Fantastic Four:
Honest-Majority Four-Party Secure Computation With Malicious Security

Anders Dalskov
Aarhus University & Partisia

Daniel Escudero
Aarhus University

Marcel Keller
CSIRO's Data61

Abstract

This work introduces a novel four-party honest-majority MPC protocol with active security that achieves comparable efficiency to equivalent protocols in the same setting, while having a much simpler design and not relying on function-dependent preprocessing. Our initial protocol satisfies security with abort, but we present some extensions to achieve guaranteed output delivery. Unlike previous works, we do not achieve this by delegating the computation to one single party that is identified to be honest, which is likely to hinder the adoption of these technologies as it centralizes sensitive data. Instead, our novel approach guarantees termination of the protocol while ensuring that no single party (honest or corrupt) learns anything beyond the output.

We implement our four-party protocol with abort in the MP-SPDZ framework for multi-party computation and benchmark multiple applications like MNIST classification training and ImageNet inference. Our results show that our four-party protocol performs similarly to an efficient honest-majority three-party protocol that only provides semi-honest/passive security, which suggests that adding a fourth party can be an effective method to achieve active security without harming performance.

Changelog: This version improves accuracy figures, which were impacted by a bug in our software in a previous version. We have also improved Section 7.1.2.

1 Introduction

Secure multi-party computation (MPC) allows a set of parties $P_1, \dots, P_n$, each with their own private input $x_1, \dots, x_n$, to compute a function f such that nothing is revealed except the output. In a nutshell, an MPC protocol ensures that any subset of at most t parties (called the corruption threshold) learns just as much from running the MPC protocol, as if everyone had just provided their inputs to a trusted party who computes f by itself, only returning the output afterwards.

The efficiency of MPC protocols is highly dependent on the size of t relative to n, as well as what is assumed about the behavior of parties. For example, protocols where nothing is assumed except $t < n$—the case where all but 1 party might collude—are less efficient than the case where more strong assumptions like $t < n/2$ or $t < n/3$ are made. In addition, protocols where no assumption is made about the behavior of the t corrupt parties are also less efficient that protocols where the t parties are assumed to behave according to the protocol specification. To make these parameters more concrete, we talk about protocols secure against an *honest majority* when $t < n/2$, as opposed to a *dishonest majority* when $t < n$. Likewise, we call corruptions *active* if the t corrupt parties can behave arbitrarily, and *passive* if the t parties behave honestly, but only try to learn more than they should be allowed to by combining their information.

Protocols with an honest majority that are secure against a passive or malicious adversary have received a great deal of attention lately due to their remarkable efficiency. For example, Araki et al. [5] propose a protocol capable of computing in excess of a billion Boolean gates per second for a malicious adversary corrupting one party out of three, and Dalskov et al. [15] present a similar one which can evaluate large convolutional neural networks for practical image prediction in just a couple of seconds.

It might seem these very efficient protocols come at the cost of a particularly restrictive threat model; after all, we need to argue that no two parties collude. However, one setting that fits very well with this threat model, is the *client/server* model wherein the parties with the inputs do not perform the actual computation, but instead share their inputs towards a small fixed set of computing parties. In a way, the cluster of computing parties act (collectively) as a trusted party for the participants who supply input.

Outsourced Secure Computation. The client/server model is particularly attractive for multiple reasons: First, it allows a large number of parties to participate in a secure computation without the need for these to actually run a heavy computation themselves. Indeed, each party only shows up to provide inputs initially, and then later returns to receive outputs in the end. This particular setting has previously been used in practical applications, for example for auctioning sugar beets [8], for computing wage statistics [28], or for detecting financial fraud [7]. In these scenarios, the client/server model of secure computation allows parties with little knowledge for running the secure computation framework, or who do not posses adequate computing power, to benefit from the added privacy of MPC.

Another area in which the client/server model has seen its use recently, and one we will explore in detail as our use-case, is secure machine learning. *Machine-Learning-as-a-Service* (MLaaS) is a popular service architecture, for which many major companies such as Amazon, Google, or Microsoft all provide their own flavor. Traditional MLaaS suffers from a lack of privacy, however. Parties have to trust the provider with both their inputs and the machine learning model: The former is quite clearly problematic as the inputs quite often involve sensitive data such as text or images. However, the latter also presents an issue as obtaining a good machine learning model often is an expensive process, both in terms of work-hours needed to design the model, as well as computation needed to train it.

Secure computation provides a solution to these privacy concerns, and the client/server model seems to be the obvious candidate for a privacy preserving alternative to the MLaaS model. Not surprisingly perhaps, many recent works have focused on providing protocols that instantiate such a privacy preserving MLaaS architecture [13, 15, 32, 35]. In this application, a collection of model input providers provide their training data (or in case just inference is wanted, the model itself) as secret-shares to the MPC providers. Later, users can query the MPC providers with a secret-sharing of their input in order to run secure inference using the model that was previously trained. Notice that all communication only happens to and from the MPC providers, and that all this communication is secret-shared and thus private.

Robust Outsourced Computation. The model outlined above has the following format: Parties show up to provide inputs, leave, and then return to receive their outputs. This is clearly attractive since, from the clients point of view, there is little difference between a privacy-preserving MLaaS, and a regular one. Although the former is slower, it provides much stronger privacy guarantees. However, the currently fastest MPC protocols only provide security *with abort*. What this means in particular, is that a malicious service provider (or one that is just faulty) will cause the computation to abort without any output being provided, and as a consequence, when the input provider later shows up, they do so in vain.

Enforcing robustness—guaranteeing that the correct output is produced—is therefore a very attractive feature in our setting, and one which has been pointed out and explored in prior work as well [10, 26].

We can consider two kinds of robustness, the first which we will call *traditional robustness* and a second which we call *private robustness*.

Traditional robustness is, as the name implies, robustness as traditionally considered. More precisely, it is a guarantee that the computation always outputs the correct value in the end. This might seem like it is sufficient for our purposes, and indeed, this is the kind of robustness that prior work consider. However, it suffers from a subtle privacy issue that is not captured by standard definitions of security, which has very real implications—in particular in the client/server model. An example will illustrate the issue: Consider four service providers *Alice*, *George*, *Mike* and *Frida* who collectively provide a privacy preserving MLaaS. The security model assumes one of these parties are corrupt (or faulty) and the protocol they run is robust. Suppose that at some point during the protocol execution, Alice sees inconsistent messages from Mike and George, concludes that one of these must be acting maliciously and broadcasts a bit stating so. Now, the computation cannot proceed normally (since Alice does not know whether to use the value she got from George, or the one she got from Mike). However, there is one thing all parties *can* conclude: Frida must be honest. Clearly, the dispute is an issue between Alice, George and Mike, and thus the malicious party must be one of these. Robustness is now quite easy to achieve: Everyone simply sends their shares to Frida who reconstructs the input and finishes the computation in the clear.[1]

The above approach works: Output is guaranteed since Frida receives a values from two honest parties and one corrupt, and so can pick the right values by majority. However, it relies on Frida learning all secrets and so is not private as commonly expected. This is not a problem, formally speaking: Frida is honest and the classical security definition for MPC only cares about protecting the input from the malicious party. On the other hand, in a practical setting this is not viable: Users expect the system to keep their inputs private and they expect that this holds towards *any* of the parties. Moreover, the (honest) providers themselves lose in such a system. Just as the users care about privacy, the same is likely the case for the providers since storing sensitive information securely is highly non-trivial. However, a single faulty machine can now in effect “force” sensitive information onto an honest provider, if that provider ends up being the one tasked with completing the computation.

It is for these reasons that we consider a *private robustness* variant as well.[2] Stated simply, this guarantee also ensures the correct output is produced in the end, but it does so *without* relying on a honest party learning the user’s private inputs. We show how this can be achieved by sophisticated protocol

[1]This way of obtaining robustness is essentially how the four-party protocol by Koti et al. [26] works. Their three-party protocol also provides robustness, albeit the process is more involved. The core trick, however, is the same, i.e., a non-malicious party is identified who learns all the secrets in order to finish the computation in the clear.

[2]This issue received a more formal treatment in [2]. Our model differs slightly from theirs, a point which we discuss in detail in Section 6.

transformations between four- and three-party protocols.

1.1 Contributions

This paper makes several contributions towards *practically* efficient secure computation that work particular well for machine learning tasks and outsourced computation. More precisely:

- We present an actively secure four-party protocol for one corruption over $\mathbb{Z}_{2^k}$. This protocol has the same overall complexity of current state-of-the-art protocols in the same setting, but does not require any preprocessing beyond constant-cost setup, unlike previous works. These [22,26] require function-dependent preprocessing, which makes their implement more involved because state of the size for the whole computation has to be stored between the two phases. On the other hand, our approach allows discarding intermediate information as soon as it is no more required for further computation.

- We also present an actively secure three-party protocol for one corruption over $\mathbb{Z}_{2^k}$. While this protocol has slightly higher communication complexity than the online phase of current state of the art [32], the overall complexity is several orders of magnitude lower. This is because, unlike said work, our protocol features a dot product where the communication is independent of the length of the inputs. Furthermore, BLAZE and other similar works rely on the interpolation-based check from [9], which require large extensions of $\mathbb{Z}_{2^k}$ and are not likely to be efficient, as we argue more thoroughly in Section 5.

- We have benchmarked our protocol by training on the MNIST dataset. To the best of our knowledge, we are the first to produce extensive accuracy results for a pure implementation in multi-party computation.[3] Furthermore, we consider the impact of parameter choices such a fixed-point precision on the accuracy of the training.

- Our four-party protocol is robust in the traditional sense: Should an error be encountered during computation, then a trusted party can be identified who can be asked to complete the computation.

- Finally, we show how our four-party protocol can be made robust *while* retaining privacy. Note that robustness as described just above (and as done in prior works) retains no privacy towards the trusted party—this is in many cases undesirable. Our modified four-party protocol is both efficient, robust *and private*.

All our protocols are available as part of MP-SPDZ.[4]

[3] In contrast to running some non-functional computation that is performance-wise equivalent to training.

[4] https://github.com/data61/MP-SPDZ

1.2 Overview of our Techniques

Secure Computation with Three or Four parties. Both our four- and three-party protocols are based on replicated secret-sharing. That is, a value x is shared (in the four-party case) as (x_0, x_1, x_2, x_3) with $x = \sum_{i=0}^{3} x_i$, where P_i holds $\{x_j\}_{j \neq i}$. A similar type of sharing is used for the three-party case. Notice that this is a sharing with threshold one, that is, the share of an individual party does not leak anything about the shared secret x, but two shares together completely determine this value. Furthermore, this sharing is clearly linear which means addition of secrets, as well as multiplications by constants, are just local operations. For multiplication, observe that the product of two secrets can be written as $xy = \sum_{i,j} x_i y_j$. In this sum, a particular term $x_i y_j$ can be computed by all parties not indexed as i or j and so each party is able to obtain part of the sum of xy. This partial sum, in a nutshell, will constitute the share of the product and parties then just have to distribute their partial sums such that everyone in the end has new sharing of the same kind as the original ones (but now of the product).

Active Security. To obtain security against malicious parties, we take different approaches depending on whether we are working with three or four parties.

For four parties, active security is obtained by leveraging the existing redundancy of each share being held by three parties. As a result, the parties can easily distribute the share of the product without the adversary being able to tamper the process.

For three parties on the other hand these ideas do not directly work, given that, although each share is held by two parties, this is not enough to ensure correct behavior when distributing the shares of the product. Instead, we utilize a slight modification of the three-party instantiation of the compiler by Abspoel et al. [1], in which parties hold the shares $([x], [r \cdot x])$ for a random r. When the protocol is run, each gate is evaluated twice: once with the real values and once in a randomized fashion because of the r. At the end of the computation, a check is run to verify that the real output matches the output after being randomized by r. Active security now follows since the adversary can only introduce additive errors (as observed by Genkin et al. [21]) and thus it cannot with high probability "undo" the randomization that is introduced by r.

Mixed-Circuit Computation. For some functionality such as comparisons we switch from arithmetic to binary circuits. This is done via a local share conversion method proposed by Mohassel et al. [30] and Barak et al. [4]. It works by creating a bit-wise sharing of *every* summand of the replicated secret sharing, which are then summed up using a binary circuit. We call the local conversion *share splitting*.

With four parties, active security for binary circuits is provided using the same way as for arithmetic circuits. With three parties however, we use the protocol by Araki et al. [5] because the randomization method above cannot be used efficiently with binary circuits.

Robust Computation. Our way of obtaining robust computation builds on the following clever observation and protocol transformations that do not incur any additional overhead.

Consider the example from above and continue from the situation in which Frida caused an error in the protocol. Like in prior work, we identify a pair of parties (say, Frida and George) of which one is guaranteed to be the culprit. However, instead of letting Alice finish the computation without privacy (as was done before), the parties instead arbitrarily exclude one of Frida or George from participating further. Suppose George is barred. After this step, the remaining parties locally transform all their shares from 1-out-of-4 into 1-out-of-3 sharings, after which they continue with the computation using our maliciously secure three-party protocol.

From this point on, the parties know that, should another error occur, then *for sure* Frida was the malicious party. Indeed, the pair (Frida, George) contained one malicious party and so, after excluding George, we are left with either all honest parties (in which case the computation clearly finishes) or Frida was the malicious party. Should another error occur at this point, Frida can be excluded and the remaining two parties can finish the computation using a passively secure protocol.[5] This series of steps never reveal the private information that is being computed on, but still allows the computation to finish and so provides robustness.

1.3 Related Work

The particular area of MPC with a small number of server, an honest majority and passive or active corruptions have been particularly rich. Araki et al. [6] demonstrate such a protocol (three parties, one corruption, passive security) which can compute 7 billion gates per second. The authors further demonstrate how their protocol can be used to handle up to 35,000 logins per second in a Kerberos system. Chida et al. [14] present an active-to-passive compiler which, as a particular instantiation, contains a very efficient three-party protocol. These works deal with computation over a finite field (the former being $\mathbb{F}_2$, the latter $\mathbb{F}_p$). Secure computation over a finite ring have also been shown to be highly efficient, and it has been shown that it is possible to perform several million multiplications per second [1, 18].

More recently, these smaller-number-of-parties-honest-majority protocols have been shown to be particularly attractive in the setting of privacy preserving machine learning. A series of works [10, 12, 13, 22, 26, 32] demonstrate variants of three- or four-party protocols, all with one corruption, that work particularly well for machine learning when compared against ABY3 [30] which is itself another efficient three-party protocol. Some of these protocols provide either fairness or robustness, however they all suffer from the issues related to these properties that we outlined before.

In more concrete terms, Wagh et al. have shown that both inference and training of large convolutional neural networks is possible with passive [35] and active [36] security.

Building in part on the work by Wagh et al., the authors of CrypTFlow [27] demonstrate that inference with CNNs that are used in practice (such as ResNet or DenseNet networks) is possible in just a few seconds even with active security. Dalskov et al. [15] also perform such experiments and present a protocol which outperforms that of CrypTFlow. Moreover, they also present an insight into the trade-offs when considering honest majority vs. dishonest majority, as well as passive vs. active security.

1.4 Outline

In Section 2 we present our main protocol with abort, including the underlying secret-sharing scheme, multiplication and useful sub-protocols. Then, in Section 3 we present protocols used both with three- and four-party computation such as truncation. This includes an optimized construction of edaBits, introduced by Escudero et al. [19], in our specific context. Section 4 highlights our adaption of the three-party protocol by Abspoel et al. [1]. Following this, we present in Section 6 the extensions of our protocol to achieve robustness, without relying on one single (identified-to-be-honest) party to finish the computation in the clear. Finally, we discuss the implementation of our protocol with abort in Section 7, as well as the applications we consider in our work, namely MNIST classification training and ImageNet inference.

2 Secure Computation Protocol

In this section we present our main protocol with abort. We begin in Section 2.1 by presenting the secret-sharing scheme construction we use in our work. Then, in Section 2.2, we describe our joint message passing protocol, which is used as a primitive for multiple protocols throughout this work. In particular, this primitive is used in Sections 2.3 and 2.4 to obtain protocols for input provision and multiplication, respectively.

2.1 Secret-Sharing

When working in the honest majority setting there are multiple linear secret-sharing schemes one can use. For instance, a popular choice that works well for an arbitrary number of parties is Shamir secret-sharing. One can also obtain linear

[5] Furthermore, the parties can reintroduce George as a cryptographic provider who will output multiplication triples in order to finish the computation.

secret-sharing schemes from linear error-correcting codes. However, when working with a small number of parties, such as three or four, less general but more efficient schemes exist. In this work, we will rely on replicated secret-sharing. While replicated secret-sharing is generic in the sense that it works for any Q2 adversarial structure and any number of parties, it is only concretely efficient for a small number of parties since the complexity scales exponentially with the number of parties if the threshold is a constant fraction thereof. Replicated secret-sharing underlines many recent efficient protocols [1,5,6,15,20]. However, it has been used mostly in the context of three parties. In this case, a value x is distributed among three parties P_0,P_1,P_2 by giving (x_{i-1},x_{i+1}) to P_i, where $x = x_1+x_2+x_3$, and the indexes wrap around modulo 3. For four parties, replicated secret-sharing has been explored somewhat less [10,13,22], sometimes seemingly but not stating so explicitly [10].

The following presentation assumes four parties and one corruption. None of our protocols require special properties of e.g., fields and so we will let $\mathcal{R}$ denote the algebraic structure we would use, which could be computing modulo any number. Using 2 as the modulus implies binary circuits whereas computing with larger moduli is commonly called arithmetic circuits. Using a power of two is particularly efficient due to the binary nature of most processors in use.

To secret-share a value $s \in \mathcal{R}$ with replicated secret-sharing for four parties, the dealer does as shown in Protocol 1.

Protocol 1: Replicated secret-sharing

Dealer distributes $s \in \mathcal{R}$ as follows:

1. Sample s_1,s_2,s_3 from $\mathcal{R}$ uniformly at random and set $s_4 = s-(s_1+s_2+s_3)$.
2. To each P_i for $i = 1,2,3,4$, send $\{s_j\}_{j\neq i}$.

By $[s]$ we mean that each party holds the three values as defined above. It should be clear that $[s]$ defines a linear secret-sharing of s with threshold one: s can be recovered from any two shares, and given shares $[x]$ and $[y]$, a share of $[x+y]$ can be computed by letting each party add the components of their shares. This is denoted by $[x+y] \leftarrow [x]+[y]$. Sometimes, when $\mathcal{R}$ is the set of integers modulo some integer M, we use the notation $[x]_M$. In general we use $M = 2^k$, and when clear from context we omit this from the sharing notation.

As the name implies, replicated secret sharing comes with some redundancy. This enables simple and efficient protocols. In the following we describe the core primitives we use in our work. The first one, described in Section 2.2, is the joint message passing protocol that enables a pair of parties knowing a common value to disseminate it to another party correctly. This primitive is used then to obtain a protocol by which the parties can obtain consistent sharings of an input value. Finally, these subprotocols are put together to obtain an efficient multiplication protocol in Section 2.4, followed by a probabilistic truncation protocol in Section 2.5. For the rest of this section, we denote parties with mutually distinct indices $i,j,g,h \in \{1,2,3,4\}$.

The protocols below admit a *cheating identification* phase, that is executed in case an abort signal is produced, and is in charge of outputting a set of at most two parties such that one of them is corrupted. If one is only interested in security with abort, this phase is not needed. However, we will make use of it in Section 6 when we explore our robust protocols.

2.2 Joint Message Passing

Similar to Koti et al. [26], we make use of a protocol that enables a pair of parties knowing a common value to send this element to another party. Protocol 2 is simple: One of the parties send the value and the other sends a hash, and the receiver compares the received value with the hash. Koti et al. aim to identify an honest party who can act as a trusted party and carry the computation in the clear. Instead, as mentioned before, the only requirement of our protocols is that, if an abort signal is generated, then a pair of identified parties where one of them is corrupt is produced.

Security of JMP. It should be clear that an honest P_g either receives *the correct* x or they abort, unless with negligible probability. Suppose that P_i is malicious (note that P_h never participates and that P_j only sends a hash; in particular, and incorrect x could only come from P_i). If P_i manages to send an $x' \neq x$ such that P_g does not output `err`, then it must be the case that $H(x') = H(x)$ for $x \neq x'$.

Regarding cheating identification, we argue by cases.

- If P_h is the corrupt party, then no abort signal will be produced
- If P_g is the corrupt party, then P_g may accuse P_i and P_j unrightfully. If $c_i = c_j$ then the parties output $\{P_g\}$, which is correct. Else, since P_i and P_j are both honest, it cannot be the case that $c_i \neq c_j$, so either P_i or P_j will accuse P_g, and either case a set containing P_g is output.
- If P_i is the corrupt party, then this party may send an incorrect value to P_g. However, P_g, being honest, will accuse P_i and P_j, and only P_i may return accusation since P_g will broadcast the correct value that P_j sent. If P_i accuses, then $\{P_i,P_g\}$ is output, else, $\{P_i,P_j\}$ is output. Either case, P_i, the corrupt party, appears in the set. A similar argument follows if P_j is corrupt.

We return in Section 6 to how the cheater identification extension can be used to obtain a protocol with privacy preserving robustness.

Protocol 2: $\mathsf{JMP}(x, P_i, P_j, P_g)$**, Joint message passing**

Input: x known to P_i and P_j.
Output: P_g learns x.
Protocol: P_i sends x to P_g.
Batch check:
Let H be a collision resistant hash function. P_j sends $c = H(x, \dots)$ to P_g, who checks if c is consistent with the value sent earlier by P_i. If c is not consistent, P_g outputs a distinguished error symbol `err`.

Cheating identification
If P_g outputs `err`, then the parties proceed as follows to agree on a set of parties with at most two parties that includes the corrupt one.

1. P_g broadcasts $(\mathsf{accuse}, P_i, P_j, c_i, c_j)$, where $c_i = H(x, \dots)$, with $(x, \dots)$ and c_j being the values received from P_i and P_j, respectively.
2. If $c_i = c_j$ then the parties output the set $\{P_g\}$. Else:
   - If c_i is different to the hash of the values that P_i sent to P_g, then P_i broadcasts (accuse, P_g) and the parties output the set $\{P_i, P_g\}$.
   - If c_j is different to the values that P_j sent to P_g, then P_j broadcasts (accuse, P_g) and the parties output the set $\{P_j, P_g\}$.
   - If both parties P_i and P_j accuse P_g, then the parties output $\{P_g\}$.
   - If none of P_i or P_j accuse, then the parties output $\{P_i, P_j\}$.

2.3 Shared Input

We now show how two parties, P_i and P_j, both holding a value $x \in \mathcal{R}$, can secret-share x towards all parties in a manner that is maliciously secure. Protocol 3, PRG_h denotes a pseudorandom generator using key K which outputs a random value $v \in \mathcal{R}$. (We view PRG_h as a stateful probabilistic algorithm; i.e., multiple successive calls return different random elements of $\mathcal{R}$.)

Non-interactive sharing If a value x is known to three parties, say P_1, P_2, P_3, rather than only two, then the parties can get shares $[x]$ without any interaction. This is achieved by defining the additive shares $x_1 = x_2 = x_3 = 0$ and $x_4 = x$. We denote this local method by $[x] \leftarrow \mathsf{INPLocal}(x, P_i, P_j, P_h)$, where P_i, P_j and P_h are the parties knowing the value x.

Protocol 3: $\mathsf{INP}(x, P_i, P_j)$**, Shared Input**

Preprocessing: P_i, P_j, P_h know a pre-shared key K_g for some g, h such that $\{i, j, g, h\} = \{0, 1, 2, 3\}$.
Input: P_i and P_j both know a value x.
Output: $[x]$.
Security: The views of P_g and P_h are independent of x.
Protocol:

1. P_i, P_j and P_h each define $x_g = \mathsf{PRG}_{K_g}()$.
2. Set $x_i = x_j = 0$ and $x_h = x - x_g$.
3. P_i and P_j call $\mathsf{JMP}(x_h, P_i, P_j, P_g)$, so that P_g learns x_h.

Cheating identification
Output the set produced by the JMP protocol.

Security of INP. Note that we just have to argue privacy in case either P_h and P_g are corrupted since P_i and P_j both know x. With respect to P_h, notice that it holds $(x_i, x_j, x_g) = (0, 0, x_g)$ where x_g was uniformly random and generated using PRG_{K_g} and thus is straightforward to simulate given K_g. With respect to P_g, it holds $(x_i, x_j, x_h) = (0, 0, x_h)$ where $x_h = x - x_g$. We simulate P_g's view by randomly choosing x_h. Should x become known to P_g at some point, we can simply compute $x_g = x - x_h$, for which the indistinguishability follows from the privacy of the PRG as K_g is unknown to P_g. Furthermore, all communications happen via JMP, and thus we only need to rely on the cheating identification there.

Composability of INP. Note that the output of the INP protocol does not follow the exact same distribution than that of a trusted dealer, since some of the shares are set to be 0. However, this does not affect security in any way as will become clear in the security proof of MULT.

2.4 Secure Multiplication

Now we present Protocol 4 for secure multiplication, which takes as input two shared values $[x]$ and $[y]$ and produces $[x \cdot y]$.

Security. It is easy to note that the protocol is correct, given that if $x = x_1 + x_2 + x_3 + x_4$ and $y = y_1 + y_2 + y_3 + y_4$, then $x \cdot y = \sum_{i,j=1}^{4} x_i y_j$, so the resulting shares indeed reconstruct to $x \cdot y$. It remains to analyze the privacy of the protocol, to which end we provide the follow simulator that produces the view of a corrupted party P_g. Recall that the output consists of the sum of six instances of INP and four instances of INPLocal. The latter are straightforward to simulate from the inputs because INPLocal is non-interactive, and the former can be simulated

Protocol 4: $\mathsf{MULT}([x],[y])$**, Multiplication**

Input: $[x]$ and $[y]$.
Output: $[x \cdot y]$.
Protocol:

1. For every pair $g,h \in \{1,2,3,4\}$ such that $g < h$, parties P_i and P_j with $i,j \notin \{g,h\}$, who both know x_h, x_g, y_h and y_g, run the protocol $[x_h y_g + x_g y_h] \leftarrow \mathsf{INP}(x_h y_g + x_g y_h, P_i, P_j)$.
2. For every $g \in \{1,2,3,4\}$, parties call the non-interactive method $[x_g y_g] \leftarrow \mathsf{INPLocal}(x_g y_g, P_i, P_j, P_h)$.
3. The parties locally add the shares $[x \cdot y] = \sum_{i \neq j}[x_i y_j + x_j y_i] + \sum_{i=1}^{4}[x_i y_i]$.

Cheating identification
Output the set produced by the JMP protocol.

from the inputs and pre-shared key known to P_g as above. It remains to argue that that the missing share of the product is indistinguishable from a uniformly random value if the product is not revealed. This follows from the same property of at least one of the inputs because that means that least one of the inputs to INP is uniformly random from the view of P_g.

Regarding communication, observe that there are six possible pairs $g,h \in \{1,2,3,4\}$ with $g < h$, and one ring element is communicated in each of the calls to $\mathsf{INP}(x_h y_g + x_g y_h, P_i, P_j)$.[6] As a result, the total asymptotic communication complexity is six ring elements, which is on par with Gordon et al. [22] and Rachuri and Suresh [13], while not requiring any form of preprocessing beyond constant-cost key sharing (in particular, not *function-dependent* preprocessing) as both of them do. While their preprocessing reduces the complexity during the online phase, we argue that this comes at the cost of a much more involved state handling. In our protocol, the only state kept is a running hash value for every combination of players in joint message passing and the state of the pseudo-random generator for shared inputs. Any shares can be discarded as soon as they are not needed any more, however. On the other hand, the protocol by Gordon et al. for example requires information for every wire to be stored between preprocessing, evaluation, and cross-checking.

2.5 Probabilistic Truncation

We have discussed until now the fundamental building blocks to obtain a secure multi-party computation protocol. However, in practice, it is customary to define more advanced subprotocols that can aid in the secure evaluation of a wide variety of functionalities. In this section we discuss a protocol for probabilistic truncation, which is particularly useful when dealing with fixed-point arithmetic, which appears in a lot of scientific applications, within MPC. In a truncation protocol the goal is to obtain shares $[y]$ from a shared value $[x]$, where $y = \lfloor x/2^\ell \rfloor$ for some publicly known value ℓ, and sometimes it is equally useful to obtain $y = \lceil x/2^\ell \rceil$. In the case of *probabilistic* truncation, we are interested in a good approximation of $\lfloor x/2^\ell \rceil$. More precisely, in this case $y = \lfloor x/2^\ell \rfloor + u$, where $u \in \{0,1\}$. Furthermore, u is "biased towards the right result", which means that u is more likely to be 1 (0) the closer $x/2^\ell$ gets to $\lceil x/2^\ell \rceil$ ($\lfloor x/2^\ell \rfloor$).

Protocol 5 for probabilistic truncation combines the special probabilistic truncation protocol by Dalskov et al. [15] with SWIFT [26]. At a high level, it proceeds by first masking the value to be truncated by a random amount and then opening this result. It turns out that, if we require that the most significant bit (MSB) of the value to be truncated is 0, we can extract useful information about the overflow generated by this masking simply from the MSB of the opened value. This in turn helps us compute the truncation of the input from the truncation of the opened value and that of the masking used. The details can be found in the protocol below.

Now we analyze the security properties of the protocol. First, we observe that privacy is preserved throughout the computation given that the sub-primitives JMP and MULT are private. The only potential leakage comes from the calls to INP. However, this only reveals $c = x + r = x + s_3 + s_4 \bmod 2^k$ to P_3 and P_4, but since s_3 and s_4 are uniformly random and unknown to P_3 and P_4 respectively, the leakage of these calls is zero.

It remains to analyze the correctness of our construction. We begin by observing that $c = x + r - 2^k u$ as integers, where u is the potential overflow bit of adding x and r. Similarly, $(c \bmod 2^{k-1}) = (x \bmod 2^{k-1}) + (r \bmod 2^{k-1}) - 2^{k-1}v$, where v is the potential overflow bit of adding $(x \bmod 2^{k-1})$ and $(r \bmod 2^{k-1})$ modulo 2^{k-1}. Notice that, since x's most significant bit is 0, it holds that $(x \bmod 2^{k-1}) = x$ and also that $u = v \cdot r_{k-1}$. Let $c = 2^{k-1} \cdot c'' + (c \bmod 2^{k-1})$, where $c'' = \lfloor c/2^{k-1} \rfloor$, the expressions above allow us to conclude that

$$\begin{aligned}
2^{k-1} \cdot c'' &= c - (c \bmod 2^{k-1}) \\
&= (x + r - 2^k u) - (x + (r \bmod 2^{k-1}) - 2^{k-1} v) \\
&= 2^{k-1} r_{k-1} + 2^{k-1} v - 2^k \cdot v \cdot r_{k-1} \\
&= 2^{k-1}(r_{k-1} \oplus v),
\end{aligned}$$

where r_{k-1} denotes the most significant bit of r. From the above it follows that $c'' = r_{k-1} \oplus v$, or $v = c'' \oplus r_{k-1}$. This is turn shows that v is equal to b from the protocol.

[6]We ignore the cost of sending the hash in the $\mathsf{INP}(\cdot)$ protocol as its complexity is independent of the total number of calls.

Protocol 5: Probabilistic truncation

Input: $[x]$ with the most significant bit of x being 0.
Preprocessing: Pre-shared key K_i known to all parties except P_i, for each $i = 3,4$.
Output: $[\lfloor x/2^m \rceil]$ rounded probabilistically.
Protocol:

1. Let $s_i = \mathsf{PRG}_{K_i}()$ for $i = 3,4$ and $s_i = 0$ for $i = 0,1$. Let $r = s_3 + s_4$. The parties have shares $[r]$ by defining the ℓ-th share to be $\{s_i\}_{i \neq \ell}$.
2. P_0 and P_1 compute r_{k-1} and $r' = \sum_{i=m}^{k-2} r_i \cdot 2^{i-m}$ for $r = \sum_{i=0}^{k-1} r_i \cdot 2^i$ being the bit decomposition of r. The parties call $[r_{k-1}] \leftarrow \mathsf{INP}(r_{k-1}, P_1, P_2)$ and $[r'] \leftarrow \mathsf{INP}(r', P_1, P_2)$.
3. All parties compute $[c] \leftarrow [x] + [r]$.
4. The parties call $\mathsf{JMP}(c_3 + c_4, P_3)$ and $\mathsf{JMP}(c_3 + c_4, P_4)$, and P_3 and P_4 reconstruct $c = \sum_{i=1}^{4} c_i$.
5. P_3 and P_4 compute $c' \leftarrow \lfloor (c \bmod 2^{k-1})/2^m \rfloor$ and $c'' = \lfloor c/2^{k-1} \rfloor$, and call $[c'] \leftarrow \mathsf{INP}(c', P_3, P_4)$ and $[c''] \leftarrow \mathsf{INP}(c'', P_3, P_4)$.
6. All parties call $[r_{k-1} \cdot c''] \leftarrow \mathsf{MULT}([r_{k-1}], [c''])$ and let $[b] \leftarrow [r_{k-1}] \oplus [c''] = [r_{k-1}] + [c''] - 2 \cdot [r_{k-1} \cdot c'']$.
7. All parties output $[c'] - [r'] + [b] \cdot 2^{k-m-1}$.

Cheating identification
Output the set produced by the first instance of JMP to fail.

Now, $(c \bmod 2^{k-1}) = x + (r \bmod 2^{k-1}) - 2^{k-1}v$, thus

$$\left\lfloor (c \bmod 2^{k-1})/2^m \right\rfloor = \left\lfloor \frac{x + (r \bmod 2^{k-1})}{2^m} \right\rfloor - 2^{k-m-1}v.$$

Furthermore, it holds that $c' = \lfloor (x + (r \bmod 2^{k-1}))/2^m \rfloor = \lfloor x/2^m \rfloor + \lfloor (r \bmod 2^{k-1})/2^m \rfloor + w$, with $w \in \{0,1\}$. Given the above, together with the fact that the r' from the protocol equals $\lfloor (r \bmod 2^{k-1})/2^m \rfloor$, we obtain that the output produced by the protocol is

$$c' - r' + 2^{k-m-1}b = \lfloor x/2^m \rfloor + w.$$

Finally, it is easy to see that $w = 1$ with probability equal to the decimal part of $\frac{x}{2^m}$, which shows that the output is biased towards $\lfloor x/2^m \rceil$.

Communication cost. The protocol invokes INP four times, JMP twice, and MULT once. INP and JMP both require sending one ring elements while MULT requires sending six of them. This results in a cost of twelve ring elements overall. Eight of them are only used to compute $[b]$, namely the multiplication as well as INP with c'' and r_{k-1}. b corresponds to the overflow when adding x and r. Previous works using a similar truncation [26, 30–32] have omitted this because b is zero with overwhelming probability if x has enough leading zeros. In Section 7.2.1 we will discuss under which circumstances it is valid to assume this using a real-word example.

2.6 Random Bit Generation

Random bit generation is a fundamental primitive in multiparty computation. We use it in particular to generate daBits [33]. These are essential to convert from binary to arithmetic secret sharing. As observed by Escudero et al. [19], bits shared additively modulo a power of two can be converted to the same sharing modulo two reducing the shares individually.

Protocol 6 shows that we can rely on splitting the players in two groups. Each group generates a random bit, and the XOR of the two is the output. Every group contains one honest party to check on the other.

Protocol 6: Random bit generation

Preprocessing: (P_0, P_1) and (P_2, P_3) have pre-shared keys K_{01} and K_{23}, respectively.
Output: $[b]$ for random $b \in \{0,1\}$.
Protocol:

1. (P_0, P_1) and (P_2, P_3) use K_{01} and K_{23} to sample $b_{01} = \mathsf{PRG}_{K_{01}}()$ and $b_{23} = \mathsf{PRG}_{K_{23}}()$, respectively.
2. The parties use INP to share them to $[b_{01}]$ and $[b_{23}]$.
3. They run $[b] \leftarrow [b_{01}] + [b_{23}] - 2 \cdot \mathsf{MULT}([b_{01}], [b_{23}])$.

Cheating identification
Output the set produced by the first sub-protocol to fail.

It easy to see that that $b \in \{0,1\}$ if this holds for b_{01} and b_{23}, which in turn is guaranteed by the fact that at least one of each pair is honest. A wrong input is caught by the INP protocol. Since b is computed as the XOR of two random bits, one of which is unknown to any party, it is unknown and uniformly random to the view of any party.

3 Mixed-Circuit Computation

Previous work has established that computing non-linear functions such as comparison and truncation is more efficient in binary computation [30]. This in turn requires to switch between arithmetic and binary computation because arithmetic computation is clearly superior for dot products. There are two ways of achieving this. With certain secret sharing schemes one can exploit their properties [4, 17, 30] for conversion. For general conversion, Rotaru and Wood have established the concept of double-authenticated bits (daBits), secret random bits shared in both computation domains. These can be used as mask for secret values. For example, if x is a bit in the computation and r is a secret random bit, $x \oplus r$ does not reveal information. Therefore, one can open $x \oplus r$ in one computation domain and then compute $x = (x \oplus r) \oplus r$ in the other because r is available in both by construction.

Escudero et al. [19] have recently extended this concept to extended daBits (edaBits), which are random m-bit values shared in both domains for some m. As daBits, they can be preprocessed in an offline phase to be used later on, in an online phase, to efficiently compute a wide range of primitives. On top of introducing the concept of edaBits and their applications to practical MPC, Escudero et al. have shown how to generate edaBits in any security model. The goal of this section is to present more efficient protocols to preprocess edaBits in the context considered in our work, that is, replicated secret sharing modulo a power of two.

The core idea of our construction lies in combining the overflow correction used in edaBit generation with the local share conversion for replicated secret sharing [4, 30], which we call share splitting. Protocol 7 shows the details. We denote the j-th bit of a value x by $x[j]$.

Protocol 7: Share splitting

Input: Shared value $[x]_{2^k}$.
Output: Binary replicated secret sharing of selected bits $\{[x[j]]_2\}_{j \in S}$ for a set of indices S.
Protocol:

1. Let x_1, x_2, x_3, x_4 be the additive shares of x, that is $\sum_{i=1}^{4} x_i = x \bmod 2^k$. Recall that each party P_i holds $\{x_j\}_{j \neq i}$.

2. The parties locally compute shares of the bits $x_i[j]$ for $j = 0, \ldots, k-1$ by calling $[x_i[j]]_2 \leftarrow \mathsf{INPLocal}(x_i[j], \{P_h\}_{h \neq i})$.

3. Given $[x_i[j]]_2$ for all i and j and the fact that $\sum_{i=1}^{4} x_i = x$, the parties can compute $[x[j]]_2$ for all desired $j \in S$ using a binary adder.

Observe that share splitting easily generalizes to n-party replicated secret-sharing. Furthermore, this method provides malicious security if said security is used for the binary adder because the replication carries over.

Protocol 8 shows how to generate edaBits efficiently. An edaBit is made of a secret-shared random value $[r]_{2^k}$, together with binary shares of its bits $\{r[i]\}_{i=0}^{k-1}$. The latter are denoted by $[r]_2$. The protocol assumes a method to convert a shared bit $[b]_2$ to the domain 2^k, which we denote by $[b]_{2^k} \leftarrow [b]_2$. This can be instantiated for example by using *daBits*, which are pairs $([r]_{2^k}, [r]_2)$ where $r \in \{0, 1\}$ is uniformly random (notice this is a particular case of edaBits), by letting the parties open $c \leftarrow [r]_2 + [b]_2$, and then compute $[b]_{2^k} = [r]_{2^k} + c - 2 \cdot c \cdot [r]_{2^k}$.

Protocol 8: edaBits with replicated secret sharing

This protocol assumes that $m \leq k$, that parties $\{P_h\}_{h \neq i}$ have a pre-shared key s_i and that a conversion method $[b]_{2^k} \rightarrow [b]_2$ is available.
Output: $[r]_{2^k}, [r]_2$ for uniform m-bit r.
Protocol:

1. Parties generate a random m-bit value r'_i for every pre-shared key s_i. This leads to a secret shared value $[r']_{2^k}$, where $r' = r'_1 + r'_2 + r'_3 + r'_4$. Notice that r' is in the range $[0, \min(2^{m+2}, 2^k) - 1]$.

2. Using share splitting, the parties compute $[r'[j]]_2$ for $j = m, \ldots, m'$, where $m' = \min(\lceil \log n \rceil, k) - 1$ for n being the number of parties.

3. The parties convert $[r'[j]]_{2^k} \leftarrow [r'[j]]_2$ for $j = m, \ldots, m'$.

4. The parties compute $[r]_{2^k} = [r']_{2^k} - \sum_{j=m}^{m'} 2^j [r'[j]]_{2^k}$.

5. The parties output $([r]_{2^k}, \{[r'[j]]_2\}_{j=0,\ldots,m-1})$.

We remark that in the context of probabilistic truncation with our three-party computation, only the first part of the edaBit is used, it is thus not necessary to compute $\{r'[j]\}_{j=0,\ldots,m-1}$. We provide further details on truncation in Appendix A.

4 Three-Party Computation

Our techniques to achieve robustness rely on three-party computation, which has received considerable attention recently [1, 4–6, 12, 18, 20, 26, 27, 30, 32, 35, 36]. We use a modified version of the three-party instantiation of the compiler by Abspoel et al. [1]. Their protocol consists of adding some authentication data to secret-shared values so that cheating can be detected in way similar to the SPDZ line of protocols [16].

However, their protocol does not allow continuous computation since it involves a final check phase in which correctness is verified. Everything before this check is not trustworthy, and no computation can be done after the check since some secret information that prevented cheating is already revealed. We modify the verification protocol by Abspoel et al. by keeping this secret information hidden to facilitate continuous computation at the cost of one extra secret multiplication in the underlying protocol.

Continuous computation has both conceptual and practical benefits: It allows to keep secret-shared information for longer. For example, one can secret-share the weights of a neural network once, and then use these shares for several individual inference computations that are verified separately. On the practical side, the checking protocol involves keeping information for every multiplication until checking. Continuous computation reduces the storage requirement because it allows for regular checking and thus deletion of the intermediate information. This provides a trade-off between storage requirement and communication.

Protocol 9 outlines the relevant parts of our protocol. We denote by $\langle x\rangle$ the SPDZ-wise sharing of x, that is the tuple $([x]_{2^{k+s}}, [r\cdot x]_{2^{k+s}})$ for a global MAC key $r \in \mathbb{Z}_{2^s}$. We instantiate the zero-check functionality $\mathcal{F}_{\mathsf{CheckZero}}$ using the post-sacrifice protocol by Eerikson et al. [18]. The conversion is straight-forward because said protocol also uses replicated secret sharing modulo a power of two.

Complexity. Recall that the underlying protocol requires every party to send $k+s$ bits per dot product and the inputting party to send $2(k+s)$ bits per input. It follows that the asymptotic cost of a dot product in the SPDZ-wise protocol is $6(k+s)$ and the asymptotic cost of an input is $3(k+s)$ over all parties. Note in particular that the cost of the product is independent of the length.

4.1 Random Bit Generation

For multi-party computation going beyond polynomials such as comparison, bit shifting etc., masking with random bits plays an integral part. Even when using edaBits, we still need daBits to convert from binary secret sharing back arithmetic secret sharing. A straightforward way of generating random bits with semi-honest security against one corrupted party is to simply compute the XOR of random bits input by two different parties. In the malicious setting however one has to mitigate dishonest parties inputting values other than zero or one. An efficient way to do this without revealing anything is to check whether $b\cdot(1-b)=0$. Even if b is in $\mathbb{Z}_{2^k}$, the equality implies that $b\in\{0,1\}$ because either b or $1-b$ is odd and thus not a zero divisor. We use this check for our random bit generation protocol in Figure 4.1. The protocol also uses the fact that the SPDZ-wise protocol provides dot products with constant communication.

Protocol 9: SPDZ-Wise Protocol

Global setup: MAC key $[r]_{2^{k+s}}$

Input: The parties let P_i input z as follows:

1. P_i inputs z to the underlying protocol, resulting in $[z]_{2^{k+s}}$.
2. Compute $[z\cdot r]_{2^{k+s}}$ using $[r]_{2^{k+s}}$ and the underlying protocol.
3. Use $\langle z\rangle = ([z]_{2^{k+s}}, [z\cdot r]_{2^{k+s}})$ for further computation and store it for verification.

Multiplication: The parties compute the dot product of $(\langle x_1\rangle,\dots,\langle x_n\rangle)$ and $(\langle y_1\rangle,\dots,\langle y_n\rangle)$ as follows:

1. Compute the dot products $\sum_i [x_i]_{2^{k+s}}\cdot[y_i]_{2^{k+s}}$ and $\sum_i [x_i]_{2^{k+s}}\cdot[r\cdot y_i]_{2^{k+s}}$ using the underlying protocol.
2. Store the resulting pair as $\langle z\rangle$ and use it for further computation.

Verification: The parties verify all results and inputs $\langle z_1\rangle,\dots,\langle z_n\rangle$ as follows:

1. Generate fresh random values $[r_1]_{2^{k+s}},\dots,[r_n]_{2^{k+s}}$. This can be done using PRSS.
2. Compute the dot products $[u]_{2^{k+s}} \leftarrow \sum_i [r_i]_{2^{k+s}}\cdot[z_i]_{2^{k+s}}$ and $[w]_{2^{k+s}} \leftarrow \sum_i [r_i]_{2^{k+s}}\cdot[z_i\cdot r]_{2^{k+s}}$ using the underlying protocol.
3. Compute $[u]_{2^{k+s}}\cdot[r]_{2^{k+s}} - [w]_{2^{k+s}}$ using the underlying protocol and check it for zero using $\mathcal{F}_{\mathsf{CheckZero}}$ as described by Abspoel et al. [1].

It is clear the final step succeeds if all b_i are zero. If any $b_i \bmod 2^k$ is non-zero however, the final step will fail similarly to the multiplication check because r_i was generated independently of b_i. Furthermore, assume w.l.o.g. that P_0 is honest, and consider that

$$b_i = b_i^0 + b_i^1 - 2\cdot b_i^0\cdot b_i^1 = \begin{cases} b_i^1 & b_i^0 = 0 \\ 1-b_i^1 & b_i^0 = 1. \end{cases}$$

It follows that, independently of b_i^0, $b_i\in\{0,1\}$ if and only if $b_i^1\in\{0,1\}$. This precludes selective failure attacks on b_i^0.

Complexity. The protocol requires two SPDZ-wise inputs and one SPDZ-wise multiplication, resulting in $12(k+s)$ bits overall.

Protocol 10: Random Bit Generation

In the following, P_0 and P_1 are placeholders for any two distinct parties.

1. P_0 and P_1 input $\langle b_1^0\rangle, \dots, \langle b_n^0\rangle$ and $\langle b_1^1\rangle, \dots, \langle b_n^1\rangle$, respectively.
2. The parties compute $\langle b_i\rangle \leftarrow \langle b_i^0\rangle + \langle b_i^1\rangle - 2 \cdot \langle b_i^0\rangle \cdot \langle b_i^1\rangle$ for all i.
3. The parties generate random public values $r_1, \dots, r_n \in \mathbb{Z}_{2^s}$.
4. The parties compute $\sum_i (r_i \langle b_i\rangle) \cdot (1 - \langle b_i\rangle$ and check whether it opens to zero.

5 Communication Complexity

Table 1 compares the communication complexity of our protocols to previous and concurrent work.

Our protocols for three parties perform up to an order of magnitude worse than previous works. However, all of them rely on the verification by Boyle et al. [9], which heavily uses arithmetic in $\mathbb{Z}_{2^k}[X]/(f)$ with f being a polynomial of degree in $[46, 72]$ (according to choices by Boyle et al. and Koti et al. [26]) and irreducible over $\mathbb{F}_2$. We are not aware of a publicly available implementation of the verification protocol and therefore have used a micro-benchmark in order to estimate the computational cost. More concretely, we have implemented multiplication in $R = \mathbb{Z}_{2^{64}}[X]/(f)$ for $f(X) = X^{46} + X + 1$ using the ZEN library [11], and we have found that the throughput is less than 22,000 multiplications per second on a single core of a 2.8 GHz i7 processor. Both Boyle et al. and Koti et al. have suggested to run the verification for batches of at least $m = 2^{20}$ multiplications. Furthermore, Boyle et al. have put the computational cost at $O(m\sqrt{m})$, and our understanding is that means at least $m\sqrt{m}$ multiplications in R. The number of multiplications in R per secure multiplication is therefore $\sqrt{m}$, which comes down to $2^{10} = 1024$ for $m = 2^{20}$. Using our estimates of at most 22,000 R multiplications per second, we conclude that the throughput would only be 22 secure multiplications per second. This pales in comparison to our solution for which we observe an overall computational throughput of more than 400,000 multiplications per second on the same setup as above, and where a single multiplication requires only 624 bits (for 40-bit statistical security). We thus conclude the computational cost for protocols in $Z_{2^{64}}$ is prohibitively expensive unless an efficient implementation of arithmetic in R is found.[7]

For four parties, the difference is at most a factor of roughly 2. We have identified two factors that make up the difference. First, the function-dependent preprocessing of SWIFT allows for a combination of protocol steps that are separate in our case. Second, the authors of SWIFT claim to compute the carry-out of a logarithmic-round parallel-prefix binary adder using $2k$ AND gates. However, we cannot reproduce that. The most common design would use k AND gates in a first step to compute k generate-propagate tuples, followed by tree-wise reduction where every step involves 2 AND gates, resulting in $3k$ AND gates overall.

[7] Boyle et al. [9] have found much more favorable results for computation modulo a prime because there is no need to use an extension ring in that case.

6 Achieving Robustness

We now turn our attention to describing how our four-party protocol can be made robust. As mentioned back in the introduction, we will consider two types of robustness: *traditional* and *private*. Traditional robustness permits an honest party to learn the users private inputs (and is the kind explored in prior works), while private robustness does not allow this.

6.1 Robustness

By relying on the *cheating identification* extension of JMP we immediately get a robust protocol: When a dispute is recorded, parties can point to at least one party which is honest. Indeed, JMP is a protocol between only three of the four total parties and so the party who did not engage in JMP must be honest. Traditional robustness then follows: Parties just send their shares to the recognized honest party who finishes the computation.

6.2 Privacy preserving robustness

We now describe how the *cheating identification* extension of the JMP protocol allows for a very efficient, and more importantly, privacy-preserving, way of obtaining a robust protocol that also guarantees privacy with respect to the views of honest parties. In a nutshell, the idea is to identify a pair of parties $\{P_i, P_j\}$ of which one is malicious. One of these two parties is then excluded and the remaining three parties convert their shares from ones that are compatible with our four party protocol, to some that are compatible with our three party protocol. After this has been done, computation continues, however now the remaining parties know that, should another error occur, then this *must* have originated from the party that was not excluded.

As a starting point, we make the following observation about the secret sharing scheme we employ:

Local Share Conversion. Consider a replicated sharing of x held by our four parties: In more detail, parties hold the following values:

$$P_0 \text{ holds } (x_1, x_2, x_3), \qquad P_1 \text{ holds } (x_0, x_2, x_3),$$

Table 1: Asymptotic global communication of building blocks in k for computation in Z_{2^k} and statistical security parameter s. Furthermore, f denotes the number of relevant bits in fixed-point representation after multiplication, and "big gap"/"small gap" stands for whether $f \leq k-s$ or not. We use "<" to indicate when binary circuit for a k-bit value has complexity slightly than less a multiple of k.

	Three parties			Four parties		
		Prep.	Online		Prep.	Online
(Dot) Product	BGIN19 [9]	-	$3k$	GRW18 [22]	$4k$	$2k$
	SWIFT [26]	$3k$	$3k$	SWIFT [26]	$3k$	$3k$
	Ours / [1]	-	$6(k+s)$	Ours	-	$6k$
(Dot) Product with truncation	SWIFT [26] ("big gap")	$15k$	$3k$	SWIFT [26] ("big gap")	$4k$	$3k$
	Ours (any case)	$76(k+s)+54f+12$	$9k+6s$	Ours ("big gap")	k	$9k$
				Ours ("small gap")	$2k$	$16k$
MSB extraction	SWIFT [26]	$9k$	$9k$	SWIFT [26]	$\approx 7k$	$\approx 7k$
	Ours	-	$< 108k$	Ours	-	$< 30k$
Bit to arithmetic	SWIFT [26]	$9k$	$4k$	SWIFT [26]	$\approx 3k$	$3k$
	Ours	$14(k+s)$	1	Ours	-	$8k+1$

$$P_2 \text{ holds } (x_0, x_1, x_3), \qquad P_3 \text{ holds } (x_0, x_1, x_2),$$

where $x = x_0 + x_1 + x_2 + x_3$. In the case where, say, P_0, is excluded, the remaining parties can locally convert their shares into shares compatible with a three party replicated secret sharing as follows: Parties P_1 and P_2 define $x' = x_0 + x_3$ and P_3 discards x_0. That is, P_1 holds (x', x_2), P_2 holds (x_1, x') and P_3 holds (x_1, x_2). It is easy to see that this still defines a valid secret-sharing of x. If we further exclude a party (say P_3), then the two remaining parties can perform a similar action as before in order to obtain a valid full threshold secret-sharing of x (e.g., P_1 could set their share to be $x' + x_2$ while P_2 sets their share to be x_1).

6.3 Robustness through protocol hopping

Using the observation above, privacy-preserving robustness is now attained in the following way.

Consider first the case where the cheating identification in JMP outputs a single party. This case is easy to handle: parties just stop talking to the cheating party, convert their shares as described above and perform the computation using a semi-honest three-party protocol.

The situation is more interesting if a pair $\{P_i, P_j\}$ is identified. In this case, we proceed as follows:

1. Parties select one of P_i or P_j arbitrarily and stop communicating with that party (e.g., pick the party with the lowest index). To be concrete, suppose P_i is kicked out.

2. All remaining parties convert their shares into three-party sharings. Notice that these can be viewed as a semi-honest sharing of the underlying value, and so in particular, can be used from step 2 onward in Protocol 9.

3. Thus, the three remaining parties continue the computation with the SPDZ-wise protocol.

The computation is clearly still private: the share conversion was local, and the SPDZ-wise protocol is secure against a malicious adversary (observe that P_i might be the honest party and so the malicious party, P_j, could still be participating).

Consider now the situation if another error happens during the execution of the three party protocol: In this case, parties will know for sure that P_j is malicious and that P_i was honest. Indeed, of the pair $\{P_i, P_j\}$ one is guaranteed to be malicious, and if P_i did not participate but malicious behavior was observed, then P_j must be the culprit.

Having identified the malicious party, it is now just a matter of finishing the computation in a privacy-preserving manner. To do so, the remaining two parties (P_g, P_h) convert their shares into full threshold shares and execute a semi-honest two party protocol. Notice that, while such a protocol is quite expensive, it is in our case easy to make very efficient. Indeed, now that we could conclude that P_i was honest, the parties can reestablish a connection with P_i who would then be tasked with producing multiplication triples.

6.4 Discussion on Private Robustness

At this point it makes sense to take a step back and consider the security and practicality of our private robustness protocol.

The issue of revealing private input to honest parties was identified recently in the work by Alon et al. [2]. In that work, the authors present a notion of security called *FaF*-security wherein security should hold against t malicious parties, as well as h honest parties.

The *FaF* notion of security is close to what we seek, although we make a simplifying assumption on the system model that allows us to bypass an issue pointed out in [2]. We outline the system model that our private robustness assumes, and then argue that it provides a satisfying level of security.

System model. An important issue that complicates the model presented in [2], is that the malicious party can send their private data to an honest party. In a nutshell, such an action means that the view of the honest party now contains the information of the malicious party as well. In particular, if the privacy threshold of the secret sharing scheme used is one, then this would it would now be possible to recover the private inputs from the view of the honest party.

We consider a slightly more restrictive model. In particular, honest parties are assumed to only store *intended* messages; that is, messages that were not part of the protocol, or malformed, will not be stored in a way which gives rise to the issue described in [2]. It goes without saying that this *is* a more limited model than the one used in the FaF definition, and so our private robustness protocol does not work if honest parties stores all incoming data for long periods of time.

Security. If honest parties do not store non protocol messages, then our private robustness protocol is secure. More precisely, the protocol hopping we perform will guarantee that (1) correctness is preserved in the presence of the malicious party, *and* (2) that the private inputs cannot be recovered from the honest parties after protocol execution. (1) hinges on the fact that we can always detect when something goes bad. This is a result of redundancy in the secret sharing scheme we employ. When an inconsistency is detected, parties will reduce their sharing from a 1-out-of-n to a 1-out-of-$(n-1)$ sharing, kick out one of the parties that were involved in the dispute and redo the computation. Because this conversion is local, we are guaranteed that the computation does not continue with invalid shares. (2) now follows from the fact that honest parties only store protocol messages, and so it is only needed that our underlying three and four party protocols to not contain steps that instruct honest parties to reveal their inputs.[8] We expect that formally defining private robustness will require significant work, especially considering the system model outlined above, which deviates from the standard ones used in MPC, and so consider this particular direction as future work.

Overhead. The overhead associated with out private robustness protocol depends on how often verification is performed. Recall that in both our three party protocol (Section 4), and our four party protocol (Section 2) verification can be batched, and so we can think of the computation as being divided into segments each of which concludes with a verification step before proceeding to the next segment (this is similar to what is called epochs in prior literature on non-private robustness). If an inconsistency is found, parties perform the steps outlined above and rerun the failed segment. This demonstrates that the overhead is closely related to the size of a segment. In an optimistic setting (where there is only one segment), the overhead is thus at most the sum of execution times of the different protocols.

7 Applications and Implementation

In order to demonstrate the benefit of our protocols, we have implemented various applications in MP-SPDZ [24], which we will present in this section. We have only implemented protocols with abort, however, because we do not see a meaningful way to benchmark robust protocols. Usually, the complexity of these vary considerably depending on the behavior of the corrupt parties.

Our applications use real number arithmetic, which we emulate by using fixed-point representation of fractional numbers, that is, $x \in \mathbb{R}$ is represented as $\lfloor x \cdot 2^{16} \rceil$. After every multiplication we round using probabilistic truncation, as described in Section 2.5.

We use three protocols in our benchmarks, the four-party protocol with abort in Section 2, the three-party protocol with abort in Section 4, and the semi-honest three-party protocol already available in MP-SPDZ. The three-party protocol with abort is based largely on work by Abspoel et al. [1] while the three-party protocol goes back to Araki et al. [6] with optimizations by Dalskov et al. [15] and Eerikson et al. [18].

We consider the following applications. First, we discuss in Section 7.1 the case of multi-class deep learning, where the goal is to learn a label from a non-binary set given some training data using deep neural networks. Then we consider in Section 7.2 training a logistic regression model to learn a binary label. Finally, Section 7.3 shows our results for ImageNet inference using established networks such as ResNet.

7.1 Multi-Class Deep Learning

We have implemented training for the MNIST dataset [29] with one to three dense layers.[9] All but the last layer are followed by a ReLU activation and output 128 values. We used a batch size of 128, resulting in 469 iterations per epoch as there are 60,000 examples in the training set. Furthermore, we use softmax to compute the loss and stochastic gradient descent with a momentum of 0.9 for training. The learning rate is set to 0.01 in the beginning and is halved whenever a reset is necessary due to divergence. To implement the exponential function used by softmax we use the approach by Aly and Smart [3].

Table 2 lists our timings and accuracy results for one run of each protocol on AWS `c5.9xlarge`. We also run the same computation using one of the semi-honest three-party protocols provided by MP-SPDZ. The results show that running malicious four-party computation costs less than twice of

[8] In a way, what we wish to state with the private robustness notion is that the protocol itself should not have an explicit instruction that breaks privacy.

[9] Relevant scripts and a Docker container are available here: https://github.com/csiro-mlai/mnist-mpc

Table 2: Time and accuracy for MNIST with various models and protocols with one corrupted party. "SH 3PC" stands for the semi-honest protocol implemented MP-SPDZ while "Mal. 4PC" and "Mal. 3PC" stand for the protocols with abort presented in this work.

No. dense layers	Seconds per epoch			Accuracy after n epochs			
	SH 3PC	Mal. 4PC	Mal. 3PC	$n = 5$	$n = 10$	$n = 15$	$n = 20$
1	12.2	22.1	92.7	91.7	92.0	92.2	92.3
2	28.2	42.4	451.5	95.8	96.9	97.2	97.6
3	33.8	51.1	573.7	96.8	97.5	97.8	97.9

Table 3: Time and accuracy for MNIST 4/9 distinction with various models and protocols with one corrupted party.

	No. dense layers	Seconds per epoch			Global comm. per epoch (MB)			Accuracy after n epochs			
		SH 3PC	Mal. 4PC	Mal. 3PC	SH 3PC	Mal. 4PC	Mal. 3PC	$n = 5$	$n = 10$	$n = 15$	$n = 20$
SWIFT [26]	1	$\perp$	103.23	143.22	$\perp$	8.8	19.3	$\perp$	$\perp$	$\perp$	$\perp$
Ours	1	0.4	0.6	1.7	27.5	33.3	560.7	96.5	96.4	96.8	96.9
	2	3.7	4.0	78.0	3,617.1	4,269.2	58,100.2	97.2	98.1	98.2	98.3
	3	4.8	5.5	101.9	4,788.4	5,900.3	74,252.5	97.6	98.2	98.7	98.6

running semi-honest three-party computation, both with one corrupted party.

7.2 Logistic Regression and Binary Classification

We have further implemented training to distinguish between "4" and "9" in the MNIST dataset. While this task is inspired by the Gisette dataset [23], we restrict ourselves to the relevant subset of MNIST in order to allow comparison with previous works that used the same number of features as MNIST for logistic regression. As previous works [26, 30, 31], we use a three-part approximation of the sigmoid function.

There are 11791 and 1191 relevant examples in the MNIST training and test set, respectively. This comes down to 93 iterations per epoch with our batch size of 128. We use the same parameters for stochastic gradient descent as above. Table 3 shows our results. The figures for SWIFT [26] therein are based on the reported 1.11 and 1.54 seconds as well as 203.47 and 92.91 KiB global communication per training iteration for malicious 3PC and 4PC training of a 1-layer model, respectively. The authors of SWIFT also report throughput figures of running several iterations in parallel. We do not use those because the training is not parallelizable.

Communication. The authors of SWIFT have confirmed in private communication that their figure does not include probabilistic truncation in the model update (at least in their 4PC protocol). In our protocols, the communication during the model update accounts for about half the total cost. One reason for this is that we use stochastic gradient descent with momentum [34], which requires an additional public-private multiplication. This together with the considerations in Section 5 explains the difference for three-party computation. For four-party computation, we note that we use the more general "small gap" method for probabilistic truncation while SWIFT requires that the difference between number of relevant bits after multiplication and the number of bits of the computation domain is at least the statistical security parameter. In the next section, we explore the limitations of this approach.

7.2.1 The Impact of Fixed-Point Precision

Cleartext training makes use of floating-point arithmetic, typically over 32-bit datatypes, and its accuracy is very well understood. However, when working in MPC floating-point arithmetic, although possible, is considerably more expensive [25], which is why one typically resorts to fixed-point arithmetic as we do in this work. Although this increases efficiency, it is not clear if, and if so, by how much, accuracy is degraded. Previous works largely disregard this issue. In this section we present data that supports experimentally that, while the choice the parameters used in previous works gives reasonably accuracy with small models, this is unlikely to extend to larger models.

We now evaluate the impact of varying the fixed-point precision and the implications on the probabilistic truncation, based on the binary classification example from above. Recall that we present fractional numbers as $\lfloor x \cdot 2^f \rceil$ for $f = 16$. Table 5 shows the impact of varying f between 8 and 20. It also shows the maximum bit length we encountered in truncation. This information is relevant because it dictates how small k can be while reduction modulo 2^k does not affect the computation. In our protocols the bit length of values can be almost the one of the computation domain (k for computing modulo 2^k) without affecting correctness. The only restriction is in our probabilistic truncation protocol from Section 2.5, which requires the most significant bit of the input sharing to be 0.

Unlike our protocols, many previous works [26, 30–32] use a probabilistic truncation that requires this bit length to be much shorter than the bit length of the computation domain. More precisely, in these works the failure probability per operation is $2^{\ell-k}$, where ℓ is the bit length of the input to the probabilistic truncation. We have computed the expected number of failures by adding up the probability for all truncation operations over 20 epochs. Table 5 shows that this number is more than one for 20 bits of precision and two dense layers. This is because said computation requires roughly 2^{29} truncations as one epoch of the training already involves more than 9 million such truncations.[10] We conclude that this limits the use of the cheaper probabilistic truncation with computation modulo 2^{64}, which is commonly used due to the ubiquity of 64-bit processors.

These considerations give rise to a trade-off between the more expensive truncation in Section 2.5 and increasing the computation modulus in order to decrease the error probability with the cheaper truncation. By more expensive we mean the truncation that uses $[b]$ in the last step of Protocol 5 in order to compensate the overflow when masking earlier on. The former triples the communication cost as outlined in Section 2.5 while the latter leads to an overall increase in the computation cost because it is twice as expensive to add 72-bit numbers and more than twice as expensive to multiply them on a 64-bit platform. Table 6 outlines this trade-off for our binary classification task. It shows that, while the more efficient truncation saves about one third in communication throughout, the required larger modulus doubles the time when training the two-layer model.

Koti et al. [26] use 13-bit fixed-point precision with a 64-bit modulus and the more efficient probabilistic truncation. Table 5 shows that this comes with an expected number of failures of roughly 2^{-15} when using two dense layers, which we consider too high for widespread usage.

7.3 ImageNet Inference

In order to compare our protocols to the ones by Dalskov et al. [15] and Kumar et al. [27], we have adapted the implementations by the former. Our results in Table 4 show that adding another honest party is competitive with Kumar et al.'s approach of relying on a trusted execution environment instead. While our approach does increase communication by a factor of 1.5–2, the increase in overall time is at most 1.6-fold and thus less than the 3-fold increase in the TEE-based solution.

Acknowledgments

We would like to thank the anonymous reviewers of USENIX Security as well as the shepherds Tamara Bonaci and Mayank Varia for their helpful insights and comments. The first author was supported by the Danish Independent Research Council under Grant-ID DFF-6108-00169 (FoCC) and did part of their work while a student at Aarhus University. The second author was supported by the European Union Horizon 2020 research and innovation programme under grant agreements No. 669255 (MPCPRO).

[10] There are 93 iterations per epoch and more than $784 \cdot 128$ parameters in the first layer.

References

[1] Mark Abspoel, Anders Dalskov, Daniel Escudero, and Ariel Nof. An efficient passive-to-active compiler for honest-majority MPC over rings. Cryptology ePrint Archive, Report 2019/1298, 2019. https://eprint.iacr.org/2019/1298.

[2] Bar Alon, Eran Omri, and Anat Paskin-Cherniavsky. MPC with friends and foes. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part II*, volume 12171 of *LNCS*, pages 677–706. Springer, Heidelberg, August 2020.

[3] Abdelrahaman Aly and Nigel P. Smart. Benchmarking privacy preserving scientific operations. In Robert H. Deng, Valérie Gauthier-Umaña, Martín Ochoa, and Moti Yung, editors, *ACNS 19*, volume 11464 of *LNCS*, pages 509–529. Springer, Heidelberg, June 2019.

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

[5] Toshinori Araki, Assi Barak, Jun Furukawa, Tamar Lichter, Yehuda Lindell, Ariel Nof, Kazuma Ohara, Adi Watzman, and Or Weinstein. Optimized honest-majority MPC for malicious adversaries - breaking the 1 billion-gate per second barrier. In *2017 IEEE Symposium on Security and Privacy*, pages 843–862. IEEE Computer Society Press, May 2017.

[6] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. High-throughput semi-honest secure three-party computation with an honest majority. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 805–817. ACM Press, October 2016.

[7] Dan Bogdanov, Marko Jõemets, Sander Siim, and Meril Vaht. How the estonian tax and customs board evaluated a tax fraud detection system based on secure multi-party computation. In Rainer Böhme and Tatsuaki Okamoto,

Table 4: Time and communication for various ImageNet models. [27] by [15] denotes the protocol by the former as benchmarked by the latter. "⊥" stands for unavailable data.

	SqueezeNet		ResNet-50		DenseNet-121	
	Time (s)	Comm. (GB)	Time (s)	Comm. (GB)	Time (s)	Comm. (GB)
Semi-honest 3PC [27]	⊥	⊥	25.9	6.9	36.0	10.5
Malicious 3PC with TEE [27]	⊥	⊥	75.4	6.9	112.9	10.5
Semi-honest 3PC [27] by [15]	10.9	2.6	26.9	6.9	37.2	10.5
Semi-honest 3PC [15]	0.6	0.8	4.7	3.8	3.6	4.6
Malicious 4PC (ours)	0.9	1.5	7.8	5.7	5.2	7.2
Malicious 3PC (ours)	13.4	8.5	82.5	47.0	84.1	55.4

Table 5: Accuracy for MNIST 4/9 distinction with various fixed-point precisions. "Prec." stands for the fixed-point precision, "Max. length" stands for the maximum bit length encountered in probabilistic truncation, "Exp. fail" stands for the expected number of failures when using cheaper truncation, and ⊥ stands for divergence. The accuracy figures are given for 10 and 20 epochs.

Layers	Prec.	Max. length	Exp. fail	Accuracy after n	
				$n = 10$	$n = 20$
1	8	22	5.5e-09	94.7	95.5
	10	27	6.6e-08	96.0	96.3
	12	31	1.0e-06	96.7	96.9
	14	35	1.6e-05	96.6	96.8
	16	39	2.6e-04	97.1	96.8
	18	43	4.1e-03	96.8	97.0
	20	47	6.2e-02	96.5	97.0
2	8	22	1.5e-07	96.5	97.1
	10	27	2.1e-06	97.5	97.8
	12	32	3.0e-05	97.7	98.5
	14	35	4.6e-04	97.9	98.3
	16	39	7.3e-03	97.7	98.6
	18	43	1.2e-01	98.2	98.6
	20	46	1.9e+00	97.8	98.2

Table 6: Time and communication per epoch for binary classification training with four parties. "Prec." stands for the fixed-point precision, and "Mod." stands for the computation modulus.

No. layers	Prec.	Mod.	Time (s)	Comm. (MB)
1	12	2^{64}	0.60	32
		2^{80}	0.56	21
	14	2^{64}	0.59	32
		2^{80}	0.55	21
	16	2^{64}	0.65	32
		2^{80}	0.59	21
2	12	2^{64}	4.17	4,217
		2^{88}	8.11	2,961
	14	2^{64}	3.90	4,218
		2^{88}	8.17	2,962
	16	2^{64}	3.86	4,218
		2^{88}	8.01	2,963

editors, *FC 2015*, volume 8975 of *LNCS*, pages 227–234. Springer, Heidelberg, January 2015.

[8] Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, Michael I. Schwartzbach, and Tomas Toft. Secure multiparty computation goes live. In Roger Dingledine and Philippe Golle, editors, *FC 2009*, volume 5628 of *LNCS*, pages 325–343. Springer, Heidelberg, February 2009.

[9] Elette Boyle, Niv Gilboa, Yuval Ishai, and Ariel Nof. Practical fully secure three-party computation via sublinear distributed zero-knowledge proofs. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 869–886. ACM Press, November 2019.

[10] Megha Byali, Harsh Chaudhari, Arpita Patra, and Ajith Suresh. FLASH: Fast and robust framework for privacy-preserving machine learning. *PoPETs*, 2020(2):459–480, April 2020.

[11] Florent Chabaud and Reynald Lercier. Zen – a toolbox for fast computation in finite extension over finite rings. `http://zenfact.sourceforge.net`, accessed 5 February 2021.

[12] H. Chaudhari, Ashish Choudhury, Arpita Patra, and A. Suresh. ASTRA: High throughput 3PC over rings with application to secure prediction. *Proceedings of the 2019 ACM SIGSAC Conference on Cloud Computing Security Workshop*, 2019.

[13] Harsh Chaudhari, Rahul Rachuri, and Ajith Suresh. Trident: Efficient 4PC framework for privacy preserving machine learning. In *NDSS 2020*. The Internet Society, February 2020.

[14] Koji Chida, Daniel Genkin, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Yehuda Lindell, and Ariel Nof. Fast large-scale honest-majority MPC for malicious adversaries. In Hovav Shacham and Alexandra Boldyreva, editors,

CRYPTO 2018, Part III, volume 10993 of *LNCS*, pages 34–64. Springer, Heidelberg, August 2018.

[15] Anders P. K. Dalskov, Daniel Escudero, and Marcel Keller. Secure evaluation of quantized neural networks. *PoPETs*, 2020(4):355–375, October 2020.

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

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

[18] Hendrik Eerikson, Marcel Keller, Claudio Orlandi, Pille Pullonen, Joonas Puura, and Mark Simkin. Use your brain! Arithmetic 3PC for any modulus with active security. In Yael Tauman Kalai, Adam D. Smith, and Daniel Wichs, editors, *ITC 2020*, pages 5:1–5:24. Schloss Dagstuhl, June 2020.

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

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

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

[22] S. Dov Gordon, Samuel Ranellucci, and Xiao Wang. Secure computation with low communication from cross-checking. In Thomas Peyrin and Steven Galbraith, editors, *ASIACRYPT 2018, Part III*, volume 11274 of *LNCS*, pages 59–85. Springer, Heidelberg, December 2018.

[23] Isabelle Guyon, Steve Gunn, Asa Ben-Hur, and Gideon Dror. Result analysis of the nips 2003 feature selection challenge. In L. K. Saul, Y. Weiss, and L. Bottou, editors, *Advances in Neural Information Processing Systems 17*, pages 545–552. MIT Press, 2005.

[24] Marcel Keller. MP-SPDZ: A versatile framework for multi-party computation. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 20*, pages 1575–1590. ACM Press, November 2020.

[25] Marcel Keller, Peter Scholl, and Nigel P. Smart. An architecture for practical actively secure MPC with dishonest majority. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 549–560. ACM Press, November 2013.

[26] Nishat Koti, Mahak Pancholi, Arpita Patra, and Ajith Suresh. Swift: Super-fast and robust privacy-preserving machine learning. Cryptology ePrint Archive, Report 2020/592, 2020. `https://eprint.iacr.org/2020/592`.

[27] Nishant Kumar, Mayank Rathee, Nishanth Chandran, Divya Gupta, Aseem Rastogi, and Rahul Sharma. CrypTFlow: Secure TensorFlow inference. In *2020 IEEE Symposium on Security and Privacy*, pages 336–353. IEEE Computer Society Press, May 2020.

[28] Andrei Lapets, Frederick Jansen, Kinan Dak Albab, Rawane Issa, Lucy Qin, Mayank Varia, and Azer Bestavros. Accessible privacy-preserving web-based data analysis for assessing and addressing economic inequalities. In *Proceedings of the 1st ACM SIGCAS Conference on Computing and Sustainable Societies*, COMPASS '18, New York, NY, USA, 2018. Association for Computing Machinery.

[29] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

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

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

[32] Arpita Patra and Ajith Suresh. BLAZE: Blazing fast privacy-preserving machine learning. In *NDSS 2020*. The Internet Society, February 2020.

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

[34] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

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

[36] Sameer Wagh, Shruti Tople, Fabrice Benhamouda, Eyal Kushilevitz, Prateek Mittal, and Tal Rabin. FALCON: honest-majority maliciously secure framework for private deep learning. *CoRR*, abs/2004.02229, 2020.

A Truncation from Share Splitting

We end with a concrete protocol for deterministic truncation with replicated secret sharing modulo a power of two that is more efficient than using edaBits because it only relies on share splitting. Let $f_\ell(x) = x - (x \bmod 2^\ell)$ for any $\ell, x > 0$, and let "/" denote floor division. It is easy to see that $f_\ell(x)$ is a multiple of 2^ℓ and hence $f_\ell(x)/2^\ell = x/2^\ell$. Let $x = (\sum_{i=1}^n x_i) \bmod 2^k$ for $x_i \in [0, 2^k - 1]$. Then,

$$\begin{aligned}
x/2^m &= \left(\left(\sum_i x_i\right) \bmod 2^k\right)/2^m \\
&= \left(\sum_i x_i - f_k\left(\sum_i x_i\right)\right)/2^m \\
&= \left(\sum_i x_i\right)/2^m - f_k\left(\sum_i x_i\right)/2^m \\
&= \left(\sum_i (f_m(x_i) + (x_i \bmod 2^m))\right)/2^m - f_k\left(\sum_i x_i\right)/2^m \\
&= \left(\sum_i f_m(x_i) + \sum_i (x_i \bmod 2^m)\right)/2^m - f_k\left(\sum_i x_i\right)/2^m \\
&= \sum_i f_m(x_i)/2^m + \left(\sum_i (x_i \bmod 2^m)\right)/2^m - f_k\left(\sum_i x_i\right)/2^m \\
&= \sum_i x_i/2^m + \left(\sum_i (x_i \bmod 2^m)\right)/2^m - f_k\left(\sum_i x_i\right)/2^m.
\end{aligned}$$

The third equality holds because 2^m divides $f_k(x)$, and the sixth equality holds because 2^m divides $f_m(x_i)$ for all i.

Now, let $n = 4$, and suppose that $x = x_1 + x_2 + x_3 + x_4 \bmod 2^k$ are the additive shares underlying a replicated sharing $[x]$. We now discuss how the parties can use the equations derived above to compute $[x/2^m]$ from $[x]$. Since x_i is known by the three parties $\{P_j\}_{j \neq i}$, the parties can locally get shares $[x_i/2^m] \leftarrow \mathsf{INPLocal}(x_i/2^m, \{P_j\}_{j \neq i})$, which in turns yields sharings of the first summand in the equation above $[\sum_{i=1}^4 x_i/2^m]$. For the other two summands, observe that, in general,

$$\sum_{i=0}^{n-1} (x_i \bmod 2^m) < n \cdot 2^m \Rightarrow \left(\sum_{i=0}^{n-1} (x_i \bmod 2^m)\right)/2^m < n$$

and

$$\sum_{i=0}^{n-1} x_i < n \cdot 2^k \Rightarrow f_k\left(\sum_{i=0}^{n-1} x_i\right) \in \{0, 2^k, \ldots, (n-1) \cdot 2^k\}.$$

Therefore, these summands consist of only $\log(n)$ non-zero bits, which in our case, since $n = 4$, leads to only two non-zero bits. It is also easy to see that these few bits can be computed using binary adders on the bits of all x_i, of which the parties can obtain shares locally. This directly leads to a protocol by computing these bits, converting them to sharings modulo 2^k, and then adding them to the sharings $[\sum_{i=1}^4 x_i/2^m]$.

B A Note on the SWIFT Benchmarks

Koti et al. [26] report relatively similar *total* timings for their 3PC and 4PC protocol (within 50 percent of each other). The two protocols differ in that the the former uses the verification by Boyle et al. [9] while the latter does not. In Section 5 we show that said verification protocol might increase the computational cost by several orders of magnitude. We find that Koti et al. do not offer enough information on their implementation to dispel these concerns because they neither specify the parameters used in their implementation (such as the batch size or the order of the extension ring), nor do they detail their implementation of the potentially expensive extension ring computation.