	170.6
Honest maj.	Malicious	2^k	76.4	119.2	170.4
		p	66.9	78.3	80.1
	Semi-hon.	2^k	500.6	1007.7	1607.6
		p	157.8	277.1	457.6

Table 8. Communication per comparison (in kbit) in various settings

		Domain	Arithm.	daBits	edaBits
Dishonest maj.	Malicious	2^k (OT)	21737.7	9058.6	1310.5
		p (OT)	40108.5	34019.1	4783.3
		p (HE)	3020.5	3210.9	1584.8
	Semi-hon.	2^k (OT)	2283.0	830.2	39.0
		p (OT)	7353.1	3503.0	134.9
		p (HE)	411.6	219.1	38.7
Honest maj.	Malicious	2^k	63.4	27.8	5.4
		p	94.3	85.0	19.9
	Semi-hon.	2^k	14.5	7.1	0.4
		p	37.4	23.1	1.4

fair comparison, we have run our implementation using one thread limiting the bandwidth and latency accordingly. Table 9 shows that our results improves on the time in the LAN setting and on communication generally as well as on the in the WAN setting for larger instances compared to their A+B setting (without garbled circuits). The WAN setting is less favorable to our solution because it is purely based on secret sharing and we have not particularly optimized the number of rounds.

Honest majority (three parties, one semi-honest corruption). Our approach is not directly comparable to ABY3 by Mohassel and Rindal [MR18] because they use the specifics of replicated secret sharing for the conversion. We do note however that their approach of restricting binary circuits to the binary domain is comparable to our solution, and that they use the same secret sharing schemes as us in the 2^k domain. We compare their results with our approach applied to logistic regression. Their software implementation [MR19] runs all parties on the same host without communication encryption. For a fair comparison, we

Table 9. Overall time and communication for biometric matching

		LAN (s)	WAN (s)	Comm. (MB)
$n = 1000$	ABY/HyCC (A+Y)	0.22	2.5	9.5
	ABY/HyCC (A+B)	0.22	6.1	10.6
	Ours	0.12	8.3	7.4
$n = 4096$	ABY/HyCC (A+Y)	0.63	6.6	40.4
	ABY/HyCC (A+B)	0.72	13.6	43.6
	Ours	0.48	12.6	29.1
$n = 13684$	ABY/HyCC (A+Y)	3.66	17.5	138.0
	ABY/HyCC (A+B)	5.4	26.2	190.8
	Ours	2.00	22.9	111.8

have run their software as well as ours in the same setting on the same desktop machine with an i7 processor. In our software, we use the special truncation according to Dalskov et al. [DEK19] and either edaBits or bit decomposition as in the work above for comparison. The comparison in turn is used for a piece-wise approximation of the sigmoid function. Table 10 shows that edaBit-based comparison generally comes close to ABY3's bit decomposition.

Dimension	Batch size	ABY3 [MR18]	Ours (ABY3 comp.)	Ours (edaBits)
10	128	1495	1801	1671
	256	1402	1407	1230
	512	1229	1014	827
	1024	976	656	479
100	128	1303	1372	1269
	256	1064	988	904
	512	732	657	560
	1024	349	387	316
1000	128	327	436	422
	256	148	284	271
	512	74	167	159
	1024	35	90	84

Table 10. Iterations per second for logistic regression

daBits. Aly et al. [AOR+19] report figures for daBit generation with dishonest majority and malicious security in eight threads over a 10 Gbps network. For two-party computation using homomorphic-encryption, they achieve 2150 daBits per second at a communication cost of 94 kbit per daBit. In a comparable setting, we found that our protocol produces 12292 daBits per second requiring a communication cost of 32 kbit. Note however that Aly et al. use somewhat homomorphic encryption while our implementation is based on cheaper semi-homomorphic encryption.

Convolutional Neural Networks. We also apply our techniques to the convolutional neural networks.Dalskov et al. [DEK19] present an implementation for deep learning inference. We have adapted their implementation to our setting and present a comparison for the simplest network (MobileNet V1 0.25_128) in Table 11. It shows that edaBits reduce the communication and time in most security models. The only exception is semi-honest honest-majority computation modulo 2^k, where Dalskov et al. use the conversion by Mohassel et al. [MR18], which has similar properties to our approach. The figures for malicious protocols have been generated using bucket size four because the batches would otherwise far exceed the required edaBits.

		Domain		Time (s)	Comm. (GB)
Dish. maj.	Mal.	2^k (OT)	[DEK19]	1264.9	1748.4
			Ours	455.3	561.9
		p (HE)	[DEK19]	1377.8	282.4
			Ours	552.9	299.9
	S-h.	2^k (OT)	[DEK19]	139.5	199.2
			Ours	23.8	32.4
		p (HE)	[DEK19]	129.1	37.1
			Ours	22.4	6.8
Hon. maj.	Mal.	2^k	[DEK19]	5.3	2.5
			Ours	3.4	2.2
		p	[DEK19]	9.0	8.7
			Ours	8.3	4.6
	S-h.	2^k	[DEK19]	0.2	0.1
			Ours	0.3	0.1
		p	[DEK19]	3.3	3.4
			Ours	2.2	0.3

Table 11. Time and communication for MobileNet inference

Acknowledgements. We thank Deevashwer Rathee and the authors of [CGR+19] for pointing out corrections to our cost analysis, Sameer Wagh for helpful comments on an earlier version of the paper, and the anonymous CRYPTO 2020 reviewers for valuable feedback. This work has been supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme under grant agreements No 669255 (MPCPRO) and

No 803096 (SPEC), the Danish Independent Research Council under Grant-ID DFF–6108-00169 (FoCC), the Concordium Blockhain Research Center, Aarhus University and an Aarhus University Forskningsfond (AUFF) starting grant.

Bibliography

[ABF$^+$18] Toshinori Araki, Assi Barak, Jun Furukawa, Marcel Keller, Yehuda Lindell, Kazuma Ohara, and Hikaru Tsuchida. Generalizing the SPDZ compiler for other protocols. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 880–895. ACM Press, October 2018.

[AOR$^+$19] Abdelrahaman Aly, Emmanuela Orsini, Dragos Rotaru, Nigel P. Smart, and Tim Wood. Zaphod: Efficiently combining LSSS and garbled circuits in SCALE. In *WAHC '19: Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. ACM, 2019. `https://eprint.iacr.org/2019/974`.

[BCG$^+$19a] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, Peter Rindal, and Peter Scholl. Efficient two-round OT extension and silent non-interactive secure computation. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 291–308. ACM Press, November 2019.

[BCG$^+$19b] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, and Peter Scholl. Efficient pseudorandom correlation generators: Silent OT extension and more. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part III*, volume 11694 of *LNCS*, pages 489–518. Springer, Heidelberg, August 2019.

[BDK$^+$18] Niklas Büscher, Daniel Demmler, Stefan Katzenbeisser, David Kretzmer, and Thomas Schneider. HyCC: Compilation of hybrid protocols for practical secure computation. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 847–861. ACM Press, October 2018.

[BGI15] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 337–367. Springer, Heidelberg, April 2015.

[BGI19] Elette Boyle, Niv Gilboa, and Yuval Ishai. Secure computation with preprocessing via function secret sharing. In Dennis Hofheinz and Alon Rosen, editors, *TCC 2019, Part I*, volume 11891 of *LNCS*, pages 341–371. Springer, Heidelberg, December 2019.

[BLN$^+$15] Sai Sheshank Burra, Enrique Larraia, Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, Emmanuela Orsini, Peter Scholl, and Nigel P. Smart. High performance multi-party computation for binary circuits based on oblivious transfer. Cryptology ePrint Archive, Report 2015/472, 2015. `http://eprint.iacr.org/2015/472`.

[BLW08] Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A framework for fast privacy-preserving computations. In Sushil Jajodia and Javier López, editors, *ESORICS 2008*, volume 5283 of *LNCS*, pages 192–206. Springer, Heidelberg, October 2008.

[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 FOCS*, pages 136–145. IEEE Computer Society Press, October 2001.

[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.

[CGR$^+$19] Nishanth Chandran, Divya Gupta, Aseem Rastogi, Rahul Sharma, and Shardul Tripathi. EzPC: Programmable and efficient secure two-party computation for machine learning. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 496–511. IEEE, 2019.

[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.

[CSI20] CSIRO's Data61. MP-SPDZ. `https://github.com/data61/MP-SPDZ`, 2020.

[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.

[DEK19] Anders Dalskov, Daniel Escudero, and Marcel Keller. Secure evaluation of quantized neural networks. Cryptology ePrint Archive, Report 2019/131, 2019. `https://eprint.iacr.org/2019/131`.

[DFK$^+$06] Ivan Damgård, Matthias Fitzi, Eike Kiltz, Jesper Buus Nielsen, and Tomas Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In Shai Halevi and Tal Rabin, editors, *TCC 2006*, volume 3876 of *LNCS*, pages 285–304. Springer, Heidelberg, March 2006.

[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.

[DN07] Ivan Damgård and Jesper Buus Nielsen. Scalable and unconditionally secure multiparty computation. In Alfred Menezes, editor, *CRYPTO 2007*, volume 4622 of *LNCS*, pages 572–590. Springer, Heidelberg, August 2007.

[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.

[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.

[FLNW17] Jun Furukawa, Yehuda Lindell, Ariel Nof, and Or Weinstein. High-throughput secure three-party computation for malicious adversaries and an honest majority. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017, Part II*, volume 10211 of *LNCS*, pages 225–255. Springer, Heidelberg, April / May 2017.

[GIP+14] Daniel Genkin, Yuval Ishai, Manoj Prabhakaran, Amit Sahai, and Eran Tromer. Circuits resilient to additive attacks with applications to secure computation. In David B. Shmoys, editor, *46th ACM STOC*, pages 495–504. ACM Press, May / June 2014.

[GIW16] Daniel Genkin, Yuval Ishai, and Mor Weiss. Binary AMD circuits from secure multiparty computation. In Martin Hirt and Adam D. Smith, editors, *TCC 2016-B, Part I*, volume 9985 of *LNCS*, pages 336–366. Springer, Heidelberg, October / November 2016.

[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.

[IKNP03] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 145–161. Springer, Heidelberg, August 2003.

[IMZ19] Muhammad Ishaq, Ana L. Milanova, and Vassilis Zikas. Efficient MPC via program analysis: A framework for efficient optimal mixing. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 1539–1556. ACM Press, November 2019.

[Kel20] Marcel Keller. MP-SPDZ: A versatile framework for multi-party computation. Cryptology ePrint Archive, Report 2020/521, 2020. `https://eprint.iacr.org/2020/521`.

[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.

[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.

[MR19] Payman Mohassel and Peter Rindal. ABY3, 2019. `https://github.com/ladnir/aby3/`.

[NNOB12] Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. A new approach to practical active-secure two-party computation. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 681–700. Springer, Heidelberg, August 2012.

[RST$^+$19] Dragos Rotaru, Nigel P. Smart, Titouan Tanguy, Frederik Vercauteren, and Tim Wood. Actively secure setup for SPDZ. Cryptology ePrint Archive, Report 2019/1300, 2019. `https://eprint.iacr.org/2019/1300`.

[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.

[WRK17a] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Authenticated garbling and efficient maliciously secure two-party computation. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 21–37. ACM Press, October / November 2017.

[WRK17b] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. Global-scale secure multiparty computation. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 39–56. ACM Press, October / November 2017.

A Classic daBits

Recall that a (classic) daBit is defined as a pair $([b]_M, [b]_2)$, where $b \in \{0,1\}$ is a random bit. We make use of these daBits to convert one single bit from the binary world to the arithmetic world. Classic daBits can be preprocessed as described in [AOR$^+$19, RW19, RST$^+$19, DEF$^+$19], for example. First, we review at a very high level how these methods work. Then, in Section A.1, we present the explicit protocols we use in our implementation for generating daBits, and their relation to the works we mentioned above.

Marbled Circuits [RW19]. Each party proposes a set of daBits, whose consistency is checked via cut-and-choose techniques. Then these bits are XORed together to output the final daBits. This method works for both $M = p$ and $M = 2^k$ with minor modifications.

Zaphod [AOR$^+$19]. First arithmetic shares of random bits are produced. Then these are converted to binary shares by observing that the overflow bits in the arithmetic world are rather predictable if the shares are only between two parties. The resulting binary-shared bits may not be correct, so a consistency check is put in place. This works by taking a linear random combination of the bits in both worlds and checking its consistency (in the arithmetic world the LSB must be extracted, which requires an extra sub-protocol). This method is suited for $M = p$.

Actively Secure Setup for SPDZ [RST$^+$19]. This works considers a much more general concept of daBits in which bits can be shared modulo many different primes. The layout of the protocol is similar to the one from Zaphod: Random bits are generated modulo a large-enough prime, and these are converted locally to shares over the integers. Then these are converted to shares modulo each desired prime, and their correctness is checked via linear combinations. Since in [RST$^+$19] the odd primes may be small, the authors have to consider a variant of the subset sum problem to argue security. When instantiating their method with 2 and our large prime p, we notice that their methods essentially lead to an optimized version of Zaphod (in fact, when the odd primes are large enough one can avoid the subset-sum assumption entirely by masking the upper bits as done in Zaphod).

SPDZ2k [DEF$^+$19]. The tools presented in this work are enough to produces daBits, although the authors do not consider this concept explicitly. In a nutshell, this approach would follow the exact same template as in Zaphod, making use of the fact that in SPDZ2k, the parties can obtain binary additive shares of an arithmetically-shared bit b by simply considering the LSB of their shares. Compare this to the field case, where the overflow bit mod p must be predicted and corrected. Furthermore, one can also observe than in SPDZ2k opening the

Generation of faulty daBits

Pre:

- $\mathcal{F}_{\mathsf{ABB}}$
- Threshold t (maximal number of corrupted parties)

Post: supposed daBit $([b]_M, [b]_2)$

1. $t+1$ parties (w.l.o.g $P_1, \dots, P_{t+1}$) each input a bit b_i into $\mathcal{F}_{\mathsf{ABB}}$ both mod M and mod 2, resulting in $([b_i]_M, [b_i]_2)$ for $i = 1, \dots, t+1$.
2. All parties compute $([b]_M, [b]_2) = ([\bigoplus_{i=1}^{t+1} b_i]_M, [\bigoplus_{i=1}^{t+1} b_i]_2)$. The first half can be computed using the fact that $a \oplus b = a + b - 2ab$ for $a, b \in \{0,1\} \subset \mathbb{Z}$ while the second is straight-forward given that $a \oplus b = a + b$ for $a, b \in \mathbb{Z}_2$.

Fig. 13. Protocol to generate supposed daBits in any domain

LSB of an arithmetically shared value is also efficient and does not require any overhead with respect to opening the full value (in fact, it is more efficient), unlike the field case.

A.1 Our daBit Implementation

Our daBit generation over is similar to the one considered in Zaphod [AOR+19]. However, we modify the first step in which arithmetic shares of a random bit are produced. Instead of using the random-bit generation from SPDZ, we let each party share an arithmetic bit and then these will be added to produce the desired bit. This is presented in Fig. 13. The result is trivially correct if all parties are honest. Furthermore, as the number of participating is larger than the number of corrupted parties, the results is a random bit from the view of the adversary in that case. The protocol costs t multiplications in $\mathcal{F}_{\mathsf{ABB}}$.

We also notice that if the arithmetic modulus is a power of two, it is easy to construct a daBit from a random bit by having the parties input the least significant bit of their share to the binary computation and then computing the XOR without communication. In other words, parties can locally convert an additive secret sharing modulo 2^k locally. Let b_i denote an additive share of b modulo 2^k held by P_i. Then, $b_i \bmod 2$ is a valid share of b modulo 2:

$$\sum (b_i \bmod 2) \bmod 2 = \left(\sum b_i \bmod 2^k\right) \bmod 2 = b \bmod 2.$$

This is precisely how Zaphod converts from modulo p to modulo 2, but they do not consider the modulo 2^k case. We present this optimization in Fig. 14. Furthermore, as a bonus, we observe that in the honest majority setting where no MAC are required this procedure can be made much simpler, and we present this in Fig. 15

Note that our protocol for SPDZ2k is more general than the one proposed by Damgård et al. [DEF+19] because theirs only works if the binary part of $\mathcal{F}_{\mathsf{ABB}}$ is

SPDZ2k daBit generation

Pre:

1. $\mathcal{F}_{\mathsf{ABB}}$ with the arithmetic part based on SPDZ2k
2. Total number of parties n

Post: supposed daBit $([b]_{2^k}, [b]_2)$ where $[b]_{2^k}$ is guaranteed to be in $\{0, 1\}$

1. The parties generate a random bit $[b]_{2^k}$ as described by Damgård et al. [DEF$^+$19].
2. Let b_i denote the additive share of b held by P_i, that is $b = \sum_{i=1}^{n} b_i \mod 2^k$. P_i inputs $b_i \bmod 2$ to the binary part of $\mathcal{F}_{\mathsf{ABB}}$.
3. The parties compute $[b]_2 = \bigoplus_{i=1}^{n} [b_i \bmod 2]_2$.

Fig. 14. Protocol to generate supposed daBits with SPDZ2k

daBit generation modulo in $\mathbb{Z}_{2^k}$ without MAC

Pre: $\mathcal{F}_{\mathsf{ABB}}$ where the arithmetic part is based on purely on additive or replicated secret sharing and the binary part uses the same secret sharing scheme

Post: supposed daBit $([b]_{2^k}, [b]_2)$ where $[b]_{2^k}$ is guaranteed to be in $\{0, 1\}$

1. The parties generate a random bit $[b]_{2^k}$ in the arithmetic part of $\mathcal{F}_{\mathsf{ABB}}$.
2. Let $\{b_i^1, \ldots, b_i^m\}$ denote the shares of b held by P_i. P_i computes $\{b_i^1 \bmod 2, \ldots, b_i^m \bmod 2\}$ and uses them as shares for the binary part of $\mathcal{F}_{\mathsf{ABB}}$.

Fig. 15. Protocol to generate supposed daBits in protocols module 2^k without MAC

implemented by SPDZ2k for $k = 1$, which has the disadvantage that computing an AND has cost quadratic in the security parameter s whereas the protocol by Frederiksen et al. [FKOS15] for example has linear cost in that regard while achieving the same security properties.

In our construction two things must be checked to prevent cheating from an active adversary. First, as in Zaphod, parties may cause the final daBit to be inconsistent, in the sense that the arithmetic and binary parts may contain different bits. Second, unlike the construction from Zaphod, it is not guaranteed that the value each party inputs is indeed a bit.

To fix the first issue we simply resort to the same technique as in Zaphod of computing s random linear combinations modulo two in both domains, after which s daBits have to be discarded for privacy. This method has asymptotically no overhead in terms of daBits being produced because the batch can be arbitrarily large. On the other hand, to fix the second issue, we check that the arithmetic part of each of the final daBits contains indeed a bit, which can be done by checking $x(1 - x) = 0$ with x being the arithmetic share. This adds one multiplication per daBit. Furthermore, we notice that we are checking that the final daBit contains a bit, rather than checking that each of the original daBits

daBit check

Pre: m supposed bits $([b_i]_M, [b_i]_2)$ in $\mathcal{F}_{\mathsf{ABB}}$ where $m > s$ for statistical security parameter s

Post: $m - s$ verified daBits

1. The parties do the following s times:
   (a) Generate m fresh public random bits r_i
   (b) Compute $[\bigoplus_{i=1}^{m} r_i \cdot b_i]_2$ and open it.
   (c) Compute $[r] := [\sum_{i=1}^{m} r_i \cdot b_i]_M$.
      – If $M = 2^k$, call $r' = \mathsf{open}([r \cdot 2^{k-1}]_{2^k})$ and compute $r'/2^{k-1} = (r \cdot 2^{k-1} \bmod 2^k)/(2^{k-1}) = r \bmod 2$.
      – If $M = p$, call $r' = \mathsf{open}([r]_p + 2 \cdot \sum_{i=0}^{s+1} [c_i]_p \cdot 2^i)$ with random bits $[c_i]_p$ and compute $r \bmod 2 = r' \bmod 2$.

      Abort if $r \bmod 2$ does not match the bit from the previous step.
2. Discard $([b_i]_M, [b_i]_2)$ for $i \in [m - s + 1, m]$.
3. For $i \in [1, m - s]$, compute and open $[b_i \cdot (1 - b_i)]_M$. Abort if any value is not zero.[a]

[a] This check may be omitted if $M = 2^k$ and the bit generation via SPDZ2k from Fig. 14 is used.

Fig. 16. Protocol to check classic daBits

input by each party contain a bit. This is more efficient and it is also secure, as there is at least one honest party who inputs a bit, and therefore the XOR operation becomes an oblivious selection between x or $1-x$, where x is the XOR of the arithmetic shares of the adversary. If the result is a bit, then x was a bit to begin with.

Fig. 16 shows our adapted checking protocol. Aly et al. argue that any incorrect daBit would lead to a 1/2 probability of failure in step 1c, hence s independent repetitions would fail at least once with overwhelming probability. They also argue that discarding s daBits after the checks protects the secrecy of the remaining ones.