Fast Large-Scale Honest-Majority MPC for Malicious Adversaries

Koji Chida[1], Daniel Genkin[2], Koki Hamada[1], Dai Ikarashi[1], Ryo Kikuchi[1], Yehuda Lindell[3], and Ariel Nof[3]

[1] NTT Secure Platform Laboratories, Japan
{ikarashi.dai, kikuchi.ryo, chida.koji, hamada.koki}@lab.ntt.co.jp
[2] University of Michigan, USA
genkin@umich.edu
[3] Bar-Ilan University, Israel*
{yehuda.lindell, ariel.nof}@biu.ac.il

Abstract. Protocols for secure multiparty computation enable a set of parties to compute a function of their inputs without revealing anything but the output. The security properties of the protocol must be preserved in the presence of adversarial behavior. The two classic adversary models considered are *semi-honest* (where the adversary follows the protocol specification but tries to learn more than allowed by examining the protocol transcript) and *malicious* (where the adversary may follow any arbitrary attack strategy). Protocols for semi-honest adversaries are often far more efficient, but in many cases the security guarantees are not strong enough.

In this paper, we present new protocols for securely computing any functionality represented by an arithmetic circuit. We utilize a new method for verifying that the adversary does not cheat, that yields a cost of just *twice* that of semi-honest protocols in some settings. Our protocols are information-theoretically secure in the presence of a malicious adversaries, assuming an honest majority. We present protocol variants for small and large fields, and show how to efficiently instantiate them based on replicated secret sharing and Shamir sharing. As with previous works in this area aiming to achieve high efficiency, our protocol is *secure with abort* and does not achieve fairness, meaning that the adversary may receive output while the honest parties do not.

We implemented our protocol and ran experiments for different numbers of parties, different network configurations and different circuit depths. Our protocol significantly outperforms the previous best for this setting (Lindell and Nof, CCS 2017); for a large number of parties, our implementation runs almost an order of magnitude faster than theirs.

1 Introduction

1.1 Background

Protocols for secure computation enable a set of parties with private inputs to compute a joint function of their inputs while revealing nothing but the output.

* Supported by the European Research Council under the ERC consolidators grant agreement n. 615172 (HIPS) and by the BIU Center for Research in Applied Cryptography and Cyber Security in conjunction with the Israel National Cyber Bureau in the Prime Minister's Office.

The security properties typically required from secure computation protocols include *privacy* (meaning that nothing but the output is revealed), *correctness* (meaning that the output is correctly computed), *independence of inputs* (meaning that a party cannot choose its input as a function of the other parties' inputs), *fairness* (meaning that if one party gets output then so do all), and *guaranteed output delivery* (meaning that all parties always receive output). Formally, the security of a protocol is proven by showing that it behaves like an ideal execution with an incorruptible trusted party who computes the function for the parties [16, 17, 3, 7]. In some cases, fairness and guaranteed output delivery are not required, in which case we say that the protocol is secure with abort. This is standard in the case of no honest majority since not all functions can be computed fairly without an honest majority [9], but security with abort can also in order to aid the construction of highly efficient protocols (e.g., as in [2, 22]).

Despite the stringent requirements on secure computation protocols, in the late 1980s it was shown that any probabilistic polynomial-time functionality can be securely computed. This was demonstrated in the computational setting for any $t < n$ [26, 16, 19] (with security with abort for the case of $t \geq n/2$), in the information-theoretic setting with $t < n/3$ [5, 8], and in the information-theoretic setting with $t < n/2$ assuming a broadcast channel [24]. These feasibility results demonstrate that secure computation is possible. However, significant work is needed to construct protocols that are efficient enough to use in practice.

1.2 Our Contributions

In this paper, we consider the problem of constructing highly efficient protocols that are secure in the presence of *static malicious adversaries* who control at most $t < n/2$ corrupted parties. Our protocol is fundamentally *information-theoretic*, but some efficient instantiations are computational (e.g., in order to generate correlated randomness). In the aim of achieving high efficiency, our protocols do not achieve fairness (even though this is fundamentally possible in our setting where $t < n/2$).

Our constructions work by securely computing an *arithmetic circuit* representation of the functionality over a finite field $\mathbb{F}$. This representation is very efficient for computations where many additions and multiplications are needed, like secure statistics. The starting point of our protocols utilizes the significant observation made by [14, 15] that many protocols for *semi-honest multiplication* are actual secure in the presence of *malicious adversaries up to an additive attack*. This means that the only way an adversary can cheat is to make the result of the multiplication of shares of x and y be shares of $x \cdot y + d$, where d is an explicit value that the adversary knows (in this paper, we formalize this property via an ideal functionality definition). Since d is known by the adversary, it is independent of the values being multiplied, unless the adversary has some prior knowledge of these values. This property was utilized by [14, 15] by having the multiplication be over randomized values, and making any cheating be detected unless the adversary is lucky enough to make the additive attack value match between different randomizations.

Our protocol works by running multiple circuit computations in parallel: one that computes the circuit over the real inputs and others which compute the circuit over randomized inputs. The outputs of the randomized circuits are then used to verify the correctness of the "original" circuit computation, thereby constituting a SPDZ-like MAC [13]. Security is achieved by the fact that the randomness is kept secret throughout the computation, and so any cheating by the adversary will be detected. All multiplications of shares are carried out using semi-honest protocols (that are actually secure for malicious adversaries up to an additive attack). Since this dominates the cost of the secure computation overall, the resulting protocol is highly-efficient. We present different protocols for the case of small and large fields, where a field is "large" if it is bigger than 2^σ where σ is the statistical security parameter. Our protocol for large fields requires computing one randomized circuit only, and the protocol for small fields requires δ randomized circuits where δ is such that $(|\mathbb{F}|/3)^\delta \geq 2^\sigma$. We note that our protocol for small fields can be run with $\delta = 1$ in the case of a large field, but in this case has about 10% more communication than the protocol that is dedicated to large fields. Both protocols have overall communication complexity that grows *linearly* with the number of parties (specifically, each party sends a constant number of field elements for each multiplication gate).

Based on the above, the running time of our protocol over large fields is just *twice* the cost of a semi-honest protocol, and the running time of our protocol over small fields is just $\delta + 1$ times the cost of a semi-honest protocol. As we discuss in the related work below, this is far more efficient than the protocols of [14, 15] and the more recent protocol of [22]. The exact efficiency of our protocols depends on the specific instantiations of the secret sharing method, multiplication protocol and more. As in [22], we consider two main instantiations: one based on Shamir secret sharing for any number of parties, and one based on replicated secret sharing for the specific case of three parties. With our protocol we show that it is possible to compute any arithmetic circuit over large fields in the presence of malicious adversaries and an honest majority, at the cost of each party sending only *12 field elements* per multiplication gate. For 3-party computation, we show that using replicated sharing, this cost can be reduced to only *2 field elements* sent by each party per multiplication gate.

1.3 Experimental Results

We implemented our protocol for large fields, using replicated secret sharing for 3 parties and Shamir sharing for any number of parties. We then ran our implementation on AWS in two configurations: a *LAN network configuration* in a single AWS region (specifically, North Virginia), and a *WAN network configuration* with parties spread over three AWS regions (specifcally, North Virginia, Germany and India). Each party was run in an independent AWS `C4.large` instance (2-core Intel Xeon E5-2666 v3 with 2.9 GHz clock speed and 3.75 GB RAM). We ran extensive experiments to analyze the efficiency of our protcols for different numbers of parties on a series of circuits of different depths, each with 1,000,000 multiplication gates, 1,000 inputs wires, and 50 output wires.

The field we used for all our experiments was the 61-bit Mersenne field (and so security is approximately 2^{-60}). Our experiments show that our protocols have very good performance for all ranges of numbers of parties, especially in the LAN configuration (due to the protocol not being constant round). In particular, this 1 million gates large circuit with depth-20 can be computed in the LAN configuration in about 300 milliseconds with three parties, 4 seconds with 50 parties, and 8 seconds with 110 parties. In the WAN configuration, the running time for this circuit is about 20 seconds for 3 parties, and about 2 minutes for 50 parties (at depth 100, the running time ranges from 45 seconds for 3 parties to 3.25 minutes for 50 parties). Thus, our protocols can be used in practice to compute arithmetic computations (like joint statistics) between many parties, while providing malicious security.

The previous best result in this setting was recently achieved in [22]. A circuit of the same size and depth-20 was computed by them in half a second with three parties, 29 seconds with 50 parties and 70 seconds with 100 parties. Our protocols run much faster than theirs, from approximately *twice as fast* for a small number of parties and up to *10 times faster* for a large number of parties.

1.4 Related Work

There is a large body of research focused on improving the efficiency of secure computation protocols. This work is roughly divided up into constructions of *concretely efficient* and *asymptotically efficient* protocols. Concretely efficient protocols are often implemented and aim to obtain the best overall running time, even if the protocol is not asymptotically optimal (e.g., it may have quadratic complexity and not linear complexity, but for a small number of parties the constants are such that the quadratic protocol is faster). Asymptotically efficient protocols aim to reduce the cost of certain parts of the protocols (rounds, communication complexity, etc.), and are often not concretely very efficient. However, in many cases, asymptotically efficient protocols provide techniques that inform the construction of concretely efficient protocols.

In the case of multiparty computation (with more than two parties) with a dishonest majority, concretely efficient protocols were given in [13, 11, 6, 20]. This setting is much harder than that of an honest majority, and the results are therefore orders of magnitude slower (the state-of-the art SPDZ protocol [21] achieves a throughput of around 30,000 multiplication gates per second with 2 parties in some settings whereas we achieve a throughput of more than 1 million gates per second). For the case of an honest majority and arithmetic circuits, the previous best protocol is that of [22], and they include an in-depth comparison of their protocol to previous work, both concretely and asymptotically. Our protocol is fundamentally different from [22]. In their protocol, they use Beaver triples to verify correctness. Their main observation is that it is much more efficient to replace expensive opening operations with multiplication operation, since multiplication can be done with constant communication cost per party in the honest majority setting. We also use this observation but do not use Beaver triples at all. Thus, in our protocol, the parties are not required to generate and store these triples. As a result, our protocol has half the communication cost of [22] for 3 parties using replicated secret sharing (with each party sending 2 field elements here versus 4 field elements in [22] per multiplication gate), and less than a third of the communication cost of [22] for many parties using Shamir sharing (with each party sending 12 field elements here versus 42 field elements in [22] per multiplication gate). Experimentally, our protocol way outperforms [22], as shown in Section 6.3, running up to almost 10 times faster for a large number of parties.

The setting of $t < n/2$ and malicious adversaries was also studied in [2, 1, 23], including implementations. However, they consider only three parties and Boolean circuits.

2 Preliminaries and Definitions

Notation. Let $P_1, ..., P_n$ denote the n parties participating in the computation, and let t denote the number of corrupted parties. In this work, we assume an honest majority, and thus $t < \frac{n}{2}$. Throughout the paper, we use H to denote the subset of honest parties and $\mathcal{C}$ to denote the subset of corrupted parties. Finally, we denote by $\mathbb{F}$ a finite field and by $|\mathbb{F}|$ its size.

2.1 Threshold Secret Sharing

A t-out-of-n secret sharing scheme enables n parties to share a secret $v \in \mathbb{F}$ so that no subset of t parties can learn any information about it, while any subset of $t + 1$ parties can reconstruct it. We require that the secret sharing scheme used in our protocol supports the following procedures:

– $\mathsf{share}(v)$: In this procedure, a dealer shares a value $v \in \mathbb{F}$. For simplicity, we consider *non-interactive* secret sharing, where there exists a probabilistic dealer D that receives v (and some randomness) and outputs shares $v_1, \ldots, v_n$, where v_i is the share intended for party P_i. We denote the sharing of a value v by $[v]$. We use the notation $[v]_J$ to denote the shares held by a subset of parties $J \subset \{P_1, \ldots, P_n\}$. We stress that if the dealer is corrupted, then the shares received by the parties may not be correct. Nevertheless, we abuse notation and say that the parties hold shares $[v]$ even if these are not correct. We will define correctness of a sharing formally below.
– $\mathsf{share}(v, [v]_J)$: This non-interactive procedure is similar to the previous procedure, except that here the shares of a subset J of parties with $|J| \leq t$ are fixed in advance. We assume that there exists a probabilistic algorithm $\tilde{D}$ that receives v, $[v]_J = \{v'_j\}_{j|P_j \in J}$ (and some randomness) and outputs shares $v_1, \ldots, v_n$ where v_i is party P_i's share, and $v_j = v'_j$ for every $P_j \in J$.
  We also assume that if $|J| = t$, then $[v]_J$ together with v *fully determine* all shares $v_1, \ldots, v_n$. This also means that any $t + 1$ shares fully determine all shares. (This follows since with $t+1$ shares one can always obtain v. However, for the secret sharing schemes we use, this holds directly as well.)

– $\mathsf{reconstruct}([v], i)$: Given a sharing of v and an index i held by the parties, this interactive protocol guarantees that if $[v]$ is not correct (see formal definition below), then P_i will output $\perp$ and abort. Otherwise, if $[v]$ is correct, then P_i will either output v or will abort.
– $\mathsf{open}([v])$: Given a sharing of v held by the parties, this procedure guarantees that at the end of the execution, if $[v]$ is not correct, then *all* the honest parties will abort. Otherwise, if $[v]$ is correct, then each party will either output v or will abort. Clearly, open can be run by any subset of $t + 1$ or more parties. We require that if any subset J of $t + 1$ honest parties output a value v, then any superset of J will output either v or $\perp$ (but no other value).
– *local operations*: Given correct sharings $[u]$ and $[v]$ and a scalar $\alpha \in \mathbb{F}$, the parties can generate correct sharings of $[u+v]$, $[\alpha \cdot v]$ and $[v+\alpha]$ using local operations only (i.e., without any interaction). We denote these local operations by $[u] + [v], \alpha \cdot [v]$, and $[v] + \alpha$, respectively.

Standard secret sharing schemes like Shamir and replicated secret sharing support all of these procedures (with their required properties). Throughout the entire paper, we set the threshold for the secret sharing scheme to be $\lfloor \frac{n-1}{2} \rfloor$, and we denote by t the number of corrupted parties. Since we assume an honest majority, it holds that $t < n/2$ and so the corrupted parties can learn nothing about a shared secret.

We now define correctness for secret sharing. Let J be a subset of honest parties of size $t + 1$, and denote by $\mathsf{val}([v])_J$ the value obtained by these parties after running the open procedure, where no corrupted parties or additional honest parties participate. Note that $\mathsf{val}([v])_J$ may equal $\perp$ if the shares held by the honest parties are not valid. Informally, a secret sharing is correct if every subset of $t + 1$ honest parties reconstruct the same value (which is not $\perp$). Formally:

Definition 2.1. *Let $H \subseteq \{P_1, \dots, P_n\}$ denote the set of honest parties. A sharing $[v]$ is* $\mathsf{correct}$ *if there exists a value $v' \in \mathbb{F}$ ($v' \neq \perp$) such that for every $J \subseteq H$ with $|J| = t + 1$ it holds that $\mathsf{val}([v])_J = v'$.*

In the full version of the paper we show how to efficiently verify that a series of m shares are correct. Although not required in our general protocol, in some of our instantiations it is needed to verify that sharing of secrets is carried out correctly.

2.2 Security Definition

We use the standard definition of security based on the ideal/real model paradigm [7, 19], with security formalized for non-unanimous abort. This means that the adversary first receives the output, and then determines for each honest party whether they will receive abort or receive their correct output.

3 Building Blocks and Sub-Protocols

In this section, we define a series of building blocks that we need for our protocol. The presentation here is general, and each basic protocol can be efficiently realized using standard secret sharing schemes. We describe these instantiations in Section 6.2.

3.1 Generating Random shares

We define the ideal functionality $\mathcal{F}_{\text{rand}}$ to generate a sharing of a random value unknown to the parties. A formal description appears in Functionality 3.1. The functionality lets the adversary choose the corrupted parties' shares, which together with the random secret chosen by the functionality, are used to compute the shares of the honest parties.

FUNCTIONALITY 3.1 ($\mathcal{F}_{\text{rand}}$- Generating Random Shares)

Upon receiving α_i for each i with $P_i \in \mathcal{C}$ from the ideal adversary $\mathcal{S}$, the ideal functionality $\mathcal{F}_{\text{rand}}$ chooses a random $r \in \mathbb{F}$, sets $[r]_{\mathcal{C}} = \{\alpha_i\}_{i|P_i \in \mathcal{C}}$ and runs $\mathsf{share}(r, [r]_{\mathcal{C}})$ to receive a share r_i for each party P_i. Then, it hands each honest party P_j its share r_j.

As we have mentioned, the way we compute this functionality depends on the specific secret sharing scheme that is being used, and will be described in the instantiations in Section 6.2.

3.2 Generating Random Coins

$\mathcal{F}_{\text{coin}}$ is an ideal functionality that chooses a random element from $\mathbb{F}$ and hands it to all parties. A simple way to compute $\mathcal{F}_{\text{coin}}$ is to use $\mathcal{F}_{\text{rand}}$ to generate a random sharing and then open it. We formally describe and prove this in the full version of the paper.

3.3 $\mathcal{F}_{\text{input}}$ – Secure Sharing of Inputs

In this section, we present our protocol for secure sharing of the parties' inputs. The protocol is very simple: for each input x belonging to a party P_j, the parties call $\mathcal{F}_{\text{rand}}$ to generate a random sharing $[r]$; denote the share held by P_i by r_i. Then, r is reconstructed to P_j, who echo/broadcasts $x - r$ to all parties. Finally, each P_i outputs the share $[r + (x - r)] = [x]$. This is secure since $\mathcal{F}_{\text{rand}}$ guarantees that the sharing of r is correct, which in turn guarantees that the sharing of x is correct (since adding $x - r$ is a local operation only). In order to ensure that P_j sends the same value $x - r$ to all parties, a basic echo-broadcast is used. This is efficient since all inputs can be shared in parallel, utilizing a single echo broadcast. The formal definition of the ideal functionality for input sharing appears in Functionality 3.2.

FUNCTIONALITY 3.2 ($\mathcal{F}_{\text{input}}$- Sharing of Inputs)

1. Functionality $\mathcal{F}_{\text{input}}$ receives inputs $v_1, \ldots, v_M \in \mathbb{F}$ from the parties. For every $i = 1, \ldots, M$, $\mathcal{F}_{\text{input}}$ also receives from $\mathcal{S}$ the shares $[v_i]$ of the corrupted parties for the ith input.
2. For every $i = 1, \ldots, M$, $\mathcal{F}_{\text{input}}$ computes all shares $(v_i^1, \ldots, v_i^n) = \mathsf{share}(v_i, [v_i]_{\mathcal{C}})$.
   For every $j = 1, \ldots, n$, $\mathcal{F}_{\text{input}}$ sends P_j its output shares $(v_1^j, \ldots, v_M^j)$.

A formal description appears in Protocol 3.3

PROTOCOL 3.3 (Secure Sharing of Inputs)

- **Inputs:** Let $v_1, \ldots, v_M \in \mathbb{F}$ be the series of inputs; each v_i is held by some P_j.
- **The protocol:**
  1. The parties call $\mathcal{F}_{\text{rand}}$ M times to obtain sharings $[r_1], \ldots, [r_M]$.
  2. For $i = 1, \ldots, M$, the parties run $\mathsf{reconstruct}([r_i], j)$ for P_j to receive r_i, where P_j is the owner of the ith input. If P_j receives $\perp$, then it sends $\perp$ to all parties, outputs abort and halts.
  3. For $i = 1, \ldots, M$, party P_j sends $w_i = v_i - r_i$ to all parties.
  4. All parties send $\vec{w} = (w_1, \ldots, w_M)$, or a collision-resistant hash of the vector, to all other parties. If any party receives a different vector to its own, then it outputs $\perp$ and halts.
  5. For each $i = 1, \ldots, M$, the parties compute $[v_i] = [r_i] + w_i$.
- **Outputs:** The parties output $[v_1], \ldots, [v_M]$.

We now prove that Protocol 3.3 securely computes $\mathcal{F}_{\text{input}}$ specified in Functionality 3.2.

Proposition 3.4. *Protocol 3.3 securely computes Functionality 3.2 with abort in the presence of malicious adversaries controlling $t < n/2$ parties.*

Proof: Let $\mathcal{A}$ be the real adversary. We construct a simulator $\mathcal{S}$ as follows:

1. $\mathcal{S}$ receives $[r_i]_{\mathcal{C}}$ (for $i = 1, \ldots, M$) that $\mathcal{A}$ sends to $\mathcal{F}_{\text{rand}}$ in the protocol.
2. For every $i \in \{1, \ldots, M\}$, $\mathcal{S}$ chooses a random r_i and computes $(r_i^1, \ldots, r_i^n) = \mathsf{share}(r_i, [r_i]_{\mathcal{C}})$. (This computation may be probabilistic or deterministic, depending on how many parties are corrupted.)
3. $\mathcal{S}$ simulates the honest parties in all $\mathsf{reconstruct}$ executions. If an honest party P_j receives $\perp$ in the reconstruction, then $\mathcal{S}$ simulates it sending $\perp$ to all parties. Then, $\mathcal{S}$ simulates all honest parties aborting.
4. $\mathcal{S}$ simulates the remainder of the execution, obtaining all w_i values from $\mathcal{A}$ associated with corrupted parties' inputs, and sending random w_j values for inputs associated with honest parties' inputs.
5. For every i for which the ith input is that of a corrupted party P_j, simulator $\mathcal{S}$ sends the trusted party computing $\mathcal{F}_{\text{input}}$ the input value $v_i = w_i + r_i$.
6. For every $i = 1, \ldots, M$, $\mathcal{S}$ defines the corrupted parties' shares $[v_i]_{\mathcal{C}}$ to be $[r_i + w_i]_{\mathcal{C}}$. (Observe that $\mathcal{S}$ has $[v_i]_{\mathcal{C}}$ and merely needs to add the scalar w_i to each corrupted party's share.) Then, $\mathcal{S}$ sends $[v_1]_{\mathcal{C}}, \ldots, [v_M]_{\mathcal{C}}$ to the trusted party computing $\mathcal{F}_{\text{input}}$.
7. For every honest party P_j, if it aborted in the simulation, then $\mathcal{S}$ sends abort_j to the trusted party computing $\mathcal{F}_{\text{input}}$; else, it sends $\mathsf{continue}_j$.
8. $\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs.

The only difference between the simulation by $\mathcal{S}$ and a real execution is that $\mathcal{S}$ sends random values w_j for inputs associated with honest parties' inputs. However, by the perfect secrecy of secret sharing, this is distributed identically to a real execution. ■

3.4 Secure Multiplication up to Additive Attacks [14, 15]

Our construction works by running a multiplication protocol (for multiplying two values that are shared among the parties) that is *not* fully secure in the presence of a malicious adversary and then running a verification step that enables the honest parties to detect cheating. In order to do this, we start with a multiplication protocols with the property that the adversary's ability to cheat is limited to carrying a so-called "additive attack" on the output. Formally, we say that a multiplication protocol is secure up to an additive attack if it realizes $\mathcal{F}_{\mathsf{mult}}$ defined in Functionality 3.5. This functionality receives input sharings $[x]$ and $[y]$ from the honest parties and an additive value d from the adversary, and outputs a sharing of $x \cdot y + d$. (Since the corrupted parties can determine their own shares in the protocol, the functionality allows the adversary to provide the shares of the corrupted parties, but this reveals nothing about the shared value.)

FUNCTIONALITY 3.5 ($\mathcal{F}_{\mathsf{mult}}$ **- Secure Mult. Up To Additive Attack)**

1. Upon receiving $[x]_H$ and $[y]_H$ from the honest parties, the ideal functionality $\mathcal{F}_{\mathsf{mult}}$ computes x, y and the corrupted parties shares $[x]_{\mathcal{C}}$ and $[y]_{\mathcal{C}}$.
2. $\mathcal{F}_{\mathsf{mult}}$ hands $[x]_{\mathcal{C}}$ and $[y]_{\mathcal{C}}$ to the ideal-model adversary/simulator $\mathcal{S}$.
3. Upon receiving d and $\{\alpha_i\}_{i|P_i \in \mathcal{C}}$ from $\mathcal{S}$, functionality $\mathcal{F}_{\mathsf{mult}}$ defines $z = x \cdot y + d$ and $[z]_{\mathcal{C}} = \{\alpha_i\}_{i|P_i \in \mathcal{C}}$. Then, it runs $\mathsf{share}(z, [z]_{\mathcal{C}})$ to obtain a share z_j for each party P_j.
4. The ideal functionality $\mathcal{F}_{\mathsf{mult}}$ hands each honest party P_j its share z_j.

As we will discuss in the instantiations section (Section 6.2), the requirements defined by this functionality can be met by several semi-honest multiplication protocols. This will allow us to compute this functionality in a very efficient way.

3.5 Checking Equality to 0

In this section, we present a protocol to check whether a given sharing is a sharing of the value 0, without revealing any further information on the shared value. The idea behind the protocol is simple. Holding a sharing $[v]$, the parties generate a random sharing $[r]$ and multiply it with $[v]$. Then, the parties open the obtained sharing and check equality to 0. This works since if $v = 0$, then multiplying it with a random r will still yield 0. In contrast, if $v \neq 0$, then the multiplication will result with 0 only when $r = 0$, which happens with probability $\frac{1}{\mathbb{F}}$ only. The protocol is formally described in Protocol 3.7. For multiplying the sharings, the parties use the $\mathcal{F}_{\mathsf{mult}}$ functionality, which allows the adversary to change the output value via an additive attack. However, since the actual value is kept unknown, the adversary does not know which value should be added in order to achieve a sharing of 0.

We prove that Protocol 3.7 realizes the ideal functionality $\mathcal{F}_{\text{checkZero}}$, which is defined in Functionality 3.6.

FUNCTIONALITY 3.6 ($\mathcal{F}_{\text{checkZero}}$ **– Checking Equality to 0)**

The ideal functionality $\mathcal{F}_{\text{checkZero}}$ receives $[v]_H$ from the honest parties and uses them to compute v. Then:

- If $v = 0$, then $\mathcal{F}_{\text{checkZero}}$ sends 0 to the simulator $\mathcal{S}$. If $\mathcal{S}$ sends reject (resp., accept), then $\mathcal{F}_{\text{checkZero}}$ sends reject (resp., accept) to the honest parties.
- If $v \neq 0$, then $\mathcal{F}_{\text{checkZero}}$ proceeds as follows:
  - With probability $\frac{1}{|\mathbb{F}|}$ it sends accept to the honest parties and $\mathcal{S}$.
  - With probability $1 - \frac{1}{|\mathbb{F}|}$ it sends reject to the honest parties and $\mathcal{S}$.

$\mathcal{F}_{\text{checkZero}}$ receives the honest parties shares, and use them to reconstruct the shared value. Then, if it is 0, then the simulator decides whether to send accept to the honest parties or reject. Otherwise, $\mathcal{F}_{\text{checkZero}}$ tosses a coin to decide what to send to the parties (i.e., in this case, the simulator is not given the opportunity to modify the output). In particular, when the checked value does not equal to 0, the output will still be accept with probability $\frac{1}{|\mathbb{F}|}$. This captures the event in Protocol 3.7 where $v \neq 0$ but $T = 0$, which happens also with probability $\frac{1}{\mathbb{F}}$ since r is uniformly distributed over $\mathbb{F}$.

PROTOCOL 3.7 (Checking Equality to 0 in the $(\mathcal{F}_{\text{rand}}, \mathcal{F}_{\mathsf{mult}})$**-Hybrid Model)**

- **Inputs:** The parties hold a sharing $[v]$.
- **The protocol:**
  1. The parties call $\mathcal{F}_{\text{rand}}$ to obtain a sharing $[r]$.
  2. The parties call $\mathcal{F}_{\mathsf{mult}}$ on $[r]$ and $[v]$ to obtain $[T] = [r \cdot v]$
  3. The parties run open$([T])$. If a party receives $\perp$, then it outputs $\perp$. Else, it continues.
  4. Each party checks that $T = 0$. If yes, it outputs accept; else, it outputs reject.

Proposition 3.8. *Protocol 3.7 securely computes* $\mathcal{F}_{\text{checkZero}}$ *with abort in the* $(\mathcal{F}_{\text{rand}}, \mathcal{F}_{\mathsf{mult}})$*-hybrid model in the presence of malicious adversaries who control* $t < n/2$ *parties.*

Proof: Let $\mathcal{A}$ be the real adversary. We construct the simulator $\mathcal{S}$ as follows. The ideal execution begins with $\mathcal{S}$ receiving the value 0 or 1 from $\mathcal{F}_{\text{checkZero}}$ (depending on if $v = 0$ or if $v \neq 0$, respectively). It then proceeds according to the following cases:
Case 1 – $v = 0$: $\mathcal{S}$ plays the role of $\mathcal{F}_{\text{rand}}$ by receiving the corrupted parties' shares. Then, $\mathcal{S}$ plays the role of $\mathcal{F}_{\mathsf{mult}}$: it receives the corrupted parties' shares of T (i.e., $[T]_{\mathcal{C}}$) and the value d to add to the output. Finally, $\mathcal{S}$ simulates the opening of $[T]$ by playing the role of the honest parties. In this case, $\mathcal{S}$ can simulate the real world execution precisely, since it knows that $T = d$ (regardless of the value of r), and thus it can define the honest parties' shares of T by running share$(d, [T]_{\mathcal{C}})$. Then, if $d = 0$ and $\mathcal{A}$ sent the correct shares when opening, then it sends accept to $\mathcal{F}_{\text{checkZero}}$. Otherwise, the parties open to a value that is not 0, or the opening fails (if $\mathcal{A}$ sends incorrect shares). In the first case, $\mathcal{S}$ sends reject to $\mathcal{F}_{\text{checkZero}}$, whereas in the latter it sends abort and simulates the honest parties aborting in the protocol. Finally, $\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs.

Observe that in this case, $\mathcal{S}$ simulates the real world execution exactly, and thus the view of $\mathcal{A}$ in the simulation is identical to its view in a real execution. Therefore, the output of the honest parties, which is determined in this case by $\mathcal{A}$, is the same in both the real and simulated execution.

Case 2 – $v \neq 0$: In this case, $\mathcal{S}$ receives the final output from $\mathcal{F}_{\text{checkZero}}$ without being able to influence it (beyond aborting). As in the previous case, $\mathcal{S}$ receives from $\mathcal{A}$ the corrupted parties' shares of r and T and the value d to add to T. In this case, it holds that $T = r \cdot v + d$, where r and v are unknown to $\mathcal{A}$. To simulate the opening of $[T]$, the simulator $\mathcal{S}$ either sets $T = 0$ (in the case where accept was received from $\mathcal{F}_{\text{checkZero}}$) or chooses a random $T \in \mathbb{F} \setminus \{0\}$ (in the case where reject was received) and defines the honest parties' shares by running $\mathsf{share}(T, [T]_{\mathcal{C}})$. Then, $\mathcal{S}$ simulates the opening by playing the role of the honest parties. If $\mathcal{A}$ sends incorrect shares, causing the opening to fail, then $\mathcal{S}$ simulated the honest parties aborting in the real world execution and sends abort to $\mathcal{F}_{\text{checkZero}}$. Finally, $\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs.

We claim that the view of $\mathcal{A}$ is identically distributed in the simulation and in the real world execution. This holds since the only difference is in the opening of T. In particular, in the real world, a random r is chosen and then the value of T is set to be $r \cdot v + d$, whereas in the simulation T is chosen randomly from $\mathbb{F}$ (this follows from the fact that T is set to be 0 with probability $\frac{1}{|\mathbb{F}|}$ and to any value other than 0 with probability $\frac{1}{|\mathbb{F}-1|}$). In both cases, therefore, T is distributed uniformly over $\mathbb{F}$. Thus, the view of the adversary $\mathcal{A}$ is distributed identically in both executions. We now proceed to show that the output of the honest parties is also distributed identically in both executions. In the simulation, the output is accept with probability $\frac{1}{|\mathbb{F}|}$, as determined by the trusted party. In the real execution, the honest parties output accept when $T = 0$. This happens when $r \cdot v + d = 0$, i.e., when $r = -d \cdot v^{-1}$ (recall that $v \neq 0$ and so it has an inverse). Since r is distributed uniformly over $|\mathbb{F}|$, then $T = 0$ with probability $\frac{1}{|\mathbb{F}|}$, and so the honest parties' output is accept with the same probability as in the simulation. This concludes the proof. ■

4 The Protocol for Large Fields

With all the building blocks described in the previous section, we are now ready to present our protocol that computes an arithmetic circuit over a large field on the private inputs of the parties. We stress that by a large field, we mean that $2/|\mathbb{F}| \leq 2^{-\sigma}$, where σ is the statistical security parameter determining the allowed error. The protocol works by computing the circuit using $\mathcal{F}_{\mathsf{mult}}$ (i.e., a multiplication protocol that is secure up to additive attacks), and then running a verification step where the computations of all multiplication gates are verified.

The idea behind the protocol is for the parties to generate a random sharing $[r]$, and then evaluate the arithmetic circuit while preserving the invariant that on every wire, the parties hold shares of the value $[x]$ on the wire and shares of the randomized value $[r \cdot x]$. This is achieved by generating $[r]$ using $\mathcal{F}_{\text{rand}}$

(Section 3.1) and then multiplying each shared input with $[r]$ using $\mathcal{F}_{\mathsf{mult}}$ (Section 3.4). For addition and multiplication-by-a-constant gates, each party can locally compute the sharings of both the output and the randomized output (since $[r \cdot x] + [r \cdot y] = [r \cdot (x + y)]$). For multiplication gates, the parties interact to compute shares of $[x \cdot y]$ and $[r \cdot x \cdot y]$ (given shares of $x, r \cdot x, y, r \cdot y$). This is achieving by running $\mathcal{F}_{\mathsf{mult}}$ on $[x]$ and $[y]$ to obtain a sharing $[z] = [x \cdot y]$, and running $\mathcal{F}_{\mathsf{mult}}$ on $[r \cdot x]$ and $[y]$ to obtain a sharing $[r \cdot z] = [r \cdot x \cdot y]$ (equivalently, this latter sharing could be generated by multiplying $[x]$ with $[r \cdot y]$). The randomized value $[r]$ essentially plays the role of the MAC key in the SPDZ approach [13], enabling the parties to validate that $[x]$ is correct by utilizing the computed MAC tag $[r \cdot x]$. As we will see, we verify all MAC values via a single random combination at the end of the computation as in the SPDZ protocol.

As we have described in Section 3.4, the multiplication subprotocol that we use – and is modeled in $\mathcal{F}_{\mathsf{mult}}$ – is only secure up to additive attacks. The circuit randomization technique described above is aimed at preventing the adversary from carrying out such an attack without getting caught. In order to see why, consider an attacker who carries out an additive attack when multiplying $[x]$ and $[y]$ so that the result is $[x \cdot y + d]$ for some $d \neq 0$. Then, in order for the invariant to be maintained, the adversary needs to cheat in the other multiplication for this gate so that the result is $[r \cdot (x \cdot y + d)]$. Now, since the adversary can only carry out an *additive* attack, it must make the result be $[r \cdot x \cdot y + d']$, where $d' = d \cdot r$. However, r is not known, and thus the attacker can only succeed in this with probability $1/|\mathbb{F}|$. Thus, in order to prevent cheating in the multiplication gates, it suffices to check that the invariant is preserved over all wires.

This verification is carried out as follows. After the entire circuit has been evaluated, the parties compute a *random* linear combination of the sharings of the values and the sharings of the randomized values on each multiplication gate's output wire; denote the former by $[w]$ and the latter by $[u]$. Then, $[r]$ is opened, and the parties locally multiply it with $[w]$. Clearly, if there was no cheating, then $r \cdot [w]$ equals $[u]$, and thus $[u] - r \cdot [w]$ equals 0, which can be checked using $\mathcal{F}_{\mathrm{checkZero}}$ (Section 3.5). In contrast, if the adversary did cheat, then as we have mentioned above, the invariant will not hold except with probability $1/|\mathbb{F}|$. In this case, as we will show below, $[u] = r \cdot [w]$ with probability only $1/|\mathbb{F}|$ (since they are generated via a random linear combination). When $[u - r \cdot w] \neq 0$, then $\mathcal{F}_{\mathrm{checkZero}}$ outputs reject except with probability $1/|\mathbb{F}|$. Overall, we therefore have that the adversary can cheat with probability at most $2/|\mathbb{F}|$. We prove this formally in Lemma 4.2. A full specification appears in Protocol 4.1.

We will provide an exact complexity analysis below (in Section 6.2), but for now observe that the cost is dominated by just 2 invocations of $\mathcal{F}_{\mathsf{mult}}$ per multiplication gate. As we have mentioned in Section 3.4, $\mathcal{F}_{\mathsf{mult}}$ is securely realized in the presence of malicious adversaries by several *semi-honest* multiplication protocols. Thus, the overall cost of achieving malicious security here is very close to the semi-honest cost.

PROTOCOL 4.1 (Computing Arithmetic Circuits Over Large Fields)

Inputs: Each party P_j ($j \in \{1,\ldots,n\}$) holds an input $x_j \in \mathbb{F}^\ell$.

Auxiliary Input: The parties hold the description of a finite field $\mathbb{F}$ (with $3/|\mathbb{F}| \leq 2^{-\sigma}$) and an arithmetic circuit C over $\mathbb{F}$ that computes f on inputs of length $M = \ell \cdot n$. Let N be the number of multiplication gates in C.

The protocol (throughout, if any party receives $\perp$ as output from a call to a sub-functionality, then it sends $\perp$ to all other parties, outputs $\perp$ and halts):

1. *Secret sharing the inputs:*
   (a) For each input v_i held by party P_j, party P_j sends v_i to $\mathcal{F}_{\text{input}}$.
   (b) Each party P_j records its vector of shares $(v_1^j,\ldots,v_M^j)$ of all inputs, as received from $\mathcal{F}_{\text{input}}$. If a party received $\perp$ from $\mathcal{F}_{\text{input}}$, then it sends abort to the other parties and halts.
2. *Generate randomizing share:* The parties call $\mathcal{F}_{\text{rand}}$ to receive a sharing $[r]$.
3. *Randomization of inputs:* For each input wire sharing $[v_m]$ (where $m \in \{1,\ldots,M\}$), the parties call $\mathcal{F}_{\mathsf{mult}}$ on $[r]$ and $[v_m]$ to receive $[r \cdot v_m]$.
4. *Circuit emulation:* Let $G_1,...,G_N$ be a predetermined topological ordering of the gates of the circuit. For $k = 1,...,N$ the parties work as follows:
   – *G_k is an addition gate:* Given pairs $([x],[r\cdot x])$ and $([y],[r\cdot y])$ on the *left* and *right* input wires respectively, the parties locally compute $([x+y],[r\cdot x]+[r\cdot y]) = ([x+y],[r\cdot(x+y)])$.
   – *G_k is a multiplication-by-constant gate:* Given $([x],[r\cdot x])$ on the input wire and constant $a \in \mathbb{F}$, the parties locally compute $([a\cdot x],[r\cdot(a\cdot x)])$.
   – *G_k is a multiplication gate:* Given pair $([x],[r\cdot x])$ and $([y],[r\cdot y])$ on the *left* and *right* input wires respectively:
   (a) The parties call $\mathcal{F}_{\mathsf{mult}}$ on $[x]$ and $[y]$ to receive $[x\cdot y]$.
   (b) The parties call $\mathcal{F}_{\mathsf{mult}}$ on $[r_i\cdot x]$ and $[y]$ to receive $[r_i\cdot x\cdot y]$.
5. *Verification stage: Before* the secrets on the output wires are reconstructed, the parties verify that all the multiplications were carried out correctly, as follows. Let $\{([z_k],[r\cdot z_k])\}_{k=1}^N$ be the pairs on the output wires of all multiplication gates and let $\{([v_m],[r\cdot v_m])\}_{m=1}^M$ be the pairs on the input wires of the circuit.
   (a) The parties call $\mathcal{F}_{\text{coin}}$ to receive $\alpha_1,\ldots,\alpha_N,\beta_1,\ldots,\beta_M \in \mathbb{F}$.
   (b) The parties locally compute
   $$[u] = \sum_{k=1}^N \alpha_k\cdot[r\cdot z_k] + \sum_{m=1}^M \beta_m\cdot[r\cdot v_m] \quad \text{and} \quad [w] = \sum_{k=1}^N \alpha_k\cdot[z_k] + \sum_{m=1}^M \beta_m\cdot[v_m].$$
   (c) The parties run open$([r])$ to receive r.
   (d) Each party locally computes $[T] = [u] - r\cdot[w]$.
   (e) The parties call $\mathcal{F}_{\text{checkZero}}$ on $[T]$. If $\mathcal{F}_{\text{checkZero}}$ outputs reject, the parties output $\perp$ and abort. Else, if it outputs accept, the parties proceed to the next step.
6. *Output reconstruction:* For each output wire of the circuit, the parties run reconstruct$([v],j)$, where $[v]$ is the sharing of the value on the output wire, and P_j is the party whose output is on the wire.
   If a party received $\perp$ in any call to the reconstruct procedure, then it sends $\perp$ to the other parties, outputs $\perp$ and halts.

Output: If a party has not output $\perp$, then it outputs the values it received on its output wires.

We begin by proving that the verification step has the property that if the adversary cheats in a multiplication gate, then the $T = 0$ with probability $\frac{2}{|\mathbb{F}|} \leq 2^{-\sigma}$ (where σ is the statistical security parameter). We call a $\mathsf{multiplication\ triple}$ of a multiplication gate, the triple of values $([x], [y], [z])$, where $[x], [y]$ are the shares on the inputs wires, and $[z]$ is the shares on the output wire, after the multiplication. (Note that z may not equal $x \cdot y$, if the adversary cheated in the multiplication.)

Lemma 4.2. *If $\mathcal{A}$ sends an additive value $d \neq 0$ in any of the calls to $\mathcal{F}_{\mathsf{mult}}$ in the execution of Protocol 4.1, then the value $[T]$ computed in the verification stage of Step 5 in Protocol 4.1 equals* 0 *with probability less than* $2/|\mathbb{F}|$.

Proof: The intuition has been discussed above, and we therefore proceed directly to the proof. Consider the multiplication triple $([x_k], [y_k], [z_k])$ for the kth multiplication gate. We stress that the values on the input wires $[x_k], [y_k]$ may not actually be the appropriate values as when the circuit is computed by honest parties. However, in order to prove the lemma, we consider each gate separately, and all that is important is whether the invariant described above holds on the output wire (i.e., the randomized result is $[r \cdot z_k]$ for whatever z_k is here). By the definition of $\mathcal{F}_{\mathsf{mult}}$, a malicious adversary is able to carry out an additive attack, meaning that it can add a value to the output of each multiplication gate. Thus, it holds that $\mathsf{val}([z_k])_H = x_k \cdot y_k + d_k$ and $\mathsf{val}([r \cdot z_k])_H = (r \cdot x_k + e_k) \cdot y_k + f_k$, where $d_k, e_k, f_k \in \mathbb{F}$ are the added values in the additive attacks, as follows. The value d_k is the value added by the adversary when $\mathcal{F}_{\mathsf{mult}}$ is called with $[x_k]$ and $[y_k]$. The value e_k is such that the input to $\mathcal{F}_{\mathsf{mult}}$ for the randomized multiplication is $[y_k]$ and $[r \cdot x_k + e_k]$. This is an accumulated error on the randomized value from previous gates. Finally, f_k is the value added by the adversary when $\mathcal{F}_{\mathsf{mult}}$ is called with the shares $[y_k]$ and $[r \cdot x_k + e_k]$. Similarly, for each input wire with sharing $[v_m]$, it holds that $\mathsf{val}([r \cdot v_m])_H = r \cdot v_m + g_m$, where $g_m \in \mathbb{F}$ is the value added by the adversary when $\mathcal{F}_{\mathsf{mult}}$ is called with $[r]$ and the shared input $[v_m]$. Thus, we have that

$$\mathsf{val}([u])_H = \sum_{k=1}^{N} \alpha_k \cdot ((r \cdot x_k + e_k) \cdot y_k + f_k) + \sum_{m=1}^{M} \beta_m \cdot (r \cdot v_m + g_m)$$

$$\mathsf{val}([w])_H = \sum_{k=1}^{N} \alpha_k \cdot (x_k \cdot y_k + d_k) + \sum_{m=1}^{M} \beta_m \cdot v_m$$

and so

$$\begin{aligned}\mathsf{val}([T])_H &= \mathsf{val}([u])_H - r \cdot \mathsf{val}([w])_H = \\ &= \sum_{k=1}^{N} \alpha_k \cdot ((r \cdot x_k + e_k) \cdot y_k + f_k) + \sum_{m=1}^{M} \beta_m \cdot (r \cdot v_m + g_m) \\ &\qquad - r \cdot \left(\sum_{k=1}^{N} \alpha_k \cdot (x_k \cdot y_k + d_k) + \sum_{m=1}^{M} \beta_m \cdot v_m \right) \\ &= \sum_{k=1}^{N} \alpha_k \cdot (e_k \cdot y_k + f_k - r \cdot d_k) + \sum_{m=1}^{M} \beta_m \cdot g_m. \qquad (1)\end{aligned}$$

where the second equality holds because r is opened and so the multiplication $r \cdot [w]$ always yields $[r \cdot w]$. Our aim is to show that $\mathsf{val}([T])_H$, as shown in Eq. (1), equals 0 with probability at most $1/|\mathbb{F}|$. We have the following cases.

- *Case 1 – there exists some $m \in \{1, \ldots, M\}$ such that $g_m \neq 0$:* Let m_0 be the smallest such m for which this holds. Then, $\mathsf{val}([T])_H = 0$ if and only if

$$\beta_{m_0} = \left(-\sum_{k=1}^{N} \alpha_k \cdot (e_k \cdot y_k + f_k - r \cdot d_k) - \sum_{\substack{m=1 \\ m \neq m_0}}^{M} \beta_m \cdot g_m \right) \cdot {g_{m_0}}^{-1}$$

  which holds with probability $\frac{1}{|\mathbb{F}|}$ since β_{m_0} is distributed uniformly over $\mathbb{F}$, and chosen independently of all other values.
- *Case 2 – all $g_m = 0$:* By the assumption in the lemma, some additive value $d \neq 0$ was sent to $\mathcal{F}_{\mathsf{mult}}$. Since none was sent for the input randomization, there exists some $k \in \{1, \ldots, N\}$ such that $d_k \neq 0$ or $f_k \neq 0$. Let k_0 be the smallest such k for which this holds. Note that since this is the first error added, it holds that $e_{k_0} = 0$. Thus, in this case, $\mathsf{val}([T])_H = 0$ if and only if

$$\alpha_{k_0} \cdot (f_{k_0} - r \cdot d_{k_0}) = - \sum_{\substack{k=1 \\ k \neq k_0}}^{N} \alpha_k \cdot (e_k \cdot y_k + f_k - r \cdot d_k)\,. \qquad (2)$$

  If $f_{k_0} - r \cdot d_{k_0} \neq 0$, then the above equality holds with probability $1/|\mathbb{F}|$ since α_{k_0} is distributed uniformly over $\mathbb{F}$, and chosen independently of all other values. However, if $f_{k_0} - r \cdot d_{k_0} = 0$, then equality may hold. (Indeed, the best strategy of an adversary is to cheat in both multiplications of a single gate, and hope that the additive values cancel each other out.) Nevertheless, the probability that $f_{k_0} - r \cdot d_{k_0} = 0$ is at most $1/|\mathbb{F}|$, since r is not known to the adversary when the k_0'th gate is computed (and by the security of the secret sharing scheme, it is completely random). Thus, the probability that Eq. (2) holds is at most $\frac{1}{|\mathbb{F}|} + \left(1 - \frac{1}{|\mathbb{F}|}\right) \cdot \frac{1}{|\mathbb{F}|} < \frac{2}{|\mathbb{F}|}$.

In both cases, the probability of equality is upper bounded by $2/|\mathbb{F}|$ and this completes the proof. ■

We are now ready to prove the security of Protocol 4.1.

Theorem 4.3. *Let σ be a statistical security parameter, and let $\mathbb{F}$ be a finite field such that $3/|\mathbb{F}| \leq 2^{-\sigma}$. Let f be an n-party functionality over $\mathbb{F}$. Then, Protocol 4.1 securely computes f with abort in the $(\mathcal{F}_{\text{input}}, \mathcal{F}_{\mathsf{mult}}, \mathcal{F}_{\text{coin}}, \mathcal{F}_{\text{rand}}, \mathcal{F}_{\text{checkZero}})$-hybrid model with statistical error $2^{-\sigma}$, in the presence of a malicious adversary controlling $t < \frac{n}{2}$ parties.*

Proof: Intuitively, the protocol is secure since if the adversary cheats in any multiplication, then the value T computed in the verification stage will equal zero with probability at most $2/|\mathbb{F}|$, as shown in Lemma 4.2. Then, if indeed $T \neq 0$, this will be detected in the call to $\mathcal{F}_{\text{checkZero}}$, except with probability

$1/|\mathbb{F}|$. Thus, overall, the adversary can avoid detection with probability at most $3/|\mathbb{F}| \leq 2^{-\sigma}$.

Let $\mathcal{A}$ be the real adversary who controls the set of corrupted parties $\mathcal{C}$; the simulator $\mathcal{S}$ works as follows:

1. *Secret sharing the inputs:* $\mathcal{S}$ receives from $\mathcal{A}$ the set of corrupted parties inputs (values v_j associated with parties $P_i \in \mathcal{C}$) and the corrupted parties' shares $\{[v_i]_{\mathcal{C}}\}_{i=1}^{M}$ that $\mathcal{A}$ sends to $\mathcal{F}_{\text{input}}$ in the protocol. For each honest party's input v_j, $\mathcal{S}$ computes $(v_j^1, \ldots, v_j^n) = \mathsf{share}(0, [v_i]_{\mathcal{C}})$ (i.e., uses 0 as the input on the wire). Then, $\mathcal{S}$ hands $\mathcal{A}$ the shares of the corrupted parties for all inputs.
2. *Generate the randomizing share:* Simulator $\mathcal{S}$ receives the share $[r]_{\mathcal{C}}$ of the corrupted parties that $\mathcal{A}$ sends to $\mathcal{F}_{\text{rand}}$.
3. *Randomization of inputs:* For every input wire $m = 1, \ldots, M$, simulator $\mathcal{S}$ plays the role of $\mathcal{F}_{\mathsf{mult}}$ in the multiplication of the mth input $[v_m]$ with r. Specifically, $\mathcal{S}$ hands $\mathcal{A}$ the corrupted parties shares in $[v_m]$ and $[r]$ (it has these shares from the previous steps). Next, $\mathcal{S}$ receives the additive value $d = g_m$ and the corrupted parties' shares $[z]_{\mathcal{C}}$ of the result that $\mathcal{A}$ sends to $\mathcal{F}_{\mathsf{mult}}$. Simulator $\mathcal{S}$ stores all of these corrupted parties shares.
4. *Circuit emulation:* Throughout the emulation, $\mathcal{S}$ will use the fact that it knows the corrupted parties' shares on the input wires of the gate being computed. This holds initially from the steps above, and we will show it computes the output wires of each gate below. For each gate G_k in the circuit,
   - *If G_k is an addition gate:* Given the shares of the corrupted parties on the input wires, $\mathcal{S}$ locally adds them as specified by the protocol, and stores them.
   - *If G_k is a multiplication-by-a-constant gate:* Given the shares of the corrupted parties on the input wire, $\mathcal{S}$ locally multiplies them by the constant, them as specified by the protocol, and stores them.
   - *If G_k is a multiplication gate:* $\mathcal{S}$ plays the role of $\mathcal{F}_{\mathsf{mult}}$ in this step (as in the randomization of inputs above). Specifically, simulator $\mathcal{S}$ hands $\mathcal{A}$ the corrupted parties' shares on the input wires as it expects to receive from $\mathcal{F}_{\mathsf{mult}}$ (it has these shares by the invariant), and receives from $\mathcal{A}$ the additive value as well as the corrupted parties' shares for the output. These additive values are d_k (for the multiplication of the actual values) and f_k (for the multiplication of the randomized value), as defined in the proof of Lemma 4.2. $\mathcal{S}$ stores the corrupted parties' shares.
5. *Verification stage:* Simulator $\mathcal{S}$ works as follows. $\mathcal{S}$ chooses random $\alpha_1, \ldots, \alpha_N$, $\beta_1, \ldots, \beta_M \in \mathbb{F}$ and hands them to $\mathcal{A}$, as it expects to receive from $\mathcal{F}_{\text{coin}}$. Then, $\mathcal{S}$ chooses a random $r \in \mathbb{F}$ and computes the shares of r by $(r_1, \ldots, r_n) = \mathsf{share}(r, [r]_C)$, using the shares $[r]_C$ provided by $\mathcal{A}$ in the "generate randomizing share" step above. Next, $\mathcal{S}$ simulates the honest parties sending their shares in $\mathsf{open}([r])$ to $\mathcal{A}$, and receives the shares that $\mathcal{A}$ sends to the honest parties in this open. If any honest party would abort (it knows whether this would happen since it has all the honest parties' shares), then $\mathcal{S}$ simulates it sending $\perp$ to all parties, externally sends abort_j for every $P_j \in H$ to the trusted party computing f, and halts.

Finally, $\mathcal{S}$ simulates $\mathcal{F}_{\text{checkZero}}$, as follows. If any non-zero g_m, d_k, f_k was provided to $\mathcal{F}_{\mathsf{mult}}$ by $\mathcal{A}$ in the simulation, then $\mathcal{S}$ simulates $\mathcal{F}_{\text{checkZero}}$ sending reject, and then all honest parties sending $\perp$. Then, $\mathcal{S}$ externally sends abort_j for every $P_j \in H$ to the trusted party computing f. Otherwise, $\mathcal{S}$ proceeds to the next step.

6. *Output reconstruction:* If no abort had occurred, $\mathcal{S}$ externally sends the trusted party computing f the corrupted parties' inputs that it received in the "secret sharing the inputs" step above. $\mathcal{S}$ receives back the output values for each output wire associated with a corrupted party. Then, $\mathcal{S}$ simulates the honest parties in the reconstruction of the corrupted parties' outputs. It does this by computing the shares of the honest parties on this wire using the corrupted parties' shares on the wire (which it has by the invariant) and the actual output value it received from the trusted party.

   In addition, $\mathcal{S}$ receives the messages from $\mathcal{A}$ for the reconstructions to the honest parties. If any of the messages in the reconstruction of an output wire associated with an honest P_j are incorrect (i.e., the shares sent by $\mathcal{A}$ are not the correct shares it holds), then $\mathcal{S}$ sends abort_j to instruct the trusted party to not send the output to P_j. Otherwise, $\mathcal{S}$ sends $\mathsf{continue}_j$ to the trusted party, instructing it to send P_j its output.

We claim that the view of the adversary in the simulation is identical to its view in the real execution, except with probability $3/|\mathbb{F}|$. In order to see this, observe first that if all g_m, d_k, f_k values equal 0, then the simulation is perfect. The only difference is that the input shares of the honest parties are to 0. However, by the perfect secrecy of secret sharing, this has the same distribution as in a real execution.

Next, consider the case that some g_m, d_k, f_k value does not equal 0. In this case, the simulator $\mathcal{S}$ *always* simulates $\mathcal{F}_{\text{checkZero}}$ outputting reject. However, in a real execution where some g_m, d_k, f_k value does not equal 0, functionality $\mathcal{F}_{\text{checkZero}}$ may return accept either if $T = 0$, or if $T \neq 0$ but it chose accept with probability $1/|\mathbb{F}$ in the computation of the functionality output. By Lemma 4.2, the probability that $T = 0$ in such a real execution is less than $2/|\mathbb{F}|$, and thus $\mathcal{F}_{\text{checkZero}}$ outputs accept with probability less than $\frac{2}{|\mathbb{F}|} + \left(1 - \frac{2}{|\mathbb{F}|}\right) \cdot \frac{1}{|\mathbb{F}|} < \frac{3}{|\mathbb{F}|}$. Since this is the only difference between the real execution and the ideal-model simulation, we have that the statistical difference between these distributions is less than $\frac{3}{|\mathbb{F}|} \leq 2^{-\sigma}$, and so the protocol is secure with statistical error $2^{-\sigma}$. ■

Using pseudo-randomness to reduce the number of calls to $\mathcal{F}_{\text{coin}}$. Observe that in the verification phase, we need to call $\mathcal{F}_{\text{coin}}$ many times; once for each input wire and multiplication gate, to be exact. Instead of calling $\mathcal{F}_{\text{coin}}$ for every value (since this would be expensive), it suffices to call it once to obtain a seed for a pseudorandom generator, and then each party locally uses the seed to obtain as much randomness as needed. (Practically, the key would be an AES key, and randomness is obtained by running AES in counter mode.) It is not difficult to show that by the pseudorandomness assumption, the probability that the adversary can cheat is only negligibly different.[4]

Concrete efficiency. We analyze the performance of our protocol. The functionality $\mathcal{F}_{\mathsf{mult}}$ is called once for every input wire and twice for every multiplication gate (once for multiplying $[x]$ and $[y]$, and another time for multiplying $[r \cdot x]$ with $[y]$). Thus, the overall number of multiplications is $(M+2N) \cdot \mathcal{F}_{\mathsf{mult}}$, where M denotes the number of inputs and N the number of multiplication gates. In addition, there are M calls to $\mathcal{F}_{\text{input}}$, which by Protocol 3.3 reduces to M invocations of $\mathcal{F}_{\text{rand}}$ and M reconstructions. Furthermore, there is one call to $\mathcal{F}_{\text{rand}}$ for generating $[r]$, one call to $\mathcal{F}_{\text{coin}}$ for generating all the α_k, β_k values (which reduces to one $\mathcal{F}_{\text{rand}}$ and one open), one call to open for $[r]$, and one call to $\mathcal{F}_{\text{checkZero}}$ (which reduces to one call to $\mathcal{F}_{\text{rand}}$, one $\mathcal{F}_{\mathsf{mult}}$ and one opening). Finally, let L denote the number of output values, and so the number of $\mathsf{reconstruct}$ operations equals L in order to obtain output. We have that the overall exact cost of the protocol is

$$(M+2N+1) \cdot \mathcal{F}_{\mathsf{mult}} + (M+3) \cdot \mathcal{F}_{\text{rand}} + (M+L) \cdot \mathsf{reconstruct} + 3 \cdot \mathsf{open}.$$

Clearly, amortizing over the size of the circuit, we have that the average cost is $2 \cdot \mathcal{F}_{\mathsf{mult}}$ per multiplication gate.

Reducing memory. One issue that can arise in the implementation of Protocol 4.1 is due to the fact that the parties need to store all of the shares used throughout the computation in order to run the verification stage. If the circuit is huge (e.g., has billions of gates), then this can be problematic. However, in such cases, it is possible to run the verification multiple times. For example, one can determine that the verification is run after every million gates processed. Since this involves opening the randomizing share $[r]$, a new randomizing share $[r']$ is chosen by running $\mathcal{F}_{\text{rand}}$, and the shares on all wires that are still "active" (meaning that they are input into later gates) are randomized using $[r']$ (in the same way that the input wires are randomized). The protocol then proceeds as before. The additional cost is calling $\mathcal{F}_{\text{rand}}$ and $\mathcal{F}_{\text{checkZero}}$ once every million gates (or whatever is determined) instead of just once, and multiplying $[r']$ by all of the active wires using $\mathcal{F}_{\mathsf{mult}}$ at each such iteration instead of just for the inputs. This will typically only be worthwhile for extremely large circuits.

Small fields. Protocol 4.1 works for fields that are large enough so that $3/|\mathbb{F}|$ is an acceptable probability of an adversary cheating. In cases where it is desired to work in a smaller field, one could consider the following strategy. Instead of having a single randomizing share $[r]$, generate δ such random shares $[r_1], \ldots, [r_\delta]$ (where $(3/|\mathbb{F}|)^\delta$ is small enough). Then, run the same circuit emulation and verification steps using each r_i separately. Since each verification is independent, this will yield a cheating probability of at most $(3/|\mathbb{F}|)^\delta$, as required. The problem

[4] Note that this is not as immediate as it seems since the adversary has the seed/key as well, and so at this point the pseudorandom property is actually lost. However, the checks work by generating the randomness after everything else is finished and then verifying that some equality holds, or that the results are correct. These properties are actually determined *before* the key is revealed, and thus security is maintained even after the key is revealed.

with such a strategy is that the simulator must be able to simulate the $[T]$ values for each verification. Unlike the case of a large field, in this case, there is a good probability that some of the $[T]$ values will equal 0, even if the adversary cheated (the only guarantee is that *not all* $[T]$ values will equal 0). Looking at Eq. (1), observe that the value of $[T]$ is dependent on the values $\alpha_k, e_k, f_k, r, d_k, \beta_m, g_m$ known to the simulator, and an unknown value y_k (this is the actual value on the wire). However, *all* of these values are known to the distinguisher (since it knows the actual inputs, and also has the adversary's view) and thus it can know for certain which $[T]$ values should equal 0 and which should not. Thus, a simulation strategy where the simulator determines whether $[T]$ equals 0 with probability $1/|\mathbb{F}|$ if the adversary cheated will fail (since the distinguisher can verify if the value should actually be zero, depending on the given values). In the next section, we present a different strategy that solves this problem. In short, the strategy involves generating the linear combinations in Step 5b of Protocol 4.1 using *shared and secret* α_k and β_m values. Since these values are never revealed, the distinguisher cannot know if an actual $[T]$ should be 0 or not, and it suffices to simulate by choosing $[T]$ to equal 0 with probability $1/|\mathbb{F}|$ in the case that the adversary cheats.

5 A Protocol for Small Fields

Motivation. As discussed at the end of Section 4, Protocol 4.1 only works for large fields. In this section, we describe a protocol variant that works for any field size. The protocol is similar to Protocol 4.1 except that multiple randomizing shares and verifications are carried out. In particular, the parties generate δ random shares $[r_1], \ldots, [r_\delta]$ and then verify the correctness of all multiplications by generating δ independent random linear combinations as in Step 5b of Protocol 4.1 (each with a different r_i, and with independent α_k, β_m values). The main difference is that instead of α_k, β_m being *public* values generated by calls to $\mathcal{F}_{\text{coin}}$, they are random shares generated by calling $\mathcal{F}_{\text{rand}}$. Furthermore, they are kept secret and not opened. We show that this yields a cheating probability of at most $(3/|\mathbb{F}|)^\delta$, which can be made arbitrarily small by increasing δ. Since $\mathcal{F}_{\text{rand}}$ is somewhat more expensive than $\mathcal{F}_{\text{coin}}$ (see Section 6.2), Protocol 4.1 is better for large fields.

Secure sum of products. In order to implement the verification step with shared and secret α_k, β_m, it is necessary to compute the following linear combinations efficiently:

$$[u] = \sum_{k=1}^{N} [\alpha_k] \cdot [r \cdot z_k] + \sum_{m=1}^{M} [\beta_m] \cdot [r \cdot v_m] \quad \text{and} \quad [w] = \sum_{k=1}^{N} [\alpha_k] \cdot [z_k] + \sum_{m=1}^{M} [\beta_m] \cdot [v_m].$$

This seems to require an additional *four multiplications* (e.g., calls to $\mathcal{F}_{\mathsf{mult}}$) per multiplication gate. Given that Protocol 4.1 requires only two calls to $\mathcal{F}_{\mathsf{mult}}$ per multiplication gate overall, this seems to be considerably more expensive. In Section 6.1, we show how to compute a sum of products, for any number of terms, essentially at the cost of just a *single multiplication*. Our construction works for Shamir and replicated secret sharing, as we use in this paper. This subprotocol is of independent interest, and can be useful in many other scenarios.

For example, in statistical computations, a sum-of-squares is often needed, and our method can be used to compute the sum-of-square of millions of values at the cost of just one multiplication. We formally define the sum-of-products functionality, denoted $\mathcal{F}_{\mathsf{product}}$, in Functionality 5.1. It is very similar to $\mathcal{F}_{\mathsf{mult}}$, with the exception that it receives two lists of values instead of a single pair. As with $\mathcal{F}_{\mathsf{mult}}$, security is defined up to additive attacks.

FUNCTIONALITY 5.1 ($\mathcal{F}_{\mathsf{product}}$ **- Sum-of-Products Up To Additive Attacks**)

1. Upon receiving $\{[x_i]_H\}_{i=1}^{\ell}$ and $\{[y_i]_H\}_{i=1}^{\ell}$ from the honest parties, the ideal functionality $\mathcal{F}_{\mathsf{product}}$ computes x_i and y_i and the corrupted parties shares $[x_i]_{\mathcal{C}}$ and $[y_i]_{\mathcal{C}}$, for each $i \in \{1, \ldots, \ell\}$.
2. $\mathcal{F}_{\mathsf{product}}$ hands $\{[x_i]_{\mathcal{C}}\}_{i=1}^{\ell}$ and $\{[y_i]_{\mathcal{C}}\}_{i=1}^{\ell}$ to the ideal-model adversary $\mathcal{S}$.
3. Upon receiving d and $\{\alpha_i\}_{i|P_i \in \mathcal{C}}$ from $\mathcal{S}$, functionality $\mathcal{F}_{\mathsf{product}}$ defines $z = \sum_{i=1}^{\ell} x_i \cdot y_i + d$ and $[z]_{\mathcal{C}} = \{\alpha_i\}_{i|P_i \in \mathcal{C}}$. Then, it runs $\mathsf{share}(z, [z]_{\mathcal{C}})$ to obtain a share z_j for each party P_j.
4. The ideal functionality $\mathcal{F}_{\mathsf{product}}$ hands each honest party P_j its share z_j.

The protocol. We now proceed to describe the protocol. As we have described above, the protocol is very similar to Protocol 4.1 with the exception that the share randomization and verification are run δ times, and the linear combinations are computed using secret and shared α_k, β_m. The formal description of the protocol appears in Protocol 5.3. Observe that the computation of $[u_i]$ and $[w_i]$ in order to compute $[T_i]$ in Steps 6(c)i–6(c)iv in Protocol 5.3 is exactly the same as the computation of T in Step 5b Protocol 4.1. Namely, we obtain

$$[u_i] = \sum_{k=1}^{N}[\alpha_{k,i}] \cdot [r_i \cdot z_k] + \sum_{m=1}^{M}[\beta_{m,i}] \cdot [r_i \cdot v_m] \text{ and } [w_i] = \sum_{k=1}^{N}[\alpha_{k,i}] \cdot [z_k] + \sum_{m=1}^{M}[\beta_{m,i}] \cdot [v_m].$$

Thus, the intuition as to why $[T_i] = [u_i] - r_i \cdot [w_i]$ equals 0 with probability $3/|\mathbb{F}|$ is the same as in Protocol 4.1. Despite this, the proof is different since here $3/|\mathbb{F}|$ is noticeable, and this affects the simulation. As such, the proof of the protocol is similar to that of Protocol 4.1 with the exception that the simulator needs to compute the *exact probability* that each $T_i = 0$, depending on the different cases of possible additive attacks. This is due to the fact that some T_i may equal 0 with probability $1/|\mathbb{F}|$ even when an additive attack does take place. Unlike the case of large fields, $1/|\mathbb{F}|$ may be noticeable and thus the simulation cannot afford to just fail in such cases. As we will see in the proof, if the adversary cheats in a multiplication gate, then each $T_i = 0$ with probability at most $3/|\mathbb{F}$, and so all $T_i = 0$ with probability at most $(3/|\mathbb{F}|)^{\delta} \leq 2^{-\sigma}$, as required. Thus, the adversary cannot cheat undetected with probability greater than $2^{-\sigma}$. Nevertheless, the simulation of when $T_i = 0$ and when $T_i \neq 0$ is needed to show that revealing this fact does not leak any information about the real input.

Theorem 5.2. *Let σ be a statistical security parameter, let $\mathbb{F}$ be a finite field, and let f be a n-party functionality over $\mathbb{F}$. Then, Protocol 5.3 securely computes f with abort in the $(\mathcal{F}_{\text{input}}, \mathcal{F}_{\mathsf{mult}}, \mathcal{F}_{\text{coin}}, \mathcal{F}_{\text{rand}}, \mathcal{F}_{\text{checkZero}}, \mathcal{F}_{\mathsf{product}})$-hybrid model with statistical error $2^{-\sigma}$, in the presence of a malicious adversary controlling $t < \frac{n}{2}$ parties.*

PROTOCOL 5.3 (Computing Arithmetic Circuits Over Any Finite $\mathbb{F}$)

Inputs: Each party P_j ($j \in \{1, \ldots, n\}$) holds an input $x_j \in \mathbb{F}^\ell$.

Auxiliary Input: The parties hold the description of a finite field $\mathbb{F}$ and an arithmetic circuit C over $\mathbb{F}$ that computes f on inputs of length $M = \ell \cdot n$. Let N be the number of multiplication gates in C.

The protocol:

1. *Parameter computation:* Set δ to be the smallest value for which $\delta \geq \frac{\sigma}{\log(|\mathbb{F}|/3)}$.
2. *Secret sharing the inputs:*
   (a) For each input v_i held by party P_j, party P_j sends v_i to $\mathcal{F}_{\text{input}}$.
   (b) Each party P_j records its vector of shares $(v_1^j, \ldots, v_M^j)$ of all inputs, as received from $\mathcal{F}_{\text{input}}$. If a party received $\perp$ from $\mathcal{F}_{\text{input}}$, then it sends abort to the other parties and halts.
3. *Generate randomizing shares:* For $i = 1$ to δ, the parties call $\mathcal{F}_{\text{rand}}$ to receive a sharing $[r_i]$.
4. *Randomization of inputs:* For each input wire sharing $[v_m]$ (where $m \in \{1, \ldots, M\}$) and for every $i = 1, \ldots, \delta$, the parties call $\mathcal{F}_{\mathsf{mult}}$ on $[r_i]$ and $[v_m]$ to receive $[r_i \cdot v_m]$.
5. *Circuit emulation:* Let $G_1, ..., G_N$ be a predetermined topological ordering of the gates of the circuit. For $k = 1, ..., N$ the parties work as follows:
   - *G_k is an addition gate:* Given tuples $([x], [r_1 \cdot x], \ldots, [r_\delta \cdot x])$ and $([y], [r_1 \cdot y], \ldots, [r_\delta \cdot y])$ on the *left* and *right* input wires respectively, the parties locally compute $([x + y], [r_1 \cdot (x + y)], \ldots, [r_\delta \cdot (x + y)])$.
   - *G_k is a multiplication-by-a-constant gate:* Given a constant $a \in \mathbb{F}$ and tuple $([x], [r_1 \cdot x], \ldots, [r_\delta \cdot x])$ on the input wire, the parties locally compute $([a \cdot x], [r_1 \cdot (a \cdot x)], \ldots, [r_\delta \cdot (a \cdot x)])$.
   - *G_k is a multiplication gate:* Given tuples $([x], [r_1 \cdot x], \ldots, [r_\delta \cdot x])$ and $([y], [r_1 \cdot y], \ldots, [r_\delta \cdot y])$ on the *left* and *right* input wires respectively:
     (a) The parties call $\mathcal{F}_{\mathsf{mult}}$ on $[x]$ and $[y]$ to receive $[x \cdot y]$.
     (b) For $i = 1$ to δ, the parties call $\mathcal{F}_{\mathsf{mult}}$ on $[r_i \cdot x]$ and $[y]$ to receive $[r_i \cdot x \cdot y]$.
6. *Verification stage:* Let $\{([z_k], [r_1 \cdot z_k], \ldots, [r_\delta \cdot z_k])\}_{k=1}^N$ be the tuples on the output wires of all multiplication gates and let $\{([\beta_{m,1}], \ldots, [\beta_{m,\delta}])\}_{m=1}^M$ be the tuples on the input wires of the circuit.
   (a) For $m = 1, \ldots, M$, the parties call $\mathcal{F}_{\text{rand}}$ to receive $[\beta_{m,1}], \ldots, [\beta_{m,\delta}]$.
   (b) For $k = 1, \ldots, N$, the parties call $\mathcal{F}_{\text{rand}}$ to receive $[\alpha_{k,1}], \ldots, [\alpha_{k,\delta}]$.
   (c) *Compute linear combinations:* For $i = 1, \ldots, \delta$:
      i. The parties call $\mathcal{F}_{\mathsf{product}}$ on vectors $([\alpha_{1,i}], \ldots, [\alpha_{N,i}], [\beta_{1,i}], \ldots, [\beta_{M,i}])$ and $([r_i \cdot z_1], \ldots, [r_i, \cdot z_N], [r_i \cdot v_1], \ldots, [r_i \cdot v_M])$ to receive $[u_i]$.
      ii. The parties call $\mathcal{F}_{\mathsf{product}}$ on vectors $([\alpha_{1,i}], \ldots, [\alpha_{N,i}], [\beta_{1,i}], \ldots, [\beta_{M,i}])$ and $([z_1], \ldots, [z_N], [v_1], \ldots, [v_M])$ to receive $[w_i]$.
      iii. The parties run $\mathsf{open}([r_i])$ to receive r_i.
      iv. Each party locally computes $[T_i] = [u_i] - r_i \cdot [w_i]$.
      v. The parties call $\mathcal{F}_{\text{checkZero}}$ on $[T_i]$. If $\mathcal{F}_{\text{checkZero}}$ outputs reject, the parties output $\perp$ and abort. Else, if it outputs accept, they proceed.
7. *Output reconstruction:* For each output wire of the circuit, the parties run $\mathsf{reconstruct}([v], j)$, where $[v]$ is the sharing of the value on the output wire, and P_j is the party whose output is on the wire.
   If a party received $\perp$ in any call to the reconstruct procedure, then it sends $\perp$ to the other parties, outputs $\perp$ and halts.

Output: If a party has not aborted, it outputs the values received on its output wires.

Proof: We have already described the intuition behind the proof, and so proceed directly. Let $\mathcal{A}$ be the real adversary; we construct the ideal adversary/simulator $\mathcal{S}$ as follows. The simulation up to the verification stage is almost identical to the simulator in the proof of Theorem 4.3 for Protocol 4.1, with appropriate differences for the fact that the randomization is carried out δ times.

We now show how to simulate the verification step. As in the proof of Theorem 4.3, the simulator $\mathcal{S}$ chooses $r_1, \ldots, r_\delta \in \mathbb{F}$ at random, and generates all shares by computing $(r_i^1, \ldots, r_i^n) = \mathsf{share}(r_i, [r_i]_\mathcal{C})$, for every $i = 1, \ldots, \delta$. Next, $\mathcal{S}$ simulates $\delta \cdot (N + M)$ calls to $\mathcal{F}_{\text{rand}}$ used to obtain all of the $\beta_{m,i}$ and $\alpha_{k,i}$ values. Now, for every $i = 1, \ldots, \delta$, $\mathcal{S}$ works as follows:

1. $\mathcal{S}$ simulates two invocations of $\mathcal{F}_{\mathsf{product}}$ with $\mathcal{A}$, receiving $d_{i,1}$ and $d_{i,2}$, respectively, as the additive attack of $\mathcal{A}$ in these invocations.
2. $\mathcal{S}$ simulates the opening of r_i by handing $\mathcal{A}$ all of the honest parties' shares as computed above. If any honest party would abort due to the opening values sent by $\mathcal{A}$ ($\mathcal{S}$ knows whether this would happen since it has all the honest parties' shares), then $\mathcal{S}$ simulates the honest party sending $\perp$ to all parties, externally sends abort_j for every $P_j \in H$ to the trusted party computing f, and halts.
3. $\mathcal{S}$ simulates $\mathcal{F}_{\text{checkZero}}$, determining the value of T_i to be equal or not equal to zero, based on the process described below. If $T_i \neq 0$, then $\mathcal{S}$ simulates an abort, as in the proof of Theorem 4.3. Else, $\mathcal{S}$ proceeds with the simulation.

If $\mathcal{A}$ carried out an additive attack when calling $\mathcal{F}_{\mathsf{mult}}$ with $[x]$ and $[y]$ on a wire (i.e., the actual value multiplication and not the randomization), and yet all $\mathcal{F}_{\text{checkZero}}$ simulations return accept (either because $T_i = 0$ or because $\mathcal{F}_{\text{checkZero}}$ returns accept with probability $1/|\mathbb{F}|$ even when $T_i \neq 0$), then $\mathcal{S}$ outputs fail.

If $\mathcal{S}$ did not halt, then it concludes the output reconstruction as in the proof of Theorem 4.3.

It remains to show how $\mathcal{S}$ determines the value of T_i as equal or not equal to zero, for each $i = 1, \ldots, \delta$, and to show that this is the same distribution as in a real execution. Fix i, and let $d_k, e_{k,i}, f_{k,i}, g_{m_i}$ be as in the proof of Lemma 4.2 (the additional subscript of i for $e_{k,i}, f_{k,i}, g_{m,i}$ is due to the fact that there are separate $\mathcal{F}_{\mathsf{mult}}$ calls for each randomization multiplication; i.e., for each $i = 1, \ldots, \delta$ and the associated r_i). $\mathcal{S}$ determines the probability that $T_i = 0$ based on the following mutually-exclusive cases:

1. *Case 1 – there exists an $m \in \{1, \ldots, M\}$ such that $g_{m,i} \neq 0$:* In this case, $\mathcal{S}$ sets $T_i = 0$ with probability $1/|\mathbb{F}|$ exactly.
2. *Case 2 – $g_{m,i} = 0$ for all $m \in \{1, \ldots, M\}$ and $d_k = 0$ for all $k \in \{1, \ldots, N\}$, but there exists some $k \in \{1, \ldots, N\}$ for which $f_{k,i} \neq 0$:* As in the previous case, in this case $\mathcal{S}$ sets $T_i = 0$ with probability $1/|\mathbb{F}|$ exactly.
3. *Case 3 – $g_{m,i} = 0$ for all $m \in \{1, \ldots, M\}$ and for all $k \in \{1, \ldots, N\}$ it holds that $f_{k,i} - r_i \cdot d_k = 0$:* In this case, $\mathcal{S}$ sets $T_i = d_{i,1} - r_i \cdot d_{i,2}$ with probability 1. (Note that this case includes cases that some $d_k, f_{k,i} \neq 0$ and it happens that $f_{k,i} - r_i \cdot d_k = 0$, as well as the case that all $d_k, f_{k,i}$ equal 0 and so $\mathcal{A}$ did not cheat.)

4. *Case 4 – $g_{m,i} = 0$ for all $m \in \{1, \ldots, M\}$ and there exists a $k \in \{1, \ldots, N\}$ such that $d_k \neq 0$ and $f_{k,i} - r_i \cdot d_k \neq 0$:* In this case, $\mathcal{S}$ sets $T_i = 0$ with probability $1/|\mathbb{F}|$ exactly.

Observe that $\mathcal{S}$ knows all the additive values, and uses the random choice of r_i above, and so can determine all of the above cases. In addition, observe that this covers all possible cases.

We now analyze all of the above cases and show that the distribution over the zero/non-zero value of T_i generated by $\mathcal{S}$ is identical to that of a real execution. As in Eq. (1) in the proof of Lemma 4.2, we have that

$$\mathsf{val}([T_i])_H = \sum_{k=1}^{N} \alpha_{k,i} \cdot (e_{k,i} \cdot y_k + f_{k,i} - r_i \cdot d_k) + \sum_{m=1}^{M} \beta_{m,i} \cdot g_{m,i} + d_{i,1} - r_i \cdot d_{i,2}. \quad (3)$$

We use this to analyze the cases:

1. *Case 1:* Let $m_0 \in \{1, \ldots, M\}$ be such that $g_{m_0,i} \neq 0$. By Eq. (3) we have $\mathsf{val}([T_i])_H = 0$ if and only if $\beta_{m_0,i} = \Big(-\sum_{k=1}^{N} \alpha_{k,i} \cdot (e_{k,i} \cdot y_k + f_{k,i} - r_i \cdot d_k) - \sum_{\substack{m=1 \\ m \neq m_0}}^{M} \beta_{m,i} \cdot g_{m,i} - d_{i,1} + r_i \cdot d_{i,2}\Big) \cdot {g_{m_0,i}}^{-1}$. By the uniform choice of $\beta_{m_0,i}$, this holds in a real execution with probability $1/|\mathbb{F}|$ exactly.
2. *Case 2:* Let $k_0 \in \{1, \ldots, N\}$ be such that $f_{k_0,i} \neq 0$. As above, $\mathsf{val}([T_i])_H = 0$ if and only if $\alpha_{k_0,i} \cdot (f_{k_0,i} - r_i \cdot d_{k_0}) = -\sum_{\substack{k=1 \\ k \neq k_0}}^{N} \alpha_{k,i} \cdot (e_{k,i} \cdot y_k + f_{k,i} - r_i \cdot d_k) - d_{i,1} + r_i \cdot d_{i,2}$, but since all $d_k = 0$ we have $\alpha_{k_0,i} \cdot (f_{k_0,i} - r_i \cdot d_{k_0}) = \alpha_{k_0,i} \cdot f_{k_0,i}$ and so $\mathsf{val}([T_i])_H = 0$ if and only if $\alpha_{k_0,i} = \Big(-\sum_{\substack{k=1 \\ k \neq k_0}}^{N} \alpha_{k,i} \cdot (e_{k,i} \cdot y_k + f_{k,i} - r_i \cdot d_k) - d_{i,1} + r_i \cdot d_{i,2}\Big) \cdot {f_{k_0,i}}^{-1}$. As in the previous case, by the uniform choice of $\alpha_{k_0,i}$, this holds in a real execution with probability $1/|\mathbb{F}|$ exactly.
3. *Case 3:* In this case, all $g_{m,i} = 0$, and all $f_{k,i} - r_i \cdot d_k = 0$. If this occurs since all $f_{k,i} = 0$ and all $d_k = 0$, then clearly $T_i = d_{i,1} - r_i \cdot d_{i,2}$ since $\mathcal{A}$ did not cheat during the circuit emulation step. Otherwise, assume that for all $f_{k,i}, d_k \neq 0$ it holds that $f_{k,i} - r_i \cdot d_k = 0$. The computation of multiplication gate G_k involves two calls to $\mathcal{F}_{\mathsf{mult}}$: one with x_k and y_k, and the other with $r_i \cdot x_k$ and y_k. By the definition of $\mathcal{F}_{\mathsf{mult}}$ and the values $d_k, f_{k,i}$, the output of the first call to $\mathcal{F}_{\mathsf{mult}}$ is $z_k = x_k \cdot y_k + d_k$, and the output of the second call to $\mathcal{F}_{\mathsf{mult}}$ is $z'_k = r_i \cdot x_k \cdot y_k + f_{k,i}$. Writing $x_k \cdot y_k = z_k - d_k$, we have that $z'_k = r_i \cdot (z_k - d_k) + f_{k,i} = r_i \cdot z_k - r_i \cdot d_k + f_{k,i}$. However, by this case assumption, $f_{k,i} - r_i \cdot d_k = 0$ and so $z'_k = r_i \cdot z_k$. This means that the invariant of the relation between the real and randomized values on the wires is maintained, and formally that the kth term in the sum for T_i equals zero. Since this holds for all $k \in \{1, \ldots, N\}$, we have that $T_i = d_{i,1} - r_i \cdot d_{i,2}$ with probability 1, as determined by the simulator. (We remark that there is no accumulated error $e_{k,i}$ in this case, since $e_{k,i}$ appears when the invariant on the wires is *not* preserved.)
4. *Case 4:* Let k_0 be the first $k \in \{1, \ldots, N\}$ for which $f_{k,i} - r_i \cdot d_k \neq 0$. Since this is the first such k, it holds that $e_{k_0,i} = 0$ (note that some previous $d_k, f_{k,i}$ may

be non-zero, but as we saw in the previous case, if $f_{k,i} - r_i \cdot d_k = 0$ then there is no accumulated error). As in case 2, we have that $\mathsf{val}([T_i])_H = 0$ if and only if $\alpha_{k_0,i} = \left(-\sum_{\substack{k=1 \\ k \neq k_0}}^{N} \alpha_{k,i} \cdot (e_{k,i} \cdot y_k + f_{k,i} - r_i \cdot d_k) - d_{i,1} + r_i \cdot d_{i,2} \right) \cdot (f_{k_0,i} - r_i \cdot d_{k_0})^{-1}$. where division by $f_{k_0,i} - r_i \cdot d_{k_0}$ is possible since this value is non-zero. As above, this equality holds with probability exactly $1/|\mathbb{F}|$, by the uniform choice of $\alpha_{k_0,i}$.

The above demonstrates that the simulation by $\mathcal{S}$ of the zero/non-zero value of T_i is identical to a real execution. Furthermore, since the actual values of $\alpha_{k,i}, \beta_{m,i}$ are never revealed in this protocol, the simulation only requires that the probability that T_i is zero/non-zero be the same as in a real execution.[5]

It remains to show that $\mathcal{S}$ outputs fail with probability at most $\left(\frac{3}{|\mathbb{F}|}\right)^{\delta}$, which is at most $2^{-\sigma}$ by the choice of δ in the protocol. Recall that $\mathcal{S}$ outputs fail if and only if there exists some $d_k \neq 0$ and yet all $\mathcal{F}_{\text{checkZero}}$ invocations return accept in the simulation. This is indeed a fail, since the outputs received by the honest parties in the real and ideal executions in this case would be different. Now, assume that $d_k \neq 0$ for some $k \in \{1, \ldots, N\}$. Then, for *every* i, the simulation case is either Case 3 or Case 4, where the actual case depends on the value of r_i chosen. $\mathcal{F}_{\text{checkZero}}$ returns accept in the ith invocation in the simulation if either **(a)** case 3 occurs, meaning that $f_{k,i} - r_i \cdot d_k = 0$ which is equivalent to $r_i = f_{k,i}/d_k$, or **(b)** case 4 occurs and $\alpha_{k,i}$ results in $T_i = 0$, or **(c)** Case 4 occurs and $T_i \neq 0$ but $\mathcal{F}_{\text{checkZero}}$ returns accept nevertheless. The probability that accept is received from $\mathcal{F}_{\text{checkZero}}$ for any given i equals the probability that one of **(a)**, **(b)** or **(c)** occur. Each one independently occurs with probability $1/|\mathbb{F}|$: **(a)** because of the random choice of r_i, **(b)** because of the random choice of $\alpha_{k,i}$, and **(c)** because of the $1/|\mathbb{F}|$ probability that $\mathcal{F}_{\text{checkZero}}$ returns accept on non-zero input. By the union bound, the probability that *one* of these occur is therefore upper bound by $3/|\mathbb{F}|$. We conclude by noting that the above holds *independently* for each $i \in \{1, \ldots, \delta\}$, and thus the probability that $\mathcal{F}_{\text{checkZero}}$ returns accept for *all* $i \in \{1, \ldots, \delta\}$ is upper bound by $(3/|\mathbb{F}|)^{\delta}$, as required. ■

Concrete efficiency. We analyze the performance of our protocol. The main difference compared to Protocol 4.1 is that functionality $\mathcal{F}_{\mathsf{mult}}$ is called δ times for every input wire and $1+\delta$ for every multiplication gate (once for multiplying $[x]$ and $[y]$, and δ additional times for multiplying $[r_i \cdot x]$ with $[y]$). Thus, the overall number of multiplications is $(\delta \cdot M + (1+\delta) \cdot N) \cdot \mathcal{F}_{\mathsf{mult}}$, where M denotes the number of inputs and N the number of multiplication gates. Another difference is that now there are δ calls to $\mathcal{F}_{\text{rand}}$ for generating $[r_i]$, and $\delta \cdot (M+N)$ calls for

[5] If $\beta_{m_0,i}$ were to be revealed, as in Protocol 4.1 for large fields, then the question of whether the equation holds is something that the distinguisher could determine (since it knows all of the y_k values from the input, and it can receive all of the $d_k, e_{k,i}, f_{k,i}, g_{m,i}$ values from the adversary). Thus, it would *not* suffice to set $T_i = 0$ with the correct probability but as a function of the actual values. However, $\mathcal{S}$ does not know the y_k values and so could not determine this.

generating all the $\alpha_{k,i}, \beta_{m,i}$ values (which are secret here, unlike in Protocol 4.1). In addition, there are 2δ calls to $\mathcal{F}_{\mathsf{product}}$, δ calls to open for $[r_i]$, δ calls to $\mathcal{F}_{\text{checkZero}}$ (each of which reduces to one call to $\mathcal{F}_{\text{rand}}$, one $\mathcal{F}_{\mathsf{mult}}$ and one opening), and $M + L$ calls to $\mathsf{reconstruct}$ as part of $\mathcal{F}_{\text{input}}$ and obtaining output (where L equals the number of output wires). Assuming that $\mathcal{F}_{\mathsf{product}}$ is equivalent to $\mathcal{F}_{\mathsf{mult}}$ (as will be shown in Section 6.2), we have that the overall exact cost of the protocol is

$$(\delta \cdot M + (1+\delta) \cdot N + 3\delta) \cdot \mathcal{F}_{\mathsf{mult}} + (\delta \cdot (M+N) + 2\delta) \cdot \mathcal{F}_{\text{rand}} + (M+L) \cdot \mathsf{reconstruct} + 2\delta \cdot \mathsf{open}.$$

Amortizing over the size of the circuit, we have that the average cost is $(1+\delta) \cdot \mathcal{F}_{\mathsf{mult}} + \delta \cdot \mathcal{F}_{\text{rand}}$ per multiplication gate.

We compare now the cost of running Protocol 5.3 with $\delta = 1$ to the cost of running Protocol 4.1 for large fields. The amortized cost of Protocol 4.1 is $2 \cdot \mathcal{F}_{\mathsf{mult}}$ per multiplication gate, whereas the cost of Protocol 5.3 with $\delta = 1$ is $2 \cdot \mathcal{F}_{\mathsf{mult}} + 1 \cdot \mathcal{F}_{\text{rand}}$. Thus, the difference between these protocols depends on the cost of $\mathcal{F}_{\text{rand}}$. As we will see in Section 6.2, the cost of $\mathcal{F}_{\text{rand}}$ for our specific instantiation for a not-small number of parties is about a third of the cost of $\mathcal{F}_{\mathsf{mult}}$, making Protocol 5.3 about 17% slower.

It is also instructive to compare the cost of running Protocol 4.1 with a large field versus running Protocol 5.3 with a smaller field. Concretely, assume that the computation being carried out is over the integers, and that all values are smaller than 2^{30}, and that security 2^{-60} is desired. Then, the question that may arise is whether one should run Protocol 4.1 over a 60-bit field, or whether one should run Protocol 5.3 with $\delta = 2$ over a 30-bit field. The amortized cost is $2 \cdot \mathcal{F}_{\mathsf{mult}}$ for Protocol 4.1 versus $3 \cdot \mathcal{F}_{\mathsf{mult}} + 1 \cdot 2 \cdot \mathcal{F}_{\text{rand}} \approx 3.66 \cdot \mathcal{F}_{\mathsf{mult}}$ for Protocol 5.3 (assuming the cost of $\mathcal{F}_{\text{rand}}$ to be one-third of $\mathcal{F}_{\mathsf{mult}}$). Clearly, the communication cost is double for a 60-bit field, and so the expected communication using Protocol 5.3 is lower in such a case. Regarding computation, empirical experimentation is needed to make a comparison.

Reducing memory. As in Protocol 4.1, when the circuit is huge, it is highly undesirable to store all values until completion in order to carry out the verification. Thus, in such cases, it is preferable to compute the verification while evaluating the circuit. However, Protocol 4.1 required running a full verification at intermediate steps to do this, and this incurred additional work to rerandomize the active wires for the next phase, and so on (see the discussion at the end of Section 4). In contrast, Protocol 5.3 is much more amenable to verification-on-the-fly because the α, β values are never revealed. Thus, it is possible to call $\mathcal{F}_{\text{rand}}$ to obtain the $[\beta_{m,i}]$ shares at the input phase, and to call $\mathcal{F}_{\text{rand}}$ to obtain $[\alpha_{k,i}]$ shares during multiplications. Then, the parties can locally store the partial sums for u_i and w_i, and all previous shares that are no longer needed for the circuit evaluation can be discarded. This method for verification-on-the-fly is also very easy to implement.

Reactive computation. In Protocol 4.1 where the α, β values are public, it is necessary to open $[r]$ in order to compute $[T] = [u] - r \cdot [w]$. This is because otherwise the adversary can input an additive value in the multiplication $[r] \cdot [w]$

that can cancel out a previous error (note that at this stage, α, β are already known and so the adversary has enough information to make the errors cancel). In contrast, in Protocol 5.3, the α, β values are never revealed. Thus, it is not necessary to open the $[r_i]$ shares, and the parties can compute $[T_i] = [u_i] - [r_i] \cdot [w_i]$, using $\mathcal{F}_{\mathsf{mult}}$ with $[r_i]$ and $[w_i]$. In a regular one-off computation, this makes no real difference. However, in the case of reactive computation, where outputs are revealed, and the computation continues, it is undesirable to open the $[r_i]$ shares, since new randomization is necessary. Thus, in such cases, one can leave the $[r_i]$ shares secret, and compute $[T_i]$ using $\mathcal{F}_{\mathsf{mult}}$ as described.

6 Instantiations and Experimental Results

Our protocol is generic and can be instantiated in many ways (with different secret sharing schemes, multiplication protocols, and more). Clearly, the efficiency of our protocol depends significantly on the instantiations. In order to demonstrate the efficiency of our protocol, we plug in the instantiations presented in [22], which meet all of our requirements. We consider two secret sharing schemes: replicated secret sharing for 3 parties, and Shamir sharing [25] for any number of parties. Recall that our protocol requires instantiations for functionalities $\mathcal{F}_{\mathsf{mult}}$ and $\mathcal{F}_{\mathrm{rand}}$, and for procedures open and reconstruct ($\mathcal{F}_{\mathrm{input}}$, $\mathcal{F}_{\mathrm{coin}}$ and $\mathcal{F}_{\mathrm{checkZero}}$ are constructed generically using these functionalities and procedures). We also need to show how to securely realize $\mathcal{F}_{\mathsf{product}}$ for Protocol 5.3; we begin by showing this in Section 6.1. Then, in Section 6.2 we present the concrete costs of the instantiations from [22] along with $\mathcal{F}_{\mathsf{product}}$ from Section 6.1. Finally, in Section 6.3, we present experimental results of the implementation of our protocol and compare it to prior work. In the full version of this paper, we describe the protocols for the instantiation based on the Shamir sharing, including proofs that the protocols securely compute $\mathcal{F}_{\mathrm{rand}}$ and $\mathcal{F}_{\mathsf{mult}}$.

6.1 Securely Realizing Functionality 5.1 – $\mathcal{F}_{\mathsf{product}}$

$\mathcal{F}_{\mathsf{product}}$ with Shamir secret sharing. We begin by describing how to securely realize $\mathcal{F}_{\mathsf{product}}$ when Shamir sharing is used. Let $[x_1], \ldots, [x_\ell]$ and $[y_1], \ldots, [y_\ell]$ be two vectors of inputs, where the parties wish to compute shares of $\sum_{i=1}^{\ell} x_i \cdot y_i$. The key observation here is that most (if not all) protocols for multiplication based on Shamir sharing have two phases:

1. *Local multiplication:* In this phase, each party locally multiplies its shares on the two values. This yields a sharing of the product of the two values on a degree-$2t$ polynomial. Since $t < n/2$, there is enough "information" to reconstruct the polynomial (since $2t < n$).
2. *Degree reduction:* In the second phase, the parties run an interactive protocol that reduces the degree of the polynomial generated in the previous step back to degree-t, without changing its constant term.

Observe that the protocols of [5, 18, 12] and others all follow this framework.

The crucial observation regarding how to compute $\mathcal{F}_{\mathsf{product}}$ is that the parties can begin by locally computing the sum of the products of their input shares. Specifically, denote P_j share of x_i and y_i by x_i^j and y_i^j, respectively. Then, each P_j can locally compute $z_j = \sum_{i=1}^{\ell} x_i^j \cdot y_i^j$, and the shares $z_1, \ldots, z_n$ constitute a sharing of degree-$2t$ polynomial with constant-term $\sum_{i=1}^{\ell} x_i \cdot y_i$. All that therefore remains is for the parties to run the *degree reduction* on these shares, and they obtain a good Shamir sharing of the sum of products.

The above strategy securely computes $\mathcal{F}_{\mathsf{product}}$ if the degree reduction phase of the protocol has the property that the only attack possible by the adversary is an additive attack. That is, if the input shares define a degree-$2t$ polynomial hiding the secret z, then the adversary can cause the parties to output a degree-t sharing of $z + d$ where d can be extracted by a simulator (exactly as in $\mathcal{F}_{\mathsf{mult}}$). In the full version of this paper, we show that this property holds for the semi-honest multiplication protocol of [12].

$\mathcal{F}_{\mathsf{product}}$ with replicated secret sharing. In the multiplication protocol of [2] which is also shown to be secure up to an additive attack in [22]), the parties first locally compute a sum of 3 products of their local shares (given replicated shares (s_i, s_{i+1}) and (t_i, t_{i+1}) of two values s and t held by P_i, each party computes $u_i = s_i \cdot t_i + s_{i+1} \cdot t_i + s_i \cdot t_{i+1}$). Then, in the next step, each party sends its share u_i – randomized using correlated randomness – to party P_{i+1}, who defines the pair (u_{i+1}, u_i) as its share of the output. The simple observation here is that if each party computes many u_i's for each product in the vector and then sums them all together, the result will be a replicated secret sharing of the entire sum of products.

Efficiency. Using the above method, the cost of a sum of products for *any number of terms* is local operations on the vector (similar to addition gates in the circuit) and interaction equivalent to a *single multiplication*. Thus, $\mathcal{F}_{\mathsf{product}}$ essentially costs the same as $\mathcal{F}_{\mathsf{mult}}$.

Applications. Beyond the use of $\mathcal{F}_{\mathsf{product}}$ in Protocol 5.3 for computing the random linear combinations, this subprotocol can be used to significantly speed up many secure statistical operations. For example, in order to securely computed the standard deviation over a large list, the main cost is computing the sum of squares (of the difference between each item and the mean), and then dividing by the length of the vector. Using our method, this can be carried out on millions of data items at the cost of a *single multiplication* followed by a *single division* (and if the number of data items is known, then the division can be carried out on the result).

6.2 Instantiations from [22] and their Cost

As we have discussed above, we present the cost of $\mathcal{F}_{\mathsf{mult}}$, $\mathcal{F}_{\text{rand}}$, open and reconstruct for Shamir sharing (for any number of parties n) and for replicated secret sharing (for 3 parties), as described in [22]. The communication costs are presented in Table 1. In the two instantiations we consider for Shamir sharing, $\mathcal{F}_{\text{rand}}$ can be instantiated using PRSS [10] which has zero communication cost but has computation that is exponential in the number of parties and so is only good for up to 7 or so parties (as shown in [22]). In addition, $\mathcal{F}_{\text{rand}}$ can be instantiated using VAN, which is the hyper-invertible matrices method of [4] that utilizes Vandermonde matrices. In both cases, Shamir sharing uses the DN multiplication protocol of [12].

	$\mathcal{F}_{\mathsf{mult}}$	$\mathcal{F}_{\mathbf{rand}}$	open	reconstruct
Replicated secret sharing (three parties)	1	0	4	2
Shamir sharing (few parties), $\mathcal{F}_{\text{rand}}$ with PRSS	6	0	$n-1$	1
Shamir sharing (many parties), $\mathcal{F}_{\text{rand}}$ with VAN	6	2	$n-1$	1

Table 1. The communication cost per party for instantiations in [22], written as the number of *field elements* sent.

Table 1 counts the communication costs of each protocol instantiation. The computational costs are low overall (since we use secret-sharing based primitives), except for PRSS which is exponential in the number of parties and thus only suitable for a small number. In Section 6.3, we show concrete running times for the replicated secret sharing and Shamir-sharing with VAN instantiations.

Overall protocol costs. As shown in Sections 4 and 5, the cost per multiplication gate of Protocol 4.1 is $2 \cdot \mathcal{F}_{\mathsf{mult}}$, and the cost per multiplication gate of Protocol 5.3 is $(1+\delta) \cdot \mathcal{F}_{\mathsf{mult}} + \delta \cdot \mathcal{F}_{\text{rand}}$. Plugging these into the above instantiations, we obtain a maliciously secured protcol for three-parties that requires each party to send only *2 field elements per multiplication gate* when the filed is large. For the multi-party setting, we obtain a protocol with a communication cost of only *12 field elements per multiplication gate* for each party when the field is large. This is shown in Table 2, including a comparison to the cost of the protocol of [22].

	Protocol of [22] with $\delta = 1$	**Protocol 4.1 (large field)**	**Protocol 5.3 with $\delta = 1$**	**Protocol 5.3 with $\delta = 2$**
Replicated secret sharing (three parties)	4	2	2	3
Shamir (few parties), $\mathcal{F}_{\text{rand}}$ with PRSS	36	12	12	18
Shamir (many parties), $\mathcal{F}_{\text{rand}}$ with VAN	42	12	14	22

Table 2. The communication cost per party for the instantiations in Table 1 and the protocol of [22], written as the number of *field elements* sent *per multiplication gate.* (Note that Protocol 5.3 with $\delta = 2$ has smaller field elements and thus more elements sent could actually mean less bandwidth.)

6.3 Experimental Results

We implemented Protocol 4.1 with two instantiations: replicated secret sharing for 3 parties and Shamir sharing using VAN for $\mathcal{F}_{\text{rand}}$ and DN [12] for $\mathcal{F}_{\mathsf{mult}}$ (see

Section 6.2). The field we used for all our experiments was the 61-bit Mersenne field (and so security is approximately 2^{-60}). We ran our protocols for different numbers of parties on a series of circuits of different depths, each with 1,000,000 multiplication gates, 1,000 inputs wires, 50 output wires. The circuits had 4 different depths: 20, 100, 1,000 and 10,000. The experiment was run on AWS in two configurations: a *LAN network configuration* in a single AWS region (specifically, North Virginia), and a *WAN network configuration* in three AWS regions (specifcally, North Virginia, Germany and India). Each party was run in an independent AWS `C4.large` instance (2-core Intel Xeon E5-2666 v3 with 2.9 GHz clock speed and 3.75 GB RAM). Each execution (configuration, number of parties, circuit) was run 5 times, and the result reported is the average run-time.

Circuit Depth	**3 (replicated)**	**3**	**5**	**7**	**9**	**11**	**30**	**50**	**70**	**90**	**110**
20	319	826	844	1,058	1,311	1,377	2,769	4,053	5,295	6,586	8,281
100	323	842	989	1,154	1,410	1,477	3,760	6,052	8,106	11,457	15,431
1,000	424	1,340	1,704	1,851	2,243	2,887	12,144	26,310	33,294	48,927	79,728
10,000	1,631	6,883	7,424	8,504	12,238	16,394	61,856	132,160	296,047	411,195	544,525

Table 3. *LAN configuration* execution times in milliseconds of a circuit with 1,000,000 multiplication gates, for different depths. The first column gives the running time for the replicated secret sharing version; all other columns are the Shamir sharing for different numbers of parties.

	3 (replicated)	**3**	**5**	**7**	**9**	**11**	**30**	**50**	**70**	**90**	**110**
Protocol 4.1	319	826	844	1,058	1,311	1,377	2,769	4,053	5,295	6,586	8,281
Protocol of [22]	513	1,229	1,890	3,056	4,009	5,187	15,954	28,978	44,599	58,966	72,096
Speedup	161%	149%	224%	289%	306%	377%	576%	715%	842%	895%	871%

Table 4. *LAN configuration* execution times in milliseconds of a circuit with 1,000,000 multiplication gates and depth 20. The times for [22] are for the best protocol for the number of parties.

In order to compare our protocol to that of [22], we compare the running times in a LAN configuration for depth 20 (this is because that is the only configuration run by them); see Table 4.

As can be seen, our protocol outperforms the best protocol of [22] significantly, even for a small number of parties. However, as the number of parties increases, the gap widens. Observe that the communication difference between the protocols, as shown in Table 2 would only predict that our protocol would run 3 times faster than that of [22], whereas experiment yield an almost 10 times faster result for a large number of parties. This may be due to additional computational work involved in generating the Beaver triples in [22].

Finally, in Table 5, we present the experimental results of running our protocol in the WAN configuration. Due to the many rounds of communication, the results are significantly slower, but demonstrate that it is even possible to run for quite a large number of parties (e.g., 50 parties) with reasonable time.

Circuit Depth	**3 (replicated)**	**3**	**5**	**7**	**9**	**11**	**30**	**50**
20	3502	20,492	27,772	28,955	24,482	24,729	87,355	128,366
100	10,712	45,250	53,872	50,719	55,716	56,482	134,860	197,321

Table 5. *WAN configuration* (North Virginia, Germany and India) execution times in milliseconds of a circuit with 1,000,000 multiplication gates, for different depths.

References

1. T. Araki, A. Barak, J. Furukawa, T. Lichter, Y. Lindell, A. Nof, K. Ohara, A. Watzman and O. Weinstein. Optimized Honest-Majority MPC for Malicious Adversaries - Breaking the 1 Billion-Gate Per Second Barrier. In the *IEEE S&P*, 2017.
2. T. Araki, J. Furukawa, Y. Lindell, A. Nof and K. Ohara. High-Throughput Semi-Honest Secure Three-Party Computation with an Honest Majority. In the 23*rd ACM CCS*, pages 805–817, 2016.
3. D. Beaver. Foundations of Secure Interactive Computing. In *CRYPTO'91*, Springer (LNCS 576), pages 377–391, 1991.
4. Z. Beerliová-Trubíniová and M. Hirt. Perfectly-secure MPC with linear communication complexity. In *TCC 2008*, Springer (LNCS 4948), pages 213–230, 2008.
5. M. Ben-Or, S. Goldwasser and A. Wigderson. Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation. In 20*th STOC,* 1988.
6. 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. *ePrint Cryptology Archive*, 2015/472.
7. R. Canetti. Security and Composition of Multiparty Cryptographic Protocols. *Journal of Cryptology*, 13(1):143–202, 2000.
Protocols.
8. D. Chaum, C. Crépeau and I. Damgård. Multi-party Unconditionally Secure Protocols. In 20*th STOC*, pages 11–19, 1988.
9. R. Cleve. Limits on the Security of Coin Flips when Half the Processors are Faulty. In 18*th STOC,* pages 364–369, 1986.
10. R. Cramer, I. Damgård and Y. Ishai, Share Conversion, Pseudorandom Secret-Sharing and Applications to Secure Computation. In *TCC*, Springer (LNCS 3378) pages 342–362, 2005.
11. 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 *18th ESORICS*, pages 1–18, 2013.
12. I. Damgård and J. Nielsen. Scalable and unconditionally secure multiparty computation. In *CRYPTO 2007*, Springer (LNCS 4622), pages 572–590, 2007.
13. I. Damgård, V. Pastro, N.P. Smart and S. Zakarias. Multiparty Computation from Somewhat Homomorphic Encryption. In *CRYPTO 2012*, pages 643–662, 2012.
14. D. Genkin, Y. Ishai, M. Prabhakaran, A. Sahai and E. Tromer. Circuits Resilient to Additive Attacks with Applications to Secure Computation. In *STOC 2014*, 2014.

15. D. Genkin, Y. Ishai and A. Polychroniadou. Efficient Multi-party Computation: From Passive to Active Security via Secure SIMD Circuits. In *CRYPTO 2015*.
16. O. Goldreich, S. Micali, and A. Wigderson. How to Play Any Mental Game. In 19*th STOC*, pages 218–229, 1987.
17. S. Goldwasser and L. Levin. Fair Computation of General Functions in Presence of Immoral Majority. In *CRYPTO'90,* Springer (LNCS 537), pages 77–93, 1990.
18. R. Gennaro, M. Rabin and T. Rabin. Simplified VSS and Fact-Track Multiparty Computations with Applications to Threshold Cryptography. In 17*th PODC*, 1998.
19. O. Goldreich. *Foundations of Cryptography: Volume 2 – Basic Applications*, 2004.
20. M. Keller, E. Orsini and P. Scholl. MASCOT: Faster Malicious Arithmetic Secure Computation with Oblivious Transfer. In *23rd ACM CCS*, pages 830–842, 2016.
21. M. Keller, V. Pastro and D. Rotaru. Overdrive: Making SPDZ Great Again. In *EUROCRYPT*, Springer (LNCS 10822(3)), pages 158-189, 2018.
22. Y. Lindell and A. Nof. A Framework for Constructing Fast MPC over Arithmetic Circuits with Malicious Adversaries and an Honest-Majority. In *ACM CCS*, 2017.
23. P. Mohassel, M. Rosulek and Y. Zhang. Fast and Secure Three-party Computation: The Garbled Circuit Approach. In *ACM CCS*, pages 591–602, 2015.
24. T. Rabin and M. Ben-Or. Verifiable Secret Sharing and Multi-party Protocols with Honest Majority. In 21*st STOC*, pages 73–85, 1989.
25. A. Shamir. How to share a secret. CACM, 22(11), pages 612–613, 1979.
26. A. Yao. How to Generate and Exchange Secrets. 27*th FOCS*, pages 162–167, 1986.