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 multi-party computation. To this end, 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, our main observations are that (1) applications that benefit from daBits can also benefit from edaBits in the same way, and (2) we can generate edaBits directly in a much more efficient way than computing them from a set of daBits. Technically, the second contribution is much more challenging, and involves a novel cut and choose technique that may be of independent interest, and requires taking advantage of natural tamper-resilient properties of binary circuits that occur in our construction to obtain the best level of efficiency. Finally, we show how our 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 yield a substantial increase in efficiency. Our edaBits save in communication by a factor that lies between 2 and 170 for secure comparisons with respect to a purely arithmetic approach, and between 2 and 60 with respect to using daBits. Improvements in throughput per second are more subdued 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 [20] (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 [2, 29], dishonest majority with malicious security [33], as well as creating compilers that automatically decide which parts of a program should done in the binary or arithmetic domain [9, 27].

A particular technique that is relevant for us is so-called *daBits* [33] (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, but seem most promising for the case of a dishonest majority with malicious security. Later works also presented more efficient ways of generating daBits [1, 32], all based on the SPDZ protocol [19] using homomorphic encryption.

Another recent work uses function secret sharing [6] for binary/arithmetic conversions and other operations such as comparison [7]. 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 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 [15, 16]. 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 [17, 19], over the arithmetic domain $\mathbb{Z}_p$ for large prime p, or $\mathbb{Z}_{2^k}$ [12]. Unlike previous works in this setting, 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 edaBits 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 *private* edaBit $(r^i_{m-1}, \dots, r^i_{m-1}), r^i$, and secret-share this in both domains in MPC. Then, the parties 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 MACs as in SPDZ [19]. Secondly, we introduce a novel 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 [8, 21, 22]. 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. To pass the check, the adversary must have corrupted both r and s so that they cancel each other out; by carefully choosing parameters, we can ensure that it is very unlikely the adversary manages to do this for every pair with a bad edaBit. For example, with 40-bit statistical security, from the analysis of [22], we can 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, using protocols based on the TinyOT family [21, 31, 34].

To reduce this cost, one possible 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 radical optimization.

Cut-and-choose with faulty check circuits. Instead of using private multiplication 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? Intuitively, 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 [23], 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. This means we can only generate restricted-length edaBits in $\mathbb{Z}_p$, that is for $m < \log p$, which turns out to be sufficient for our applications. It as an interesting open problem to construct a simple, weakly additively tamper-resilient circuit for addition modulo p.

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.

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 [5]. In practice, the communication cost when using this "silent OT" method can be more than 100x less than OT extension based on previous techniques [26], with a modest increase in computation [4]. 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. [1]. 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 and 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 [15]. 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, in all our constructions we take great care 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 [14, 15], 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 [1]. 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. [25] 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, 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, the reduction in communication lies between a factor of 2 and 170 for comparisons from purely arithmetic to edaBits and between 2 and 60 from daBits to edaBits. Improvements in throughput per second are more subdued but still as high as a factor of 47. Generally, the improvements are higher for dishonest-majority computation and semi-honest security.

We have also compared our implementation with the most established software for mixed circuits [9] 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 eight 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. In Section 4.1 we describe abstract games that model the cut-and-choose, 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 $M = 2$. Secret shared values in these domains are denoted by $[x]_M$.

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 [10]. 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 [33]), 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, since we are interested in the relation between binary and arithmetic computation, we will consider an extended version of the arithmetic black box model. First, in one single instance of the functionality the computation can be both binary or arithmetic, where the latter can be either modulo p or modulo 2^k. Furthermore, the functionality allows the parties to convert binary shares to arithmetic shares.[3] The details are presented in Fig. 1.

There are several ways to instantiate the basic arithmetic commands of this functionality, depending on the adversarial setting. For passive security basic secret-sharing techniques suffices. For active security in the honest majority setting one can use for example Shamir secret-sharing or replicated-secret sharing [3, 18], and for active security in the dishonest majority scenario they can be instantiated with secret-sharing-based techniques coupled with MACs [12, 17, 28, 34]. Furthermore, the conversions between the arithmetic bits and binary sharings can be implemented via daBits, as shown in [1, 32, 33]. We present a short summary of this daBit generation in Section A in the appendix.

[3] Converting m bits to arithmetic shares of the integer they represent requires calling this command m times. Using our edaBits, we can optimize this substantially, as we briefly discuss in Section 5. In our protocols we will only call this command to convert one single bit from binary to arithmetic representation.

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 $x \in \mathbb{Z}_M$, 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.

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 $(r_{m-1}, \ldots, r_0)$ 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 $r_{i,0}^i, \ldots, r_{i,m-1}^i$, 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.

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

This functionality samples m random bits $r_0, \ldots, r_{m-1} \in \mathbb{Z}_2$ and stores them in the binary ABB. It also stores $r = \sum_{i=0}^{m-1} r_i 2^i$ in the arithmetic ABB. Here $M \in \{2^k, p\}$. The functionality has the same features as $\mathcal{F}_{\mathsf{ABB}}$, together with the following:

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.

3.1 Functionality for Private Extended daBits

Recall the protocol for generating edaBits begins with each party proposing a set of edaBits, which will be checked and combined afterwards. Functionality $\mathcal{F}_{\mathsf{edaBitsPriv}}$ formalizes this notion. This functionality is defined similarly to $\mathcal{F}_{\mathsf{edaBits}}$, except that the bits $r_0, \ldots, r_{m-1}$ are provided using the Input command by one party.

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

3.2 From Private to Global Extended daBits

As we discussed already at the beginning of this section, one can instantiate the $\mathcal{F}_{\mathsf{edaBits}}$ functionality assuming access to a $\mathcal{F}_{\mathsf{edaBitsPriv}}$ functionality and combining the different private edaBits to ensure no individual party knows the underlying values. The protocol is described in detail in the following subsections. 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 $r' = \sum_{i=1}^{n} r_i$, and they can also add their binary shares using a binary n-input adder, denoted BitADD, which results in shares of the bits of r'. Since we want to output a random m-bit integer, the parties need to remove the bits of r' beyond the m-th bit, which can be done since the carry bits are part of the output of the binary adder. This requires $\log(n)$ calls to $\mathsf{ConvertB2A}$ of $\mathcal{F}_{\mathsf{ABB}}$, each of which uses a (regular) daBit, except for the case of $M = 2^k$ and $m = k$, where we can omit the conversions.

One must be careful with potential overflows modulo M. If this addition overflows and $M = 2^k$, then the overflow bits beyond the k-th position must be discarded. On the other hand, if $M = p$, an overflow modulo p would affect all bits, and to avoid this we require that $m < \log(p)$. The details are in Fig. 3, and the security of the protocol is stated in Theorem 1, whose proof follows in a straightforward manner from the correctness of the additions in the protocol. In the protocol, nBitADD denotes an n-input binary adder.

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 $([r_i]_M, [r_{i,0}]_2, \ldots, [r_{i,m-1}]_2)$ for $i = 1, \ldots, n$, where party P_i knows $(r_{i,0}, \ldots, r_{i,m-1})$ and where $r_i = \sum_{j=1}^{m-1} r_{i,j} 2^j$.
2. Parties invoke the ABB to compute $[r']_M = \sum_{i=1}^{n} [r_i]_M$.
3. Parties invoke the ABB to compute $\mathsf{nBitADD}\left(([r_{1,j}]_2)_j, \ldots, ([r_{n,j}]_2)_j\right)$. This produces $m + \log n$ bits $([b_0]_2, \ldots, [b_{m+\log(n)-1}]_2)$.
4. Parties use ConvertB2A 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. Parties use the ABB to compute $[r]_M = [r']_M - 2^m \sum_{j=0}^{\log(n)-1} [b_{j+m}]_M 2^j$.
6. Parties output $([r]_M, [b_0]_2, \ldots, [b_{m-1}]_2)$.

Fig. 3. Protocol for generating global edaBits from private 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 $\mathsf{edaBits}$ (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 cut-and-choose-based check in which a random subset of certain size of $\mathsf{edaBits}$ is opened, its correctness is checked, and then the remaining $\mathsf{edaBits}$ are randomly placed into buckets. Within each bucket, all $\mathsf{edaBits}$ but the first one are checked against the first edaBit by adding the two and opening the result. With high probability, the first edaBit will be correct if all the checks pass.

In the method above, when adding two $\mathsf{edaBits}$ together, the parties must make some calls to the binary multiplication feature of the ABB, and these multiplications must be correct for the basic analysis to work. In practice these multiplications are instantiated via TinyOT triples, and thus the correctness requirement can be translated into these triples being correct. To generate these triples, first potentially incorrect triples are produced, and then correctness of these triples is checked via "sacrificing" techniques.

The sacrifice step required to produce correct triples is not cheap. In this work we take a different approach that leads to much higher efficiency in practice. First, we allow some of the triples used to perform the check within each bucket to be incorrect, which saves in resources as the sacrifice 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 propose the triples together with the $\mathsf{edaBits}$, which is much cheaper than letting the parties jointly sample (even incorrect) triples.

Now, an issue we face when describing the protocol sketched above is that our arithmetic black box model does not consider the case of a "faulty multiplication" in which the product can be flipped (which is the effect of an incorrect bit multiplication triple). Furthermore, even if we added this feature to the functionality, this would not suffice for our purposes since we cannot allow the adversary to choose the exact multiplication gates in which he can cause the additive attack. Instead, we consider the following *macros*, which can be instantiated given the existing features of the functionality:

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 $z = x \cdot y \oplus (c \oplus a \cdot b)$,[4] and store (id, z).

Now we are ready to present our protocol to preprocess private $\mathsf{edaBits}$. Our protocol is described in Fig. 4. It makes use of a subroutine $\mathsf{CutNChoose}$ to check that the $\mathsf{edaBits}$ provided by party P_i are consistent, which is defined in Fig. 5.

[4] It can be seen that in Beaver-based binary multiplication, the error in the product is exactly equal to the error in the triple.

In the protocol, BitADDCarry denotes the same circuit with carry. As we will see later in this section it can be computed with m AND gates.[5]

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

Pre: $\mathcal{F}_{\mathsf{ABB}}$
Post: Batch of N shared edaBits $\{([r_j]_M, [r_{j,0}]_2, \ldots, [r_{j,m-1}]_2)\}_{j=1}^{L}$, where party P_i knows the underlying bits.

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

Fig. 4. Protocol for producing private extended daBits.

The remaining subsections are devoted to proving that the cut-and-choose method used in our protocol is sound, as stated in the following (informal) theorem.

Theorem 2. (Informal.) *If the* CutNChoose *check in protocol $\Pi_{\mathsf{edaBitsPriv}}$ passes, then, with overwhelming probability, the first* edaBit *of each bucket is correct.*

Let us assume for now the theorem above. In this case, we can easily prove that our protocol instantiates the desired functionality, as stated in the following theorem. Its proof follows trivially from the previous theorem and we therefore omit it.

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

4.1 Cut-and-choose Analysis

The cut and choose analysis on the edaBits proposed by one party P_i is modeled by a game played between an adversary and a challenger, where party P_i is

[5] This circuit is rather naive, and in fact there exist circuits that have logarithmic depth at a small extra cost in the 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 as they have certain resilient properties that we exploit explicitly. Furthermore, they are only used in the preprocessing phase, and thus the overhead is not noticeable in practice.

Procedure CutNChoose

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

1. Open randomly selected C edaBits in both worlds and C' triples. Abort if any of the edaBits or the triples are inconsistent.
2. Shuffle the remaining edaBits and triples using 2 public permutations π_1, π_2 respectively and put the edaBits into buckets of size B and the triples into buckets of size $B \cdot m$.
3. Perform the following check on every bucket. If the check passes in all the buckets, Accept the edaBits, else Abort:
   (a) Select the top edaBit, $[r]_M, [r_0]_2, \ldots, [r_{m-1}]_2$ from the bucket and call $\mathcal{F}_{\mathsf{ABB}}$ to compute $[r+s]_M$ for every other edaBit s in the bucket.
   (b) Compute BitADDCarry($[r_0]_2, \ldots, [r_{m-1}]_2, [s_0]_2, \ldots, [s_{m-1}]_2$) using the FaultyMult command with every other edaBit s in the bucket. Then extract the carry bit c_{m+1} at position $(m+1)$.
   (c) Convert $[c_{m+1}]_2 \mapsto [c_{m+1}]_M$.
   (d) Let $c' = [r+s]_M - 2^{m+1} \cdot [c_{m+1}]_M$. Open c' and the values from the binary world.
   (e) If all the checks pass, Accept, else Abort.

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

the adversary. We start by presenting the RealGame for the cut and choose procedure, and an abstract version of the game, which makes it easier to prove certain desired properties. Motivation for simplifying the game is argued by presenting the complexities with analyzing the RealGame, due to which we move into the SimpleGame. Following that, it is shown that if the adversary can win in the SimpleGame, it can be win in the RealGame. Finally, the concrete probability of winning in the SimpleGame is analyzed and shown to be not more than 2^{-s}, where s is the statistical security parameter.

4.2 The RealGame

The RealGame is played between an adversary $\mathcal{A}$ and a challenger. The goal of the adversary is to pass the cut and choose game such that the game outputs some corrupted edaBits. It involves the adversary proposing a set of edaBits and a set of TinyOT triples [8, 31], which are used in the game to verify the correctness of the proposed edaBits. The challenger takes all the edaBits and the triples proposed, and randomly chooses C edaBits and C' group of triples to be opened. If all of them are consistent, it randomly permutes the edaBits and the triples and places them into buckets. Then, similar to what is done by

Furukawa et al. [22], the top edaBit from each bucket is checked with every other edaBit in the bucket using the triples. Every individual check in the CutNChoose procedure takes two edaBits of m bits each, and consumes m triples, as shown in the checking mechanism. The formal description of CutNChoose procedure is given in Figure 5.

The Checking Mechanism: For every pair of edaBits we select, we add the shares of them over $\mathbb{Z}_M$ and $\mathbb{Z}_2$ and open them. Adding the shares over $\mathbb{Z}_M$ is local whereas adding them over $\mathbb{Z}_2$ requires evaluation of a binary adder circuit. We use a Ripple Carry Adder circuit which computes the carry at every bit 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_i, c_{i+1} are the carries at positions $i, i+1$ respectively, and x_i, y_i are the values at position i in the binary world.

To do CutNChoose naively, the TinyOT triples used in Equation (1) would have to be authenticated and verified. The most efficient way to do this is to perform a cut-and-choose procedure on the TinyOT triples proposed by $\mathcal{A}$. Instead of going through this process, we propose using TinyOT triples with MACs which may potentially be corrupted. We let $\mathcal{A}$ propose the required batch of the triples (with MACs) along with the set of edaBits. We will show in the analysis later that giving the adversary this advantage will not affect the outcome of the game except with negligible probability.

Before going into details about whether the check will pass, we describe a property of the circuits that is required. Then, using the abstract game defined in $\Pi_{\mathsf{AbstractGame}}$, we argue that the RealGame satisfies the property.

Definition 1. *A circuit $\mathcal{C} : \mathbb{F}_M^2 \to \mathbb{F}_M$ is weakly additively tamper resilient, if given a tampered circuit $\mathcal{C}^*$, obtained by additively tampering $\mathcal{C}$, either of the following properties holds:*

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

To prove that the binary adder circuit used in the CutNChoose procedure satisfies this property, we must first model tampering of the circuit. In the adder circuit, the adversary can only change the output of the circuit by using bad triples instead of good ones. Since a set of triples is used to compute the AND between two bits, a bad set of triples can only flip the output, which is equivalent to saying that the adversary introduced an additive error in the circuit. An abstraction of the RealGame using $\mathcal{C}, \mathcal{C}^*$ is presented in Figure 6.

Lemma 1. *Protocol $\Pi_{\mathsf{AbstractGame}}$ is an abstraction of the* RealGame

Proof. In the abstract game, both the checks, over $\mathbb{Z}_{2^m}$ and $\mathbb{Z}_p$ are modeled by the equation:

$$[x+y]_M = \mathcal{C}^*([x_1]_2, \dots, [x_{m-1}]_2, [y_1]_2, \dots, [y_{m-1}]_2) \tag{2}$$

Protocol $\Pi_{\mathsf{AbstractGame}}$

Pre: A batch of $NB + C$ 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 $\{\mathcal{C}^*{}_i\}_{i=1}^{N(B-1)}$
Post: Batch of N shared edaBits, where party P_i knows the underlying bits.

1. Open C edaBits in both worlds and C' circuits randomly. Abort if any of the edaBits or the circuits are tampered with.
2. Shuffle the remaining edaBits and the AND gates across all circuits using 2 permutations and put the edaBits and the circuits into buckets.
3. Within each bucket, for every pair of edaBits, r, s, check that $[r+s]_M = \mathcal{C}^*([r_1]_2, \ldots, [r_{m-1}]_2, [s_1]_2, \ldots, [s_{m-1}]_2)$.

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

Fig. 6. Protocol for the Abstract Game.

The triples are abstracted away and represented by the tampered circuit $\mathcal{C}^*$. However, when computing the carry in both domains, $2^{m+1} \cdot c_{m+1}$ is subtracted from $[x+y]_M$, where c_{m+1} is the carry at position $(m+1)$.

Let $\mathcal{C}^*([x_1]_2, \ldots, [x_{m-1}]_2, [y_1]_2, \ldots, [y_{m-1}]_2) = (sum_B, c_{m+1})$, where sum_B is the sum of the first m bits. By definition, we can write this as:

$$sum_B = \mathcal{C}^*([x_1]_2, \ldots, [x_{m-1}]_2, [y_1]_2, \ldots, [y_{m-1}]_2) - 2^{m+1} \cdot c_{m+1}. \qquad (3)$$

In the arithmetic world, the output can be written as:

$$sum_A = [r+s]_M - 2^{m+1} \cdot c_{m+1} \qquad (4)$$

In the CutNChoose procedure, the equality of both outputs sum_A, sum_B is checked. From Equations (3) and (4), it is clear that this is equivalent to checking if $[r+s]_M = \mathcal{C}^*([x_1]_2, \ldots, [x_{m-1}]_2, [y_1]_2, \ldots, [y_{m-1}]_2)$. Since this is equivalent to checking the locally added shares over $\mathbb{Z}_M$ and the output of the binary circuit, we conclude that Protocol $\Pi_{\mathsf{AbstractGame}}$ is a valid abstraction of the RealGame. □

We need this property because it restricts the adversary from being able to use a tampered circuit with bad edaBits as well as with good edaBits. It ensures that if the circuit has been tampered in any position, the check at that position would only pass with either a good edaBit, or a bad edaBit. There exists a public function g, which takes two edaBits as input, and outputs the circuit ($\mathcal{C}$ or $\mathcal{C}^*$), for which the two edaBits will pass the check.

Lemma 2. *The ripple carry adder circuit used in Protocol 5 satisfies Definition 1.*

Proof. (Sketch)

For inputs (x, y) such that $x = \{x_0, \ldots, x_{m-1}\}$ and $y = \{y_0, \ldots, y_{m-1}\}$, we compute the carry with the following equation:

$$c_{i+1} = c_i \oplus ((x_i \oplus c_i) \wedge (y_i \oplus c_i)), \forall i \in \{0, m-1\} \tag{5}$$

Considering the case of the carry computation at position i, if the circuit is tampered, then there is a bad triple at this position. If we can show that with this kind of tampering, the circuit has one of the properties from Definition 1, then the circuit is weakly additively tamper resilient.

Since a bad triple produces an additive error regardless of the inputs used in the carry computation, the output of the circuit $\mathcal{C}$ and $\mathcal{C}^*$ are always going to be different. More specifically, with equation 5, the following property holds: $\mathcal{C}(x, y) = \mathcal{C}^*(x, y) \oplus 1$. Therefore, the ripple carry adder circuit is weakly additively tamper resilient. □

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 the case of generating edaBits over $\mathbb{Z}_p$, we have the restriction that $\log(p) > m$ when using the adder circuit mentioned above, as explained in detail in Section 3.2. To be able to remove this restriction, we need to use Algebraic Manipulation Detection (AMD) [23, 24] circuits, which also satisfy much stronger requirements than being weakly additively tamper resilient.

We discuss the difficulties with analyzing the RealGame directly and describe the simplified game. This is followed by arguing that the SimpleGame is only easier for the adversary to win, which means if it can win in the SimpleGame, it can win in the RealGame.

Complexities of the RealGame*:* In the real game, the adversary can pass the check with a bad edaBit in two different ways. The first is to corrupt edaBit in multiples of the bucket size B, and assume that they all end up in the same bucket so that the error cancels out. The second way is to corrupt a set of 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 can corrupt in order to pass the checks in all the buckets.

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 k-bit, there are up to 2^k 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 while giving the adversary a better chance of winning. However, we show that analyzing the SimpleGame is sufficient for our purpose.

4.3 The SimpleGame

In this section we analyze a simplified game and show that the advantage of the adversary $\mathcal{A}$ to win that game is negligible in s. Before we start explaining the simple game we go out of 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 (●). A edaBits is *BAD* when at least one of the edaBit inside that edaBits is 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.

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

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

```
Simple BucketCheck

Input: $N$ buckets and a function $f$. Each bucket contains $B$ balls $\{x_1, \ldots, x_B\}$
and $(B-1)$ triangles $\{y_1, \ldots, y_{B-1}\}$.
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

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 g (described in Section 4.2) from the real world and the mapping is public. The BucketCheck procedure uses f to check all the buckets. Figure 8 illustrates the check in details. $\mathcal{A}$ passes BucketCheck if all the check configurations are favorable to the adversary. Table 1 shows favorable check configurations for the adversary. After throwing triangles, in each bucket, if the check configuration of

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

Table 1. Favorable combination of balls and triangles for the adversary.

balls and triangles are from the first three entries of Table 1, 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 1 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 guess favorable number of required bad triangles. Thus, if an adversary cannot win the SimpleGame then it must be more difficult for it to succeed in the RealGame.

Lemma 3. *Security against all adversaries in* SimpleGame *implies security against all adversaries in* RealGame*.*

The proof is presented in Section B in the appendix.

We would like to mention that our analysis is not very tight and there is room for improvement. However we will see later that the parameters we obtain from the analysis are sufficient for our purpose. 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.

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 $\alpha \geq \frac{2^{s/3}-1}{2^{s/3}}$ and $C = 3$.[6] Thus,

$$\Pr[C \text{ balls are good}] \approx (1 - \alpha)^C = (2^{-s/3})^3 = 2^{-s}.$$

So if the challenger opens 3 balls to check then in order to pass the first check $\mathcal{A}$ must corrupt less than α fraction of the balls, where $\alpha = \frac{2^{s/3}-1}{2^{s/3}}$. Lemma 4 follows from the above analysis.

Lemma 4. *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 $\alpha = \frac{2^{s/3}-1}{2^{s/3}}$ and the challenger opens $C = 3$ 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

[6] Note that here we implicitly assume $(NB + C) \cdot \alpha < (NB + C)$.

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 $\beta = \frac{2^{s/3}-1}{2^{s/3}}$, we can upper bound the success probability of $\mathcal{A}$ by $(2^{-s/3})^{C'}$. Thus for $C' = 3$ the success probability of $\mathcal{A}$ in the second check can be bounded by 2^{-s}. Lemma 5 follows from the above analysis.

Lemma 5. *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 balls for $\beta = \frac{2^{s/3}-1}{2^{s/3}}$ and the challenger opens $C' = 3$ balls.*

If we consider both the checks together then the probability that $\mathcal{A}$ passes first two checks is bounded by:

$$\Pr[C \text{ balls are good}] \cdot \Pr[C' \text{ triangles are good}] \leq (2^{-s/3})^{C} \cdot (2^{-s/3})^{C'}.$$

If we set the sum of C and C' to be equal to 3, then the probability is less than $(2^{-s/3})^3 = 2^{-s}$. However, later in the analysis we show that the adversary might make the number of corrupted balls to be very small, while increasing the number of corrupted triangles (or vice versa) to make the total probability of passing in the first two checks to be greater than 2^{-s}. Though in that case it might be possible to catch the adversary if we consider the BucketCheck along with first two checks. Still in our analysis we set C and C' to be equal to 3 to make it simple for all boundary cases.

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 1 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'll 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 $(2^{s/3}-1)/2^{s/3} \geq \alpha \geq 1/NB$, 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 $(2^{s/3}-1)/2^{s/3}$ if the challenger opens 3 balls and 3 triangles.

Now let us try to bound $\Pr[\mathcal{A}$ passes 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. These will be discussed in Section B.1 in the appendix.

Lemma 6. *The probability of $\mathcal{A}$ passing the* BucketCheck *in* SimpleGame *is less than 2^{-s}, if $N(B-1) \geq 2^{s/2+1}$ and the challenger opens $C = 3$ balls and $C' = 3$ triangles during first two checks of* SimpleGame.

Proof. The lemma follows from the case-by-case analysis in Appendix B.1, Lemma 4 and Lemma 5. □

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

Proof. The proof follows directly from Lemma 4, Lemma 5 and Lemma 6. $\square$

Remark 1. As we already mentioned the bound we obtain is not sharp. However, for $s = 40$ and $N \geq 2^{20}$, which is sufficient for the applications we are considering in this work, it is enough to set the bucket size to be 3. We leave it as an open problem to improve the bound in the general case. However, we would like to point out that we do not require N to be large for efficient amortization purposes, but instead it is only to comply with our conservative security analysis. Producing N edaBits for a much smaller N, which may be needed for some applications, can still be done efficiently using our methods, but it requires a tighter security analysis.

Finally, we notice that Theorem 2 follows from Theorem 4 and Lemma 3, which concludes our security analysis.

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 [11] and [15] 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 $a = \mathsf{Rep}(\alpha)$. 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 q 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 $y = \frac{a-(a \bmod 2^m)}{(2^m)^{-1}}$.

We focus below in truncation over $\mathbb{Z}_{2^k}$ as it is the less studied case. For the case of truncation over $\mathbb{F}_p$ we refer the reader to Section C in the appendix.

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

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

this procedure will yield shares of

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

Then, to compute the arithmetic shift, we use the fact that[7]

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

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 $(a_{k-1},\dots,a_m,0,\dots,0)$. 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.

[7] 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

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 [29], 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.

Here 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, 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. [14]. Below, we provide an analysis for our extension that also applies to said protocol.

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 $\mathsf{edaBits}$

Now we analyze the protocol. First we notice that $c = 2^{k-\ell-1}c'$ where $c' = (2^m r + r') + a + 2^\ell b - 2^{\ell+1} vb$, 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 $\lfloor (c' \bmod 2^\ell)/2^m \rfloor - r + 2^{\ell-m} = \lfloor a/2^m \rfloor + u$.

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.

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. [11, 15]), 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.

6 Applications and Benchmarks

We have implemented 63-bit[8] comparison using edaBits, only daBits, and neither, and we have run one million comparisons in parallel 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 2 shows the throughput for various security models and computation domains, and Table 3 does so for communication. For computation modulo a prime with dishonest majority, we present figures for arithmetic computation both using oblivious transfer (OT) and LWE-based semi-homomorphic encryption (HE). 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 will be added to MP-SPDZ [13].

		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	14.2	241.5
		p (OT)	1.6	3.3	75.4
		p (HE)	5.9	12.3	141.5
Honest maj.	Malicious	2^k	76.4	109.6	107.1
		p	66.9	71.3	46.2
	Semi-hon.	2^k	500.6	1007.7	1607.6
		p	157.8	277.1	457.6

Table 2. Number of comparisons (in 1000s) per second in various settings

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 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

[8] 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.

		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	813.9	13.5
		p (OT)	7353.1	3487.2	103.7
		p (HE)	411.6	202.8	7.5
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

Table 3. Communication per comparison (in kbit) in various settings

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.

6.1 Comparison to Previous Works

Dishonest majority. The authors of HyCC [9] report figures for biometric matching with semi-honest two-party computation in ABY [20] 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 fair comparison, we have run our implementation using one thread limiting the bandwidth and latency accordingly. Table 4 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. Our approach is not directly comparable to the one by Mohassel and Rindal [29] 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. Section D.2 shows a comparison of their results with our approach applied to logistic regression.

daBits. Aly et al. [1] 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

		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	5.2
$n = 4096$	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	21.1
$n = 13684$	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	84.4

Table 4. Overall time and communication for biometric matching

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 considered be Dalskov et al. [14]. See Section D.1 for details.

Bibliography

[1] A. Aly, E. Orsini, D. Rotaru, N. P. Smart, and T. 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`.

[2] T. Araki, A. Barak, J. Furukawa, M. Keller, Y. Lindell, K. Ohara, and H. Tsuchida. Generalizing the SPDZ compiler for other protocols. In D. Lie, M. Mannan, M. Backes, and X. Wang, editors, *ACM CCS 2018*, pages 880–895. ACM Press, Oct. 2018.

[3] D. Bogdanov, S. Laur, and J. Willemson. Sharemind: A framework for fast privacy-preserving computations. In S. Jajodia and J. López, editors, *ESORICS 2008*, volume 5283 of *LNCS*, pages 192–206. Springer, Heidelberg, Oct. 2008.

[4] E. Boyle, G. Couteau, N. Gilboa, Y. Ishai, L. Kohl, P. Rindal, and P. Scholl. Efficient two-round OT extension and silent non-interactive secure computation. In L. Cavallaro, J. Kinder, X. Wang, and J. Katz, editors, *ACM CCS 2019*, pages 291–308. ACM Press, Nov. 2019.

[5] E. Boyle, G. Couteau, N. Gilboa, Y. Ishai, L. Kohl, and P. Scholl. Efficient pseudorandom correlation generators: Silent OT extension and more. In A. Boldyreva and D. Micciancio, editors, *CRYPTO 2019, Part III*, volume 11694 of *LNCS*, pages 489–518. Springer, Heidelberg, Aug. 2019.

[6] E. Boyle, N. Gilboa, and Y. Ishai. Function secret sharing. In E. Oswald and M. Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 337–367. Springer, Heidelberg, Apr. 2015.

[7] E. Boyle, N. Gilboa, and Y. Ishai. Secure computation with preprocessing via function secret sharing. In D. Hofheinz and A. Rosen, editors, *TCC 2019, Part I*, volume 11891 of *LNCS*, pages 341–371. Springer, Heidelberg, Dec. 2019.

[8] S. S. Burra, E. Larraia, J. B. Nielsen, P. S. Nordholt, C. Orlandi, E. Orsini, P. Scholl, and N. 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`.

[9] N. Büscher, D. Demmler, S. Katzenbeisser, D. Kretzmer, and T. Schneider. HyCC: Compilation of hybrid protocols for practical secure computation. In D. Lie, M. Mannan, M. Backes, and X. Wang, editors, *ACM CCS 2018*, pages 847–861. ACM Press, Oct. 2018.

[10] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, Oct. 2001.

[11] O. Catrina and S. de Hoogh. Improved primitives for secure multiparty integer computation. In J. A. Garay and R. D. Prisco, editors, *SCN 10*, volume 6280 of *LNCS*, pages 182–199. Springer, Heidelberg, Sept. 2010.

[12] R. Cramer, I. Damgård, D. Escudero, P. Scholl, and C. Xing. SPD $\mathbb{Z}_{2^k}$: Efficient MPC mod 2^k for dishonest majority. In H. Shacham and A. Boldyreva, editors, *CRYPTO 2018, Part II*, volume 10992 of *LNCS*, pages 769–798. Springer, Heidelberg, Aug. 2018.

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

[14] A. Dalskov, D. Escudero, and M. Keller. Secure evaluation of quantized neural networks. Cryptology ePrint Archive, Report 2019/131, 2019. `https://eprint.iacr.org/2019/131`.

[15] I. Damgård, D. Escudero, T. K. Frederiksen, M. Keller, P. Scholl, and N. 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.

[16] I. Damgård, M. Fitzi, E. Kiltz, J. B. Nielsen, and T. Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In S. Halevi and T. Rabin, editors, *TCC 2006*, volume 3876 of *LNCS*, pages 285–304. Springer, Heidelberg, Mar. 2006.

[17] I. Damgård, M. Keller, E. Larraia, V. Pastro, P. Scholl, and N. P. Smart. Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In J. Crampton, S. Jajodia, and K. Mayes, editors, *ESORICS 2013*, volume 8134 of *LNCS*, pages 1–18. Springer, Heidelberg, Sept. 2013.

[18] I. Damgård and J. B. Nielsen. Scalable and unconditionally secure multiparty computation. In A. Menezes, editor, *CRYPTO 2007*, volume 4622 of *LNCS*, pages 572–590. Springer, Heidelberg, Aug. 2007.

[19] I. Damgård, V. Pastro, N. P. Smart, and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. In R. Safavi-Naini and R. Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 643–662. Springer, Heidelberg, Aug. 2012.

[20] D. Demmler, T. Schneider, and M. Zohner. ABY - A framework for efficient mixed-protocol secure two-party computation. In *NDSS 2015*. The Internet Society, Feb. 2015.

[21] T. K. Frederiksen, M. Keller, E. Orsini, and P. Scholl. A unified approach to MPC with preprocessing using OT. In T. Iwata and J. H. Cheon, editors, *ASIACRYPT 2015, Part I*, volume 9452 of *LNCS*, pages 711–735. Springer, Heidelberg, Nov. / Dec. 2015.

[22] J. Furukawa, Y. Lindell, A. Nof, and O. Weinstein. High-throughput secure three-party computation for malicious adversaries and an honest majority. In J. Coron and J. B. Nielsen, editors, *EUROCRYPT 2017, Part II*, volume 10211 of *LNCS*, pages 225–255. Springer, Heidelberg, Apr. / May 2017.

[23] D. Genkin, Y. Ishai, M. Prabhakaran, A. Sahai, and E. Tromer. Circuits resilient to additive attacks with applications to secure computation. In D. B. Shmoys, editor, *46th ACM STOC*, pages 495–504. ACM Press, May / June 2014.

[24] D. Genkin, Y. Ishai, and M. Weiss. Binary AMD circuits from secure multiparty computation. In M. Hirt and A. D. Smith, editors, *TCC 2016-B, Part I*, volume 9985 of *LNCS*, pages 336–366. Springer, Heidelberg, Oct. / Nov. 2016.

[25] C. Hazay, P. Scholl, and E. Soria-Vazquez. Low cost constant round MPC combining BMR and oblivious transfer. In T. Takagi and T. Peyrin, editors, *ASIACRYPT 2017, Part I*, volume 10624 of *LNCS*, pages 598–628. Springer, Heidelberg, Dec. 2017.

[26] Y. Ishai, J. Kilian, K. Nissim, and E. Petrank. Extending oblivious transfers efficiently. In D. Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 145–161. Springer, Heidelberg, Aug. 2003.

[27] M. Ishaq, A. L. Milanova, and V. Zikas. Efficient MPC via program analysis: A framework for efficient optimal mixing. In L. Cavallaro, J. Kinder, X. Wang, and J. Katz, editors, *ACM CCS 2019*, pages 1539–1556. ACM Press, Nov. 2019.

[28] M. Keller, E. Orsini, and P. Scholl. MASCOT: Faster malicious arithmetic secure computation with oblivious transfer. In E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, editors, *ACM CCS 2016*, pages 830–842. ACM Press, Oct. 2016.

[29] P. Mohassel and P. Rindal. ABY^3: A mixed protocol framework for machine learning. In D. Lie, M. Mannan, M. Backes, and X. Wang, editors, *ACM CCS 2018*, pages 35–52. ACM Press, Oct. 2018.

[30] P. Mohassel and P. Rindal. ABY3, 2019. `https://github.com/ladnir/aby3/`.

[31] J. B. Nielsen, P. S. Nordholt, C. Orlandi, and S. S. Burra. A new approach to practical active-secure two-party computation. In R. Safavi-Naini and R. Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 681–700. Springer, Heidelberg, Aug. 2012.

[32] D. Rotaru, N. P. Smart, T. Tanguy, F. Vercauteren, and T. Wood. Actively secure setup for SPDZ. Cryptology ePrint Archive, Report 2019/1300, 2019. `https://eprint.iacr.org/2019/1300`.

[33] D. Rotaru and T. Wood. MArBled circuits: Mixing arithmetic and Boolean circuits with active security. In F. Hao, S. Ruj, and S. Sen Gupta, editors, *INDOCRYPT 2019*, volume 11898 of *LNCS*, pages 227–249. Springer, Heidelberg, Dec. 2019.

[34] X. Wang, S. Ranellucci, and J. Katz. Global-scale secure multiparty computation. In B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, editors, *ACM CCS 2017*, pages 39–56. ACM Press, Oct. / Nov. 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 [1, 15, 32, 33], 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 [33]. 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 [1]. 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 [32]. 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 [32] 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 [15]. 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 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 [1]. 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. 11. 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 $\mathcal{F}_{\mathsf{ABB}}$.

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 $P_1, \ldots, P_{t+1}$) 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 $i = 1, \ldots, t+1$.
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. 11. Protocol to generate supposed daBits in any domain

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. 12. 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. 13

Note that our protocol for SPDZ2k is more general than the one proposed by Damgård et al. [15] because theirs only works if the binary part of $\mathcal{F}_{\mathsf{ABB}}$ is 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. [21] for example has linear cost in that regard while achieving the same security properties.

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. [15].
2. Let b_i denote the additive share of b held by P_i, that is $b = \sum_{i=1}^{n} b_i \bmod 2^k$. 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. 12. 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. 13. Protocol to generate supposed daBits in protocols module 2^k without MAC

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 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.

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. 12 is used.

Fig. 14. Protocol to check classic daBits

Fig. 14 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.

B Missing Proofs from Cut and Choose Analysis

Lemma 7. *(Lemma 3, restated) 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 set of triples to $\mathcal{A}$. $\mathcal{A}$ transforms the edaBits into circles. It randomly permutes the set of triples, batch them to form many TRIPs and transform them into triangles. Clearly, a ball (or triangle) is good or bad depends on whether that come from a good or bad edaBit (or TRIP).

$\mathcal{A}$ 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 TRIPs according to the arrangement of balls and triangles in the SimpleGame. Clearly, the simulation is indistinguishable from a RealGame challenger. This is due to the fact that $\mathcal{A}$ randomly permutes the set of triples before grouping them into TRIPs. Thus from the final distribution of triangles $\mathcal{B}$ cannot distinguish whether it is in the RealGame or in the simulation. Also in the simple game the BucketCheck uses the public function f, which is isomorphic to the function g from the real world. Consequently, if $\mathcal{B}$ wins with non-negligible probability then $\mathcal{A}$ wins the SimpleGame with a non-negligible probability. □

B.1 Case-by-Case Analysis of BucketCheck

Case I $(2 \leq t \leq N(B-1)-2)$**:** Here we are considering the cases when $\mathcal{A}$ chooses number of bad triangles t from the range $[2, N(B-1)-2]$ to maximize its success probability. As discussed earlier the probability reaches its minimum at $\beta \approx 1/2$, in that case:

$$\begin{aligned}\Pr[\mathcal{A} \text{ passes } \mathsf{BucketCheck} | L_i] &= \frac{N(B-1)/2! \cdot N(B-1)/2!}{N(B-1)!} \\ &\approx 2^{-N(B-1)}.\end{aligned}$$

If we consider $N(B-1) \gg s$, then this probability is much less than 2^{-s}. Now if we can show that the probability at $t = 2$ and $t = N(B-1)-2$ can be bounded by 2^{-s} then we can say that for all t within this range the probability is upper bounded by 2^{-s}. In both the cases:

$$\begin{aligned}\Pr[\mathcal{A} \text{ passes } \mathsf{BucketCheck}|L_i] &= \frac{2! \cdot (N(B-1)-2)!}{N(B-1)!} \\ &= \frac{2}{(N(B-1))(N(B-1)-1)}.\end{aligned}$$

If we consider $N(B-1) \geq 2^{s/2+1}$, then the probability can be upper bounded by 2^{-s}. Thus for a given b if the adversary chooses number of bad triangles $t \in [2, N(B-1)-2]$, then:

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

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:

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

Case II ($t > (N(B-1)-2)$)**:** If t is greater than $(N(B-1)-2)$ then the adversary will not be able to pass the first two checks as the challenger opens $C = 3$ balls and $C' = 3$ triangles, assuming N and b is large enough. Thus irrespective of the result of the BucketCheck, the winning probability of $\mathcal{A}$ is bounded by 2^{-s}. The adversary can try to decrease b to increase its success probability for the first check. Still in the second check:

$$\Pr[C' \text{ triangles are good}] = \frac{\binom{N(B-1)+C'-t}{C'}}{\binom{N(B-1)+C'}{C'}} \leq \frac{4}{\binom{N(B-1)+3}{3}} \leq 2^{-s}.$$

Case III ($t < 2$)**:** 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 wins the game. However in that case $\mathcal{A}$ fails with probability 1 due to the first check. If $\mathcal{A}$ corrupts α fraction of NB balls, where $\alpha \geq \frac{2^{s/3}-1}{2^{s/3}}$; Then the success probability of the $\mathcal{A}$ can be bounded by 2^{-s}, if the challenger opens $C = 3$ balls in the first check, given $NB \geq 2^{s/2}$. To pass the first check $\mathcal{A}$ can 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 ($B = 3, N = 3, b = 6, t = 0$) only these three configurations are favorable for the adversary:

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

Note that in this case we can consider the bad balls not to be distinct, as here for all the bad balls f(●, ●, △) returns 1. Let us consider $b = KB$, where $1 \leq K \leq (N-1)$, 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 1 bucket, so that it can pass the first check and hope to hit a favorable configuration in the BucketCheck. However, still in that case the probability is negligible in s. Note that this case is same as the

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 considering the case when K is N, as in that $\mathcal{A}$ passing the first check is $\mathsf{negl}(s)$. For example for ($B = 3, N = 4, t = 1$), 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:

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

Clearly, in the second case the probability of success is more than that of the first case. In the second case the probability reaches its maximum value when $K = 1$; Which is same as that of the maximum success probability of $\mathcal{A}$ in the third case for $K = (N-1)$. Consequently the best strategy of the adversary would be to corrupt minimum number of balls, thus making the failure probability of the first check minimum, and try to achieve the maximum success probability from the BucketCheck. That means $\mathcal{A}$ is in the second case with $K = 1$. Thus,

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

C Truncation over Fields

We begin with a protocol, presented originally by Catrina and de Hoogh [11], 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. 15.

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.

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. 15. Deterministic truncation over fields with share gap

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.

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 de Hoogh [11]. 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. 16. 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.

D More Experimental Results

D.1 Convolutional Neural Networks

Dalskov et al. [14] 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 5. It shows that $\mathsf{edaBits}$

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. 16. Probabilistic truncation over fields.

reduce the communication in all security models as well as the time with a dishonest majority.

D.2 Logistic Regression

Mohassel and Rindal [29] present an implementation of logistic regression in the setting of three parties with one semi-honest corruption. Their software implementation [30] runs all parties on the same host without communication encryption. For a fair comparison, we 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. [14] 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 6 shows that edaBit-based comparison generally comes close to ABY3's bit decomposition.

		Domain		Time (s)	Comm. (GB)
Dish. maj.	Mal.	2^k (OT)	[14]	1264.9	1748.4
			Ours	614.6	576.5
		p (HE)	[14]	1377.8	282.4
			Ours	1033.8	270.4
	S-h.	2^k (OT)	[14]	139.5	199.2
			Ours	24.3	31.0
		p (OT)	[14]	465.1	655.3
			Ours	59.3	75.5
Hon. maj.	Mal.	2^k	[14]	9.5	5.3
			Ours	18.6	2.2
		p	[14]	9.2	8.7
			Ours	35.7	4.2
	S-h.	2^k	[14]	0.9	0.8
			Ours	1.0	0.1
		p	[14]	3.7	3.4
			Ours	3.4	0.3

Table 5. Time and communication for MobileNet inference

Dimension	Batch size	ABY3 [29]	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 6. Iterations per second for logistic regression