High-Throughput Secure Three-Party Computation for Malicious Adversaries and an Honest Majority

Jun Furukawa[1], Yehuda Lindell[2], Ariel Nof[2], and Or Weinstein[2]

[1] NEC Corporation, Japan.
j-furukawa@ay.jp.nec.com

[2] Bar-Ilan University, Israel*
lindell@biu.ac.il, nofdinar@google.com, oror.wn@gmail.com

Abstract. In this paper, we describe a new protocol for secure three-party computation of any functionality, with an honest majority and a *malicious* adversary. Our protocol has both an information-theoretic and computational variant, and is distinguished by extremely low communication complexity and very simple computation. We start from the recent semi-honest protocol of Araki et al. (ACM CCS 2016) in which the parties communicate only a single bit per AND gate, and modify it to be secure in the presence of malicious adversaries. Our protocol follows the paradigm of first constructing Beaver multiplication triples and then using them to verify that circuit gates are correctly computed. As in previous work (e.g., the so-called TinyOT and SPDZ protocols), we rely on the cut-and-choose paradigm to verify that triples are correctly constructed. We are able to utilize the fact that at most one of three parties is corrupted in order to construct an extremely simple and efficient method of constructing such triples. We also present an improved combinatorial analysis for this cut-and-choose which can be used to achieve improvements in other protocols using this approach.

1 Introduction

1.1 Background

In the setting of secure computation, a set of parties with private inputs wish to compute a joint function of their inputs, without revealing anything but the output. Protocols for secure computation guarantee *privacy* (meaning that the protocol reveals nothing but the output), *correctness* (meaning that the correct function is computed), and more. These security guarantees are provided in the presence of adversarial behavior. There are two classic adversary models that are typically considered: *semi-honest* (where the adversary follows the protocol specification but may try to learn more than allowed from the protocol transcript)

* 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.

and *malicious* (where the adversary can run any arbitrary polynomial-time attack strategy). In the *information-theoretic* model, security is obtained unconditionally and even in the presence of computationally unbounded adversaries. In contrast, in the *computational* model, security is obtained in the presence of polynomial-time adversaries and relies on cryptographic hardness assumptions.

Despite its stringent requirements, it has been shown that any polynomial-time functionality can be securely computed with computational security [22, 12, 3] and with information-theoretic security [4, 8, 19]. These results hold both for semi-honest and malicious adversaries, but a two-thirds honest majority must be assumed in order to obtain information-theoretic security (or an honest majority when assuming broadcast).

There are two main approaches for constructing secure computation protocols: the secret-sharing approach (followed by [4, 8, 12]) works by having the parties interact for every gate of the circuit, whereas the garbled-circuit approach (followed by [22, 3]) works by having the parties construct an encrypted version of the circuit which can be computed at once. Both approaches have importance and have settings where they perform better than the other. On the one hand, the garbled-circuit approach yields protocols with a constant number of rounds. Thus, in high-latency networks, they far outperform secret-sharing based protocols which have a number of rounds that is linear in the depth of the circuit being computed. On the other hand, protocols based on secret-sharing typically have low bandwidth, in contrast to garbled circuits that are large and costly in bandwidth. Given that the bandwidth is often the bottleneck, it follows that protocols with low communication have the potential to achieve much higher throughput.

1.2 Our Results

In this paper, we focus on the question of achieving secure computation in the presence of malicious adversaries with *very high throughput* on a *fast network* (without utilizing special-purpose hardware). We start with the recent *three-party* protocol of [1] that achieves security in the presence of semi-honest adversaries. The protocol requires transmitting only a single bit per AND gate, and the computation per gate is very simple. On a cluster of three 20-core servers with a 10Gbs connection, the protocol of [1] achieves a rate of computation of *7 billion AND gates per second*. This can be used, for example, to securely compute 1.3 million AES block operations per second.

Our approach to achieving malicious security follows the Beaver multiplication triple approach [2] used in [18, 9, 5] (and many follow-up works). According to this approach, the parties securely generate shares of triples (a, b, c) where a, b are random and $c = ab$ (for the case of Boolean circuits, this is equivalent to $c = a \wedge b$). Such triples can then be used to *verify* that AND gates are computed correctly. In the (difficult) case of no honest majority considered in [18, 9, 5], there are two major challenges: **(a)** how to generate such triples without malicious parties causing either the shares to be invalid or causing $c \neq ab$, and **(b)** how to force the parties to send their "correct" values in the multiplication

triple in the verification stage. The first problem is solved in [18, 9, 5] by using cut-and-choose: many triples are generated, some are opened, and the others are put in buckets and used to verify each other. The second problem is solved in [18, 9, 5] by using homomorphic MACs on all of the values. The generation of the triples to start with and the use of MACs adds additional overhead that is very expensive.

In this paper, we heavily utilize the fact that at most one party (out of 3) is corrupted in order to generate triples at very little expense, and to force the parties to send the correct values. In fact, the secret-sharing method used in [1] is such that it is possible to generate shares of random values *without any interaction* (using correlated randomness which is generated by the parties at almost no cost). Furthermore, we show how it is possible to detect if a malicious party sends incorrect values (in the prepared multiplication triples) when there is an honest majority, without requiring MACs of any kind. As a result, generating multiplication triples is very cheap. In turn, this enables us to generate a large number of triples at once, which further improves the parameters of the cut-and-choose step as well.

Overall, our protocol requires very simple computation, and achieves malicious security at very low communication cost. Specifically, with a statistical error of 2^{-40} each party needs to send only 10 bits per AND gate to one other party; for 2^{-80} this rises to only 16 bits per AND gate.

Based on the implementation results in [1], our estimates are that our new protocol should achieve a rate of over 500 million AND gates per second on the same setup as [1]. This is orders of magnitude faster than any other protocol achieving malicious security (see related work below).

1.3 Outline of Our Solution and Organization

In this section, we describe of the different subprotocols and constructions that make up our protocol, and provide the high-level ideas behind our constructions.

In Section 2.1, we present the 2-out-of-3 secret-sharing scheme used in [1] and some important properties of it. Then, in Section 2.2 we describe the semi-honest protocol of [1] for multiplication (AND) gates. In addition, we prove a crucial property of this protocol that we heavily rely on in our construction: for any malicious adversary, the honest parties always hold a *valid sharing* after the multiplication protocol; the shared value may either equal the AND of the input (if the adversary follows the protocol) or its complement (if the adversary cheats).

In Section 2.3 we show how to generate correlated randomness (functionality $\mathcal{F}_{\mathrm{cr}}$); after an initial exchange of keys for a pseudorandom function, the protocol is non-interactive. This makes it highly efficient, and also secure for malicious adversaries (since there is no interaction at all and so no way to cheat). In Section 2.4, we use $\mathcal{F}_{\mathrm{cr}}$ to secure compute functionality $\mathcal{F}_{\mathrm{rand}}$ that provides random shares to all parties. A very important feature of our protocol is based on the fact that $\mathcal{F}_{\mathrm{rand}}$ can be securely computed *non-interactively* using correlated randomness. This means that the first step in generating multiplication triples

– generating shares of random a, b via calls to $\mathcal{F}_{\text{rand}}$ – can be carried out non-interactively and thus at a very fast rate.

In Section 2.5, we use $\mathcal{F}_{\text{rand}}$ to carry out secure coin tossing (by generating shares of random values and then just opening them); each coin is generated by sending just a single bit. We explain how this is achieved, since it introduces a key technique that we use throughout. As we have mentioned, shares of a random value can be generated non-interactively and thus this is secure for malicious adversaries by default. However, when opening the shares to obtain the coin, a malicious adversary can cheat by sending an incorrect share value. Here we critically utilize the fact that we have an *honest majority*. In particular, we can simply have all pairs of parties send their shares to each other. Since the sharing is valid and any two parties can reconstruct the secret, each party can reconstruct separately based on the shares received from each other party, and compare. If the adversary cheats, then the result will be *different* reconstructed secrets, which will result in an abort. Our secret-sharing scheme has two bits and so this would cost each party sending 4 bits. However, we observe that in order to open it suffices to send 1 bit of each share only. Furthermore, we observe that if each party sends its bit to only one other party (P_1 to P_2, P_2 to P_3, and P_3 to P_1) then the bit sent by one honest to another will result in the correct coin (there is always one such pair since only one party is corrupted). Thus, it actually suffices for each party to send its bit to only one other party and to record the result of the coin on a "public view" string. Then, at the end of the entire execution, before any output is revealed, the parties can compare their views by sending a collision-resistant hash of their local public view. If the two honest parties received a different coin at any point then they will have different local public views and so will abort before anything is revealed. As a result, coin-tossing can be achieved by each party sending just a single bit to one other party.

In Sections 2.6– 2.8, we introduce additional functionalities needed for our protocol. First, in order to carry out the cut-and-choose, a random permutation must be applied to the tuples generated. This is carried out using $\mathcal{F}_{\text{perm}}$ (Section 2.6) which computes a random permutation of array indices. This functionality is easily realized by the parties just coin tossing the amount of randomness needed to define the permutation. In addition, in order for the parties to share inputs and obtain output, we need a way to deal shares and open shares that is secure for malicious adversaries. These are constructed in Section 2.7 ($\mathcal{F}_{\text{reconst}}$ for robustly reconstructing a secret to one party) and Section 2.8 ($\mathcal{F}_{\text{share}}$ for robustly sharing a value).

We now explain how the above subprotocols can be used to generate correct multiplication triples. The parties first call $\mathcal{F}_{\text{rand}}$ to generate shares of random values a, b and then run the semi-honest multiplication protocol of [1] to generate shares of c. As we have mentioned above, the semi-honest multiplication protocol has the property that even if the adversary is malicious, the shares of c are valid. However, if the adversary cheats then it may be the case that $c = ab \oplus 1$ instead of equalling ab. In order to prevent this from happening, the parties

generate many triples and use some to check the others. Namely, the parties first randomly choose a subset of the triples which are opened to verify that indeed $c = ab$. This uses the subprotocol in Section 2.9 which carries out this exact check. Next, the remaining triples are partitioned randomly into buckets of size B (the random division is carried out using $\mathcal{F}_{\text{perm}}$). Then, in each bucket, $B - 1$ of the triples are used to verify that one of the triples is correct, except with negligible probability (without revealing the triple being checked). This uses the subprotocol of Section 2.10 which shows how to use one triple to verify that another is correct. This protocol is described in Section 3, and it securely computes functionality $\mathcal{F}_{\text{triples}}$ that generates an array of random multiplication triples for the parties.

Finally, we show how to securely compute any functionality f using random multiplication triples. Intuitively, this works by the parties running the semi-honest multiplication protocol for each AND gate and verifying each multiplication using a triple. The verification method, as used in [18, 9], has the property that if a multiplication triple is good and the adversary cheats in the gate multiplication, then this is detected by the honest parties. As with all of our protocols, we take care to minimize the communication, and verify each gate by sending only 2 bits (beyond the single bit needed for the multiplication itself). The dependency graph of our constructions appears in Figure 1.

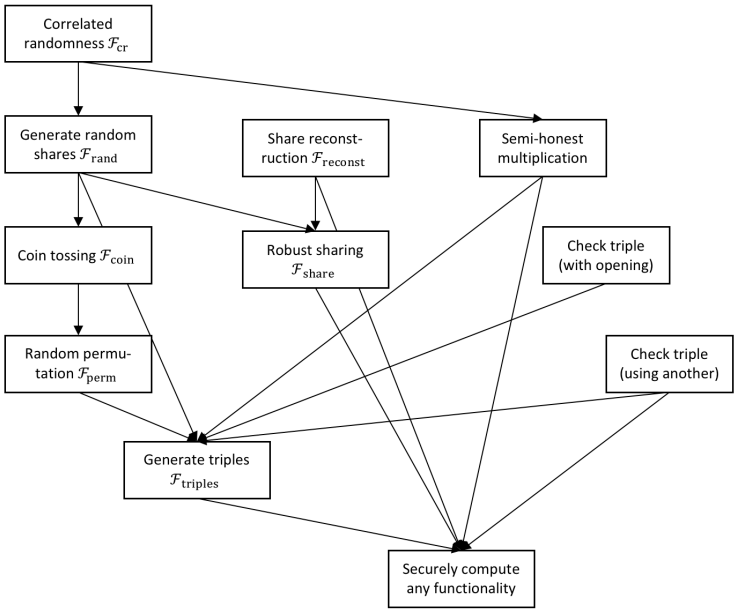

Fig. 1. Graph of dependencies of our constructions

The efficiency of our construction relies heavily on the cut-and-choose parameters, both with respect to how many triples need to be opened and checked and the bucket size. In Section 5 we provide a tight analysis of this cut-and-choose game which yields a significant improvement over previous analyses fo similar

games in [18, 5]. For concrete parameters that are suitable for our protocol, our analysis is approximately 25% better than [18, 5].

Caveats. We stress that our protocol is specifically defined for the case of 3 parties only. This case is of interest for outsourced computations, as in the Sharemind business model [21], for two-party setting where a third auxiliary server can be used, and in other settings of interest as described in [1]. We leave the generalization of the protocol to more parties to future work. Our protocol is also secure *with abort*; this is unlike other protocols for the honest majority case that achieve fairness. Nevertheless, this is sufficient for many applications. For this setting, we are able to achieve security for *malicious adversaries* with efficiency way beyond any other known protocol.

1.4 Related Work

Most of the work on concretely-efficient secure computation has focused on the dishonest majority case. These protocols are orders of magnitude less efficient than ours, but deal with a much more difficult setting. For example, the best protocols based on garbled circuits for batch executions [16, 20] require only sending 4 garbled circuits per execution. Even ignoring all of the additional work and communication (which is very significant), 4 garbled circuits per execution means sending 1000 bits per gate, which is 100 times the cost of our protocol. The same is true for all other dishonest-majority protocols; c.f. [18, 9, 5].

In the setting of an honest majority, the only highly-efficient protocol with security for malicious adversaries that has been implemented, to the best of our knowledge, is that of [17]. We compare our protocol to [17] in detail in Section 6. Our protocol is more than an order of magnitude cheaper both in communication and computation; however, their protocol is constant-round and therefore better suited to slow networks.

1.5 Definition of Security

Our protocols are proven secure under the standard ideal/real simulation paradigm, for the case of malicious adversaries and with abort. For the sake of completeness, this definition is provided in the appendix.

2 Building Blocks and Subprotocols

2.1 The Secret Sharing Scheme

We denote the three parties by P_1, P_2 and P_3. Throughout the paper, in order to simplify notation, when we use an index (say i) to denote the ith party, we will write $i-1$ and $i+1$ to mean the "previous" and "subsequent" party, respectively. That is, when $i = 1$ then P_{i-1} is P_3 and when $i = 3$ then P_{i+1} is P_1.

We use the 2-out-of-3 secret sharing scheme of [1], defined as follows. In order to share a bit v, the dealer chooses three random bits $t_1, t_2, t_3 \in \{0, 1\}$ under the constraint that $t_1 \oplus t_2 \oplus t_3 = 0$. Then:

- P_1's share is the pair (t_1, s_1) where $s_1 = t_3 \oplus v$.
- P_2's share is the pair (t_2, s_2) where $s_2 = t_1 \oplus v$.
- P_3's share is the pair (t_3, s_3) and $s_3 = t_2 \oplus v$.

It is clear that no single party's share reveals anything about v. In addition, any two parties can obtain v; e.g., given $(t_1, s_1), (t_2, s_2)$ one can compute $v = t_1 \oplus s_2$. We denote by $[v]$ a 2-out-of-3 sharing of the value v according to the above scheme.

Claim 2.1 *The secret v together with the share of one party fully determine the shares of the other parties.*

Proof: By the definition of the secret sharing scheme, it holds that $t_{i-1} = s_i \oplus v$. Since (t_i, s_i) for some $i \in \{1, 2, 3\}$ and v are determined, this determines t_{i-1} as well. Thus, t_i and t_{i-1} are determined. Utilizing the fact that $t_1 \oplus t_2 \oplus t_3 = 0$, this determines $t_{i+1} = t_i \oplus t_{i-1}$. Finally, for every j we have that $s_{j-1} = t_j \oplus v$ and thus s_1, s_2, s_3 are also determined. ■

Opening shares. Based on the definition of the secret sharing scheme, we define a subprocedure denoted $\mathsf{open}([v])$ as follows. Denote the shares of v by $\left\{(t_i, s_i)\right\}_{i=1}^{i=3}$. Then, each party P_i sends t_i to P_{i+1}, and each P_i outputs $v = s_i \oplus t_{i-1}$.

Local operators for shares. We define the following local *operators* on shares:

- *Addition* $[v_1] \oplus [v_2]$*:* Given a share (t_i^1, s_i^1) of v_1 and a share (t_i^2, s_i^2) of v_2, each party P_i computes: $(t_i^1 \oplus t_i^2, s_i^1 \oplus s_i^2)$.
- *Multiplication by a scalar* $\sigma \cdot [v]$*:* Given a share (t_i, s_i) of v and a value $\sigma \in \{0, 1\}$, each party P_i computes $(\sigma \cdot t_i, \sigma \cdot s_i)$.
- *Addition of a scalar* $[v] \oplus \sigma$*:* Given a share (t_i, s_i) of v and a value $\sigma \in \{0, 1\}$, each party P_i computes $(t_i, s_i \oplus \sigma)$.
- *Complement* $\overline{[v]}$*:* Given a share (t_i, s_i) of v, each party P_i computes $(t_i, \overline{s_i})$ (where $\overline{b}$ is b's complement)

We stress that when writing $[v_1] \oplus [v_2]$ the symbol "$\oplus$" is an operator on shares and not bitwise XOR, whereas when we write $v_1 \oplus v_2$ the symbol "$\oplus$" *is* bitwise XOR; likewise for the product and complement notation. We now prove that these local operators achieve the expected results.

Claim 2.2 *Let $[v_1], [v_2]$ be shares and let $\sigma \in \{0, 1\}$ be a scalar. Then, the following properties hold:*

1. $[v_1] \oplus [v_2] = [v_1 \oplus v_2]$
2. $\sigma \cdot [v_1] = [\sigma \cdot v_1]$
3. $[v_1] \oplus \sigma = [v_1 \oplus \sigma]$
4. $\overline{[v_1]} = [\overline{v_1}]$

Proof: Denote the shares of v_1 and v_2 by $\left\{(t_i^1, s_i^1)\right\}_{i=1}^{i=3}$ and $\left\{(t_i^2, s_i^2)\right\}_{i=1}^{i=3}$, respectively.

1. We prove that $[v_1] \oplus [v_2] = [v_1 \oplus v_2]$ by showing that $\{(t_i^1 \oplus t_i^2, s_i^1 \oplus s_i^2)\}_{i=1}^{i=3}$ is a valid sharing of $v_1 \oplus v_2$. First, observe that the t-parts are valid since $(t_1^1 \oplus t_1^2) \oplus (t_2^1 \oplus t_2^2) \oplus (t_3^1 \oplus t_3^2) = (t_1^1 \oplus t_2^1 \oplus t_3^1) \oplus (t_1^2 \oplus t_2^2 \oplus t_3^2) = 0$. Furthemore, for every i, $s_i^1 \oplus s_i^2 = (t_{i-1}^1 \oplus v_1) \oplus (t_{i-1}^2 \oplus v_2) = (t_{i-1}^1 \oplus t_{i-1}^2) \oplus (v_1 \oplus v_2)$ as required.
2. We prove that $\sigma \cdot [v_1] = [\sigma \cdot v_1]$ by showing that $\{(\sigma \cdot t_i, \sigma \cdot s_i)\}_{i=1}^{i=3}$ is a valid sharing of $\sigma \cdot v_1$. This is true since $\sigma \cdot t_1 \oplus \sigma \cdot t_2 \oplus \sigma \cdot t_3 = \sigma \cdot (t_1 \oplus t_2 \oplus t_3) = 0$ and $\sigma \cdot s_i = \sigma \cdot (t_{i-1} \oplus v_1) = \sigma \cdot t_{i-1} \oplus \sigma \cdot v_1$ as required.
3. We prove that $[v_1] \oplus \sigma = [v_1 \oplus \sigma]$ by showing that $\{(t_i, \sigma \oplus s_i)\}_{i=1}^{i=3}$ is a valid sharing of $\sigma \oplus v_1$. This is true since $t_1 \oplus t_2 \oplus t_3 = 0$ and $\sigma \oplus s_i = \sigma \oplus (t_{i-1} \oplus v_1) = t_{i-1} \oplus (v_1 \oplus \sigma)$ as required.
4. We prove that $\overline{[v_1]} = [\overline{v_1}]$ by showing that $\{(t_i, \overline{s}_i)\}_{i=1}^{i=3}$ is a valid sharing of $\overline{v}_1$. This holds since $t_1 \oplus t_2 \oplus t_3 = 0$ and $\overline{s}_i = \overline{t_{i-1} \oplus v_1} = t_{i-1} \oplus \overline{v}_1$ as required.

■

Consistency. In the setting that we consider here, one of the parties may be maliciously corrupted and thus can behave in an arbitrary manner. Thus, if parties define their shares based on values received, it may be possible that the honest parties hold values that are not a valid sharing of any value. We therefore define the notion of *consistency of shares*. We stress that this definition relates only to the shares held by the honest parties, since the corrupted party can always change its local values. As we will show after the definition, shares are consistent if they define a unique secret v.

Definition 2.3. *Let $(t_1, s_1), (t_2, s_2)$ and (t_3, s_3) be the shares held by parties P_1, P_2 and P_3 respectively, and let P_i be the corrupted party. We say that the shares are* consistent *if it holds that $s_{i+1} = s_{i+2} \oplus t_{i+2}$.*

In order to understand the definition, recall that in a valid sharing of v it holds that $s_i = v \oplus t_{i-1}$. Thus, we obtain that $s_{i+1} = v \oplus t_i = v \oplus t_{i+1} \oplus t_{i+2} = s_{i+2} \oplus t_{i+2}$ as the definition requires. The intuition behind this is that, in order to reconstruct the secret, the honest parties P_{i+1} and P_{i+2} need to learn t_i and t_{i+1} respectively. However, since t_i is held by the corrupted party, we use the fact that $t_i = t_{i+1} \oplus t_{i+2}$ to obtain that P_{i+1} can reconstruct the secret using t_{i+1} which it knows and t_{i+2} which is held by the other honest party. The definition says that computing the secret using P_{i+1}'s share and t_{i+2}; i.e., computing $s_{i+1} \oplus t_{i+1} \oplus t_{i+2}$, yields the same value as computing the secret using P_{i+2}'s share and t_{i+1}; i.e., computing $s_{i+2} \oplus t_{i+1}$. We stress that shares may be inconsistent. For example, if P_1 is the corrupted party and the shares of the honest parties P_2, P_3 are $(1, 1)$ and $(1, 1)$ respectively, then the shares are inconsistent since $s_2 = 1$ whereas $s_3 \oplus t_3 = 1 \oplus 1 = 0$. Thus, these shares cannot be the result of *any* sharing of *any* value.

2.2 Computing AND Gates – One Semi-Honest Corrupted Party

We review the protocol for securely computing AND (equivalently, multiplication) gates for semi-honest adversaries from [1] as it will be used in a subprotocol

in our protocol for malicious adversaries. This subprotocol requires each party to send a single bit only. The protocol works in two phases: in the first phase the parties compute a simple $\binom{3}{3}$ XOR-sharing of the AND of the input bits, and in the second phase they convert the $\binom{3}{3}$-sharing into the above-defined $\binom{3}{2}$-sharing.

Let $(t_1, s_1), (t_2, s_2), (t_3, s_3)$ be a secret sharing of v_1, and let $(u_1, w_1), (u_2, w_2), (u_3, w_3)$ be a secret sharing of v_2. We assume that the parties P_1, P_2, P_3 hold *correlated randomness* $\alpha_1, \alpha_2, \alpha_3$, respectively, where $\alpha_1 \oplus \alpha_2 \oplus \alpha_3 = 0$. The parties compute $\binom{3}{2}$-shares of $v_1 v_2 = v_1 \wedge v_2$ as follows :

1. **Step 1 – compute $\binom{3}{3}$-sharing:**
   (a) P_1 computes $r_1 = t_1 u_1 \oplus s_1 w_1 \oplus \alpha_1$, and sends r_1 to P_2.
   (b) P_2 computes $r_2 = t_2 u_2 \oplus s_2 w_2 \oplus \alpha_2$, and sends r_2 to P_3.
   (c) P_3 computes $r_3 = t_3 u_3 \oplus s_3 w_3 \oplus \alpha_3$, and sends r_3 to P_1.

   These messages are computed and sent in parallel.
2. **Step 2 – compute $\binom{3}{2}$-sharing:** In this step, the parties construct a $\binom{3}{2}$-sharing from their given $\binom{3}{3}$-sharing and the messages sent in the previous step. This requires local computation only.
   (a) P_1 stores (e_1, f_1) where $e_1 = r_1 \oplus r_3$ and $f_1 = r_1$.
   (b) P_2 stores (e_2, f_2) where $e_2 = r_2 \oplus r_1$ and $f_2 = r_2$.
   (c) P_3 stores (e_3, f_3) where $e_3 = r_3 \oplus r_2$ and $f_3 = r_3$.

It was shown in [1], that $e_1 \oplus e_2 \oplus e_3 = 0$, $f_1 \oplus f_2 \oplus f_3 = r_1 \oplus r_2 \oplus r_3 = v_1 v_2$ and that $f_i = e_{i-1} \oplus v_1 v_2$. Thus, the obtained sharing is a consistent sharing of $v_1 v_2$. We now show something far stronger; specifically, we show that the above multiplication protocol (for *semi-honest adversaries*) always yields consistent shares, even when run in the presence of a *malicious adversary*. Depending on the adversary, the result is either a consistent sharing of the product or its complement, but it is *always* consistent.

Lemma 2.4. *If $[v_1]$ and $[v_2]$ are consistent and $[v_3]$ was generated by executing the (semi-honest) multiplication protocol on $[v_1]$ and $[v_2]$ in the presence of one malicious party, then $[v_3]$ is a consistent sharing of either $v_1 v_2$ or $v_1 v_2 \oplus 1$.*

Proof: If the corrupted party follows the protocol specification then $[v_3]$ is a consistent sharing of $v_1 v_2$. Else, since the multiplication protocol is symmetric, assume without loss of generality that P_1 is the corrupted party. Then, the only way that P_1 can deviate from the protocol specification is by sending $r_1 \oplus 1$ to the honest P_2 instead of r_1, and in this case P_2 will define its share to be $(e_2, f_2) = (r_2 \oplus r_1 \oplus 1, r_2)$. Meanwhile, P_3 defines its share to be $(e_3, f_3) = (r_3 \oplus r_2, r_3)$, as it receives r_2 from the honest P_2. Thus, $f_3 \oplus e_3 = r_3 \oplus (r_3 \oplus r_2) = r_2 = f_2$ meaning that $[v_3]$ is consistent by Definition 2.3 . Furthermore, it is a sharing of $v_1 v_2 \oplus 1$ since $f_3 \oplus e_2 = r_3 \oplus (r_1 \oplus 1 \oplus r_2) = v_1 v_2 \oplus 1$ (utilizing the fact that $r_1 \oplus r_2 \oplus r_3 = v_1 v_2$). ■

2.3 Generating Correlated Randomness – $\mathcal{F}^1_{\text{cr}}/\mathcal{F}^2_{\text{cr}}$

Our protocol relies strongly on the use of random bits which are *correlated*. We define two types of correlated randomness:

- **Type 1:** Consider an ideal functionality $\mathcal{F}^1_{\text{cr}}$ that chooses $\alpha_1, \alpha_2, \alpha_3 \in \{0,1\}$ at random under the constraint that $\alpha_1 \oplus \alpha_2 \oplus \alpha_3 = 0$, and sends α_i to P_i for every i.
- **Type 2:** Consider an ideal functionality $\mathcal{F}^2_{\text{cr}}$ that chooses $\alpha_1, \alpha_2, \alpha_3 \in \{0,1\}$ at random, and sends (α_1, α_2) to P_1, (α_2, α_3) to P_2, and (α_3, α_2) to P_3.

Generating correlated randomness efficiently. It is possible to securely generate type-1 correlated randomness with perfect security by having each party P_j simply choose a random $\rho_j \in \{0,1\}$ and send it to P_{j+1}. Then, each P_j defines $\alpha_j = \rho_j \oplus \rho_{j-1}$ (observe that $\alpha_1 \oplus \alpha_2 \oplus \alpha_3 = 0$ since each ρ-value appears twice). In order to compute type-2 correlated randomness, each party P_j sends a random ρ_j as before, but now each P_j outputs the pair (ρ_{j-1}, ρ_j). (Formally, the ideal functionalities $\mathcal{F}^1_{\text{cr}}/\mathcal{F}^2_{\text{cr}}$ must be defined so that the corrupted party P_i has some influence, but this suffices.) Despite the elegance and simplicity of this solution, we use a different approach that does not require any communication. Although the above involves sending just a single bit, this would actually *double* the communication per AND gate which is the bottleneck of efficiency.

Protocol 2.5 describes a method for securely compute correlated randomness *computationally* without *any interaction* beyond a short initial setup. Observe that in the output of $\mathcal{F}^1_{\text{cr}}$, it holds that $\alpha_1 \oplus \alpha_2 \oplus \alpha_3 = 0$. Furthermore, for every j, P_j does not know k_{j+1} which is used to generate α_{j+1} and α_{j+2}. Thus, α_{j+1} and α_{j+2} are pseudorandom to P_j, under the constraint that $\alpha_2 \oplus \alpha_3 = \alpha_1$. This was proven formally in [1] and the same proof holds for the malicious setting.

PROTOCOL 2.5 (Computing $\mathcal{F}^1_{\text{cr}}$ / $\mathcal{F}^2_{\text{cr}}$)

- **Auxiliary input**: Each party holds a security parameter κ and a description of a pseudorandom function $F : \{0,1\}^\kappa \times \{0,1\}^\kappa \to \{0,1\}$.
- **Setup (executed once)**:
  1. Each party P_j chooses a random $k_j \in \{0,1\}^\kappa$.
  2. Each party P_j sends k_j to party P_{j+1}.
- **Generating randomness**: Upon input id,
  - *Computing* $\mathcal{F}^1_{\text{cr}}$*:* each party P_j computes $\alpha_j = F_{k_j}(id) \oplus F_{k_{j-1}}(id)$ and outputs it.
  - *Computing* $\mathcal{F}^2_{\text{cr}}$*:* each party P_j outputs $\big(F_{k_j}(id), F_{k_{j-1}}(id)\big)$.

Formally defining the $\mathcal{F}^1_{\text{cr}}$ / $\mathcal{F}^2_{\text{cr}}$ ideal functionalities. A naive definition would be to have the ideal functionality choose $\alpha_1, \alpha_2, \alpha_3$ and send α_j to P_j for $= j \in \{1,2,3\}$ (or send α_j, α_{j-1} to P_i in the $\mathcal{F}^2_{\text{cr}}$ functionality). However, securely realizing such a functionality would require a full-blown coin tossing protocol. In order to model our non-interactive method, which suffices for our protocol, we need to take into account that the corrupted party P_i can choose

its k_i and this influences the output, as P_i's value is generated in a very specific way using a pseudorandom function. In order for the view of the corrupted party to be like in the real protocol, we define the functionality $\mathcal{F}^1_{\text{cr}}$ / $\mathcal{F}^2_{\text{cr}}$ so that they generate the corrupted party's value in this exact same way.

The functionalities are described formally in Functionality 2.6. The corrupted party chooses two keys k, k' for the pseudorandom function F and sends them to the functionality. These keys are used to generate the values that are influenced by the corrupted party, whereas the other values are chosen uniformly. We denote by κ the computational security parameter, and thus the length of the keys k, k'.

FUNCTIONALITY 2.6 ($\mathcal{F}^1_{\text{cr}}$ / $\mathcal{F}^2_{\text{cr}}$ – correlated randomness)

Let $F : \{0,1\}^\kappa \times \{0,1\}^\kappa \to \{0,1\}$ be a keyed function. Upon invocation, the adversary controlling party P_i chooses a pair of keys $k, k' \in \{0,1\}^\kappa$ and sends them to $\mathcal{F}^1_{\text{cr}}$ / $\mathcal{F}^2_{\text{cr}}$. Then:

- $\mathcal{F}^1_{\text{cr}}$: Upon receiving input id from all parties, functionality $\mathcal{F}^1_{\text{cr}}$ computes $\alpha_i = F_k(id) \oplus F_{k'}(id)$ and chooses random values $\alpha_{i-1}, \alpha_{i+1} \in \{0,1\}$ under the constraint that $\alpha_1 \oplus \alpha_2 \oplus \alpha_3 = 0$. $\mathcal{F}^1_{\text{cr}}$ sends α_j to P_j for every j.
- $\mathcal{F}^2_{\text{cr}}$: Upon receiving input id from all parties, functionality $\mathcal{F}^2_{\text{cr}}$ computes $\alpha_i = F_k(id)$ and $\alpha_{i-1} = F_{k'}(id)$ and chooses a random value $\alpha_{i+1} \in \{0,1\}$. $\mathcal{F}^2_{\text{cr}}$ sends (α_{j-1}, α_j) to P_j for every j.

The following is proved in Appendix B.

Proposition 2.7. *If F is a pseudorandom function, then Protocol 2.5 securely computes functionalities $\mathcal{F}^1_{\text{cr}}$ and $\mathcal{F}^2_{\text{cr}}$, respectively, with abort in the presence of one malicious party.*

2.4 Generating Shares of a Random Value – $\mathcal{F}_{\text{rand}}$

In this section, we show how the parties can generate a sharing of a random secret value v known to none of them. Formally, we define the functionality $\mathcal{F}_{\text{rand}}$ that chooses a random $v \in \{0,1\}$, computes a sharing $[v]$, and sends each party its share of $[v]$. However, $\mathcal{F}_{\text{rand}}$ allows the corrupted party to determine its own share, and thus computes the honest parties' shares from the corrupted party's share and the randomly chosen v. $\mathcal{F}_{\text{rand}}$ is formally specified in Functionality 2.8.

FUNCTIONALITY 2.8 ($\mathcal{F}_{\text{rand}}$ – generating shares of a random value)

- $\mathcal{F}_{\text{rand}}$ receives (t_i, s_i) from the corrupted party P_i.
- $\mathcal{F}_{\text{rand}}$ chooses a random $v \in \{0,1\}$ and defines the respective shares $(t_{i-1}, s_{i-1}), (t_{i+1}, s_{i+1})$ of P_{i-1}, P_{i+1} based on (t_i, s_i) and v (as described in Claim 2.1).
- $\mathcal{F}_{\text{rand}}$ sends (t_{i-1}, s_{i-1}) to P_{i-1}, and sends (t_{i+1}, s_{i+1}) to P_{i+1}.

Protocol 2.9 describes how to securely compute $\mathcal{F}_{\text{rand}}$ in the $\mathcal{F}^2_{\text{cr}}$-hybrid model, without any interaction.

PROTOCOL 2.9 (Securely computing $\mathcal{F}_{\text{rand}}$)

1. The parties call $\mathcal{F}^2_{\text{cr}}$ and receive (r_3, r_1), (r_1, r_2), and (r_2, r_3), respectively.
2. P_1 defines $t_1 = r_3 \oplus r_1$ and $s_1 = r_1$.
3. P_2 defines $t_2 = r_1 \oplus r_2$ and $s_2 = r_2$.
4. P_3 defines $t_3 = r_2 \oplus r_3$ and $s_3 = r_3$.

Observe that $t_1 \oplus t_2 \oplus t_3 = 0$. Furthermore, define $v = s_1 \oplus t_3 = r_1 \oplus r_2 \oplus r_3$. Observe that $s_2 \oplus t_1$ and $s_3 \oplus t_2$ also both equal the same v. Thus, this non-interactive protocol defines a valid sharing $[v]$ for a random $v \in \{0, 1\}$. The fact that v is random follows from the fact that it equals $r_1 \oplus r_2 \oplus r_3$. Now, by the definition of $\mathcal{F}^2_{\text{cr}}$, a corrupted P_i knows nothing of $r_{i+1} = \alpha_{i+1}$ which is chosen uniformly at random, and thus the defined sharing is of a random value.

Proposition 2.10. *Protocol 2.9 securely computes functionality $\mathcal{F}_{\text{rand}}$ with abort in the $\mathcal{F}^2_{\text{cr}}$-hybrid model, in the presence of one malicious party.*

Proof: Let $\mathcal{A}$ be a real adversary; we define $\mathcal{S}$ as follows:

- $\mathcal{S}$ receives $\mathcal{A}$'s input k, k' to $\mathcal{F}^2_{\text{cr}}$.
- Upon receiving id from $\mathcal{A}$ as intended for $\mathcal{F}^2_{\text{cr}}$, simulator $\mathcal{S}$ simulates $\mathcal{A}$ receiving back $(r_{i-1}, r_i) = (F_{k'}(id), F_k(id))$ from $\mathcal{F}^2_{\text{cr}}$.
- $\mathcal{S}$ defines $t_i = r_{i-1} \oplus r_i$ and $s_i = r_i$, and externally sends (t_i, s_i) to $\mathcal{F}_{\text{rand}}$.
- $\mathcal{S}$ outputs whatever $\mathcal{A}$ outputs.

We show that the joint distribution of the outputs of $\mathcal{S}$ and the honest parties in an ideal execution is identical to the outputs of $\mathcal{A}$ and the honest parties in a real execution. In order to see this, observe that in a real execution, given a fixed r_{i-1}, r_i (as viewed by the adversary), the value v is fully determined by r_{i+1}. In particular, by the definition of the secret-sharing scheme, $v = s_i \oplus t_{i-1} = r_i \oplus r_{i-2} \oplus r_{i-1} = r_1 \oplus r_2 \oplus r_3$. Since r_{i+1} is randomly generated by $\mathcal{F}^2_{\text{cr}}$, this has the same distribution as $\mathcal{F}_{\text{rand}}$ choosing v randomly (because choosing v randomly, or choosing some r randomly and setting $v = t_i \oplus r$ is identical). Thus, the joint distributions are identical. ■

2.5 Coin Tossing – $\mathcal{F}_{\text{coin}}$

We now present a highly-efficient three-party coin tossing protocol that is secure in the presence of one malicious adversary. We define the functionality $\mathcal{F}_{\text{coin}}$ that chooses s random bits $v_1, \ldots, v_s \in \{0, 1\}$ and sends them to each of the parties. The idea behind our protocol is simply for the parties to invoke s calls to $\mathcal{F}_{\text{rand}}$ and to then open the result (by each P_i sending t_i to P_{i+1}; see Section 2.1). Observe that this in itself is not sufficient since a malicious party may send an *incorrect opening*, resulting in the honest parties receiving different output. This can be solved by using a subprocedure called compareview() in which each party P_j sends its output to party P_{j+1}. If any party receives a different output, then it aborts. The reason why this is secure is that the protocol guarantees that if P_i is corrupted then P_{i+2} receives the correct outputs $v_1, \ldots, v_s$; this holds

because when opening the shares, the only values received by P_{i+2} are sent by the honest P_{i+1} and are not influenced by P_i. Thus, P_{i+2}'s output is guaranteed to be correct, and if P_{i+1} and P_{i+2} have the same output then P_{i+1}'s output is also correct. This is formally described in Protocol 2.11.

PROTOCOL 2.11 (Securely computing $\mathcal{F}_{\text{coin}}$)

1. The parties invoke s calls to $\mathcal{F}_{\text{rand}}$; denote their outputs by $[v_1], ..., [v_s]$.
2. For every $j \in \{1, .., s\}$, the parties run the $\mathsf{open}([v_j])$ procedure defined in Section 2.1 to obtain v_j.
3. The parties run $\mathsf{compareview}(v_1, \ldots, v_s)$ by each P_j sending the outputs $v_1, \ldots, v_s$ to P_{j+1}. If a party receives different output, then it outputs $\perp$. Otherwise, it outputs $v_1, \ldots, v_s$.

Proposition 2.12. *Protocol 2.11 securely computes functionality $\mathcal{F}_{\text{coin}}$ with abort in the $\mathcal{F}_{\text{rand}}$-hybrid model, in the presence of one malicious party.*

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

1. $\mathcal{S}$ receives $v_1, \ldots, v_s$ from the trusted party computing $\mathcal{F}_{\text{coin}}$.
2. $\mathcal{S}$ invokes $\mathcal{A}$ and simulates s calls to $\mathcal{F}_{\text{rand}}$, as follows:
   (a) $\mathcal{S}$ receives P_i's share in every call to $\mathcal{F}_{\text{rand}}$.
   (b) Given $v_1, \ldots, v_s$ and P_i's shares, $\mathcal{S}$ computes the value t_{i-1} that $\mathcal{A}$ should receive from P_{i-1}. (Specifically, for the ℓth value, let (t_i^ℓ, s_i^ℓ) be P_i's share and let v_ℓ be the bit received from $\mathcal{F}_{\text{coin}}$. Then, $\mathcal{S}$ sets $t_{i-1} = v_\ell \oplus s_i^\ell$. This implies that the "opening" is to v_ℓ.)
   (c) If $\mathcal{A}$ sends an incorrect t_i value in any open procedure, then $\mathcal{S}$ sends abort_{i+1} to $\mathcal{F}_{\text{coin}}$ causing P_{i+1} to abort in the ideal model (see Appendix A for the definition of the ideal model). Otherwise, it sends $\mathsf{continue}_{i+1}$ to $\mathcal{F}_{\text{coin}}$. (In all cases it sends $\mathsf{continue}_{i-1}$ to $\mathcal{F}_{\text{coin}}$ since P_{i-1} never aborts.)

By the way $\mathcal{F}_{\text{rand}}$ is defined, the output distribution of $\mathcal{A}$ and the honest parties in a real execution is identical to the output distribution of $\mathcal{S}$ and the honest parties in an ideal execution. This is because each v_j is uniformly distributed, and $\mathcal{S}$ can fully determine the messages that $\mathcal{A}$ receives for any fixed $v_1, \ldots, v_s$. ■

Deferring compareview. Our protocol has the property that nothing is revealed to any party until the end, even if a party behaves maliciously. As such, it will suffice for us to run compareview only once at the very end of the protocol before outputs are revealed. This enables us to have the parties compare their views by only sending a *collision-resistant hash* of their outputs, thereby reducing communication. As we will see below, this method will be used in a number of places, and all compareview operations will be done together.

2.6 Random Shuffle – $\mathcal{F}_{\mathbf{perm}}$

In our protocol, we will need to compute a random permutation of an an array of elements (where each element is a "multiplication triple"). Let $\mathcal{F}_{\text{perm}}$ be an ideal functionality that receives a vector $\boldsymbol{d}$ of length M from all parties, chooses a random permutation π over $\{1, ..., M\}$ and returns the vector $\boldsymbol{d}'$ defined by $\boldsymbol{d}'[i] = \boldsymbol{d}[\pi[i]]$ for every $i \in \{1, ..., M\}$. Functionality $\mathcal{F}_{\text{perm}}$ can be securely computed by the parties running the Fisher-Yates shuffle algorithm [10], and obtaining randomness via $\mathcal{F}_{\text{coin}}$. This is formally described in Protocol 2.13.

PROTOCOL 2.13 (Securely computing $\mathcal{F}_{\text{perm}}$)

All parties hold the same input $\boldsymbol{d}$, and work as follows:

1. For $j = 1$ to M:
   (a) The parties call $\mathcal{F}_{\text{coin}}$ enough times to generate a random index $i \in \{j, \ldots, M\}$.
   (b) Each party swaps $\boldsymbol{d}[j]$ and $\boldsymbol{d}[i]$.
2. Each party output the resulting vector $\boldsymbol{d}$.

The following proposition follows trivially from the security of $\mathcal{F}_{\text{coin}}$:

Proposition 2.14. *Protocol 2.13 securely computes functionality $\mathcal{F}_{\text{perm}}$ with abort in the $\mathcal{F}_{\text{coin}}$-hybrid model, in the presence of one malicious party.*

2.7 Reconstruct a Secret to One of the Parties – $\mathcal{F}_{\mathbf{reconst}}$

In this section we show how the parties can open a consistent sharing $[v]$ of a secret v to *one* of the parties in a secure way. We will use this subprotocol for reconstructing the outputs in our protocol. We remark that we consider security with abort only, and thus the party who should receive the output may abort. We stress that this procedure is fundamentally different to the open procedure of Section 2.1 in two ways: first, only one party receives output; second, the open procedure does not guarantee correctness. In contrast, here we ensure that the party either receives the correct value or aborts. We stress, however, that the protocol is only secure if the sharing $[v]$ is consistent; otherwise, nothing is guaranteed. We formally define $\mathcal{F}_{\text{reconst}}$ in Functionality 2.15.

FUNCTIONALITY 2.15 ($\mathcal{F}_{\text{reconst}}$ – secure reconstruction)

Let $\mathcal{S}$ be the adversary and P_i the corrupted party. $\mathcal{F}_{\text{reconst}}$ receives (t_{i+1}, s_{i+1}, j) from P_{i+1} and (t_{i+2}, s_{i+2}, j) from P_{i+2}, and works as follows:

- $\mathcal{F}_{\text{reconst}}$ computes $v = s_{i+2} \oplus t_{i+1}$ and sends v to P_j. In addition, $\mathcal{F}_{\text{reconst}}$ sends (t_i, s_i) to $\mathcal{S}$ (where (t_i, s_i) is P_i's share as defined by the shares received from the honest parties).

Note that $\mathcal{F}_{\text{reconst}}$ also sends P_i's share to $\mathcal{S}$. This is needed technically in the proof to enable simulation; it reveals nothing since this is the corrupted party's

share anyway. Also, observe that the output is determined solely by the honest parties' shares; this guarantees that the corrupted party cannot influence the output beyond causing an abort.

We show how to securely compute $\mathcal{F}_{\text{reconst}}$ in Protocol 2.16. Intuitively, the protocol works by the parties sending their shares to P_j who checks that they are consistent, and reconstructs if yes. In order to reduce communication, we actually show that it suffices for the parties to send only the "t" parts of their shares.

PROTOCOL 2.16 (Reconstruct a Secret to One Party)

- **Inputs:** The parties hold a sharing $[v]$ and an index $j \in \{1, 2, 3\}$.
- **The protocol:**
  1. Parties P_{j+1} and P_{j-1} send t_{j+1} and t_{j-1}, respectively, to P_j.
  2. Party P_j checks that $t_j = t_{j+1} \oplus t_{j+2}$. If yes, it outputs $v = s_j \oplus t_{j-1}$; otherwise, it outputs $\perp$.

Proposition 2.17. *If the honest parties' inputs shares are consistent as in Definition 2.3, then Protocol 2.16 securely computes $\mathcal{F}_{\text{reconst}}$ with abort, in the presence of one malicious party.*

Proof: Let $\mathcal{A}$ be the real adversary controlling P_i, and assume that the honest parties' shares are consistent. We first consider the case that P_j is corrupt (i.e., $i = j$). In this case, the simulator $\mathcal{S}$ receives v and $(t_i, s_i) = (t_j, s_j)$ from $\mathcal{F}_{\text{reconst}}$. These values fully define all other shares, and in particular the values t_{j+1} and t_{j-1}. Thus, $\mathcal{S}$ can simulates P_{j+1} and P_{j-1} sending t_{j+1} and t_{j-1} to P_j.

We next consider the case that P_j is honest (i.e., $j \neq i$). In this case, $\mathcal{S}$ receives P_i's share (t_i, s_i) from $\mathcal{F}_{\text{reconst}}$. Then, $\mathcal{S}$ invokes $\mathcal{A}$ and receives the bit t'_i that P_i would send to P_j. If $t'_i = t_i$ (where t_i is the correct share value as received from $\mathcal{F}_{\text{reconst}}$), then $\mathcal{S}$ sends $\mathsf{continue_j}$ to $\mathcal{F}_{\text{reconst}}$ so that the honest P_j receives v. Otherwise, $\mathcal{S}$ sends $\mathsf{abort_j}$ to $\mathcal{F}_{\text{reconst}}$ so that the honest P_j outputs $\perp$. Observe that in a real protocol P_j aborts unless $t_1 \oplus t_2 \oplus t_3 = 0$ (which is equivalent to $t_j = t_{j+1} \oplus t_{j+2}$). Thus, if the corrupted party sends an incorrect t_i value, then P_j will certainly abort. In contrast, if the adversary controlling P_i sends the correct t_i, then the output will clearly be the correct v, again as in the ideal execution with $\mathcal{S}$. ■

2.8 Robust Sharing of a Secret – $\mathcal{F}_{\text{share}}$

In this section, we show how to share a secret that is held by one of the parties who may be corrupt. This sub-protocol will be used to share the parties' inputs in the protocol. We define $\mathcal{F}_{\text{share}}$ in Functionality 2.18. We note that the corrupted party always provides its share as input, as in $\mathcal{F}_{\text{rand}}$. In addition, the dealer provides v and the (honest) parties receive their correct shares as defined by these values.

FUNCTIONALITY 2.18 (Functionality $\mathcal{F}_{\text{share}}$ – sharing a secret)

Let P_j be the party playing the dealer, and let P_i be the corrupted party:

- The corrupted P_i sends (t_i, s_i) to $\mathcal{F}_{\text{share}}$.
- The dealer P_j sends v to $\mathcal{F}_{\text{share}}$.
- $\mathcal{F}_{\text{share}}$ computes (t_{j+1}, s_{j+1}) and (t_{j+2}, s_{j+2}) from (t_i, s_i) and v (as described in Claim 2.1) and sends the honest P_{i-1} and P_{i+1} their respective shares.

We show how to securely compute $\mathcal{F}_{\text{share}}$ in Protocol 2.19. The idea behind the protocol is to first generate a random sharing via $\mathcal{F}_{\text{rand}}$ which guarantees a *consistent* sharing of a random value (recall that this requires no communication). Next, the parties reconstruct the shared secret to the dealer, who can then send a single bit to "correct" the random share to its actual input. This ensures that the honest parties hold consistent shares, as long as a corrupt dealer sent the same bit to both; this is enforced by the parties comparing to ensure that they received the same bit from the dealer.

PROTOCOL 2.19 (Robust Sharing of a Secret)

- **Inputs:** Party P_j holds a bit $v \in \{0, 1\}$.
- **The protocol:**
  1. The parties call $\mathcal{F}_{\text{rand}}$ to obtain $[a]$ for a random $a \in \{0, 1\}$.
  2. The parties call $\mathcal{F}_{\text{reconst}}$ with $[a]$ and j as its inputs, and so P_j receives a. If P_j receives $\perp$, it sends $\perp$ to all other parties and halts. Else, it proceeds to the next step.
  3. Party P_j sends $b = a \oplus v$ to the other parties.
  4. The parties run $\mathsf{compareview}(b)$ by each P_j sending the bit b to P_{j+1}. If any party sees different b values, then it sends $\perp$ to all other parties and halts.
  5. The parties each set their share $[v] = [a] \oplus b$ (using the operator defined in Section 2.1).
- **Output**: Each party outputs its share in $[v]$.

The following is proven in Appendix C.

Proposition 2.20. *Protocol 2.19 securely computes $\mathcal{F}_{\text{share}}$ with abort in the $(\mathcal{F}_{\text{rand}}, \mathcal{F}_{\text{reconst}})$-hybrid mode, in the presence of one malicious party.*

Deferring $\mathsf{compareview}$**.** As in Section 2.5, the $\mathsf{compareview}$ step can be deferred to the end of the execution (before any output is revealed). When using this mechanism, the bits to be compared are simply added to the parties local view and stored, and they are compared at the end.

2.9 Triple Verification With Opening

A multiplication triple is a triple of shares $([a], [b], [c])$ with the property that $c = a \cdot b$. Our protocol works by constructing triples and verifying that indeed $c = a \cdot b$. We begin by defining what it means for such a triple to be correct.

Definition 2.21. $([a],[b],[c])$ *is a* correct *multiplication triple if* $[a],[b]$ *and* $[c]$ *are consistent sharings, and* $c = a \cdot b$.

In our main protocol for secure computation, the parties will generate multiplication triples in two steps:

1. The parties generate random sharings $[a]$ and $[b]$ by calling $\mathcal{F}_{\text{rand}}$ twice.
2. The parties run the semi-honest multiplication protocol described in Section 2.2 to obtain $[c]$.

Recall that by Lemma 2.4, the sharing $[c]$ is *always* consistent. However, if one of the parties is malicious, then it may be that $c = ab \oplus 1$. Protocol 2.22 describes a method of verifying that a triple is correct. The protocol is very simple and is based on the fact that if the shares are consistent and $c \neq ab$, then one of the honest parties will detect this in the standard open procedure defined in Section 2.1. This protocol is called verification "with opening" since the values a, b, c are revealed.

PROTOCOL 2.22 (Triple Verification With Opening)

- **Inputs:** The parties holds the triple $([a],[b],[c])$.
- **The protocol:**
  1. The parties run the procedures $\mathsf{open}([a]), \mathsf{open}([b])$ and $\mathsf{open}([c])$. Denote the output of party P_j from the three procedures by a_j, b_j and c_j respectively.
  2. Each party P_j checks that $c_j = a_j \cdot b_j$. If no, it sends $\perp$ to both parties and aborts.
  3. If no $\perp$ message is received, each party outputs accept.

Lemma 2.23. *If* $[a],[b],[b]$ *are consistent shares, but* $([a],[b],[c])$ *is not a correct multiplication triple, then both honest parties output* $\perp$ *in Protocol 2.22.*

Proof: Let P_i be the corrupted party, and assume that $[a],[b],[c]$ are consistent shares, but $([a],[b],[c])$ is not a correct multiplication triple. This implies that $c = a \cdot b \oplus 1$. Therefore, in the open procedures, party P_{i+2} will receive values $a_{i+2}, b_{i+2}, c_{i+2}$ such that $c_{i+2} \neq a_{i+2} \cdot b_{i+2}$, and will send $\perp$ to both parties. (This holds since P_{i+2} receives messages only from P_{i+1} that are independent of what P_i sends.) Thus, both honest parties will output $\perp$. ■

2.10 Triple Verification Using Another (Without Opening)

We have seen how to check a triple by opening it and revealing its values a, b, c. Such a triple can no longer be used in the protocol. In this section, we show how to verify that a multiplication triple is consistent *without opening it*, by using (and wasting) an additional random multiplication triple that is assumed to be consistent. The method is described in Protocol 2.24. The idea behind the protocol is as follows. Given shares of x, y, z and of a, b, c, the parties compute and open shares of $\rho = x \oplus a$ and $\sigma = y \oplus b$; these values reveal nothing about

x and y since a, b are both random. As we will show in the proof below, if one of (x, y, z) or (a, b, c) is correct and the other is incorrect (e.g., $x \neq y \cdot z$ but $c = a \cdot b$) then $z + c + \sigma \cdot a + \rho \cdot b + \rho \cdot \sigma = 1$. Thus, this value can be computed and opened by the parties. If x, y, z is incorrect and a, b, c is correct, then the honest parties will detect this and abort. In order to save on communication, we observe that if the value to be opened must equal 0 then it must hold that $s_j = t_{j-1}$. Thus, it suffices for the parties to compare a single bit.

PROTOCOL 2.24 (Triple Verification Using Another Without Opening)

- **Inputs:** The parties hold a triple $([x], [y], [z])$ to verify and an additional triple $([a], [b], [c])$.
- **The protocol:**
  1. Each party locally computes $[\rho] = [x] \oplus [a]$ and $[\sigma] = [y] \oplus [b]$.
  2. The parties run $\mathsf{open}([\rho])$ and $\mathsf{open}([\sigma])$, as defined in Section 2.1. Denote by ρ_j and σ_j the respective output received by P_j in the openings.
  3. The parties run $\mathsf{compareview}(\rho_j, \sigma_j)$ by each P_j sending (ρ_j, σ_j) to P_{j+1}. If a party sees different values, then it sends $\perp$ to all parties and outputs $\perp$.
  4. Each party P_j computes $[z] \oplus [c] \oplus \sigma_j \cdot [a] \oplus \rho_j \cdot [b] \oplus \rho_j \cdot \sigma_j$. Denote by (t_j, s_j) the result of the computation held by party P_j.
  5. The parties run $\mathsf{compareview}(t_j)$ by each P_j sending t_j to P_{j+1}. Upon receiving t_{j-1} from P_{j-1}, party P_j checks that $s_j = t_{j-1}$. If yes, it outputs accept; else, it sends $\perp$ to all other parties and outputs $\perp$.
  6. If no abort messages are received, then output accept.

Lemma 2.25. *If $([a], [b], [c])$ is a correct multiplication triple and $[x], [y], [z]$ are consistent shares, but $([x], [y], [z])$ is not a correct multiplication triple, then all honest parties output $\perp$ in Protocol 2.24.*

Proof: Let P_i be the corrupted party. Assume that $([a], [b], [c])$ is a correct multiplication triple, that $[x], [y], [z]$ are consistent sharings, but $([x], [y], [z])$ is not a correct multiplication triple. This implies that all values a, b, c, x, y, z are well defined (from the honest parties' shares) and that $c = ab$ and $z \neq xy$.

Let $\rho = x \oplus a$ and $\sigma = y \oplus b$. If P_{j+1} receives an incorrect bit from P_j in the openings of ρ and σ (i.e., if $\rho_j \neq \rho$ or $\sigma_j \neq \sigma$) then it detects this in $\mathsf{compareview}$ of Step 3 with P_{j+2} and thus both honest parties output $\perp$. (Observe that P_{j+2} receives the openings from P_{j+1} who is honest and thus it is guaranteed that $\rho_{j+1} = \rho$ and $\sigma_{j+1} = \sigma$.)

We now show that if P_{i+1} and P_{i+2} did not output $\perp$ in Step 3 (and thus $\sigma_{i+1} = \sigma_{i+2} = \sigma$ and $\rho_{i+1} = \rho_{i+2} = \rho$), then P_{i+1} and P_{i+2} output $\perp$ with probability 1 in Step 5. In order to show this, we first show that in this case, $[z] \oplus [c] \oplus \sigma_j \cdot [a] \oplus \rho_j \cdot [b] \oplus \rho_j \cdot \sigma_j = [1]$. Observe that $z \neq xy$ and thus $z = xy \oplus 1$, and that $\sigma = y \oplus b$ and $\rho = x \oplus a$. Thus, we have:

$$\begin{aligned}
&[z] \oplus [c] \oplus \sigma[a] \oplus \rho[b] \oplus \rho\sigma \\
&\qquad = [xy \oplus 1] \oplus [c] \oplus (y \oplus b)[a] \oplus (x \oplus a)[b] \oplus (x \oplus a)(y \oplus b)
\end{aligned}$$

$$
\begin{aligned}
&= [xy \oplus 1] \oplus [c] \oplus [(y \oplus b)a] \oplus [(x \oplus a)b] \oplus (xy \oplus ay \oplus xb \oplus ab) \\
&= [xy \oplus 1 \oplus c \oplus (y \oplus b)a \oplus (x \oplus a)b \oplus xy \oplus ay \oplus xb \oplus ab] \\
&= [xy \oplus 1 \oplus c \oplus ya \oplus ba \oplus xb \oplus ab \oplus xy \oplus ay \oplus xb \oplus ab] \\
&= [1]
\end{aligned}
$$

where the last equality follows from simple cancellations and the fact that by the assumption $c = ab$. We therefore have that the honest parties hold a consistent sharing of 1. Denoting the respective shares of P_{i+1} and P_{i+2} by (t_{i+1}, s_{i+1}) and (t_{i+2}, s_{i+2}), by the definition of the secret-sharing scheme we have that $s_{i+2} = t_{i+1} \oplus 1$ and so $s_{i+2} \neq t_{i+1}$. This implies that P_{i+2} sends $\perp$ to all other parties in Step 5 of the protocol, and all output $\perp$. ■

Deferring compareview**:** In the first compareview, all parties include ρ_j, σ_j in their view. In the second compareview, P_j includes t_j in its joint view with P_{j+1} and includes s_j in its joint view with P_{j-1}. Observe that the requirement that $s_j = t_{j-1}$ is automatically fulfilled by requiring that the pairwise views be the same. This holds since P_j include s_j in its view with P_{j-1}, whereas P_{j-1} includes t_{j-1} in its view with P_j. As a consequence, in the protocol when compareview is deferred, each party stores two strings – one for its joint view with each of the other parties – and hashes these two strings separately at the end of the protocol. We remark that it is possible to use only a universal hash function by choosing the function after the views have been fixed, if this is desired. Recall that in compareview, it suffices for each party to send its view to one other party. Thus, all communication in our protocol follows the pattern that P_i sends messages to P_{i+1} only, for every $i \in \{1, 2, 3\}$.

3 Secure Generation of Multiplication Triples – $\mathcal{F}_{\text{triples}}$

In this section, we present a three-party protocol for generating an array of correct multiplication triples, as defined in Section 2.21. Formally, we securely compute the functionality $\mathcal{F}_{\text{triples}}$ defined in Functionality 3.1.

FUNCTIONALITY 3.1 ($\mathcal{F}_{\text{triples}}$ – generating multiplication triples)

Let P_i be the corrupted party. Upon receiving N from P_1, P_2, P_3, and receiving N triples of pairs $\left\{(t^j_{a_i}, s^j_{a_i}), (t^j_{b_i}, s^j_{b_i}), (t^j_{c_i}, s^j_{c_i})\right\}_{j=1}^{N}$ from P_i, functionality $\mathcal{F}_{\text{triples}}$ works as follows:

- For $j = 1, \ldots, N$, $\mathcal{F}_{\text{triples}}$ chooses random $a_j, b_j \in \{0, 1\}$ and computes $c_j = a_j b_j$.
- For $j = 1, \ldots, N$, $\mathcal{F}_{\text{triples}}$ defines a vector of sharings $\boldsymbol{d} = ([a_j], [b_j], [c_j])$. The sharings are computed from $\left[(t^j_{a_i}, s^j_{a_i}), (t^j_{b_i}, s^j_{b_i}), (t^j_{c_i}, s^j_{c_i})\right]$ provided by P_i and the chosen a_j, b_j, c_j (as in Claim 2.1).
- $\mathcal{F}_{\text{triples}}$ sends each party its shares in all of the generated shares.

We show how to securely compute $\mathcal{F}_{\text{triples}}$ in Protocol 3.2.

PROTOCOL 3.2 (Generating Multiplication triples)

- **Input:** The number N of triples to be generated.
- **Auxiliary input:** Parameters B and C.
- **The Protocol:**
  1. *Generate random sharings:* The parties invoke $2(NB+C)$ calls to $\mathcal{F}_{\text{rand}}$; denote the shares that they receive by $[([a_i],[b_i])]_{i=1}^{NB+C}$.
  2. *Generate multiplication triples:* For $i = 1,\ldots,NB+C$, the parties run the semi-honest multiplication protocol of Section 2.2 to compute $[c_i] = [a_i]\cdot[b_i]$. Denote $\boldsymbol{D} = [([a_i],[b_i],[c_i])]_{i=1}^{NB+C}$; observe that $[c_i]$ is the result of the protocol and is not necessarily "correct".
  3. *Cut and bucket:* Let $M = NB + C$. In this stage, the parties perform a first verification that the triples were generated correctly by opening C triples, and then randomly divide the remainder into buckets.
     (a) The parties call $\mathcal{F}_{\text{perm}}$ with vector $\boldsymbol{D}$.
     (b) The parties run Protocol 2.22 (triple verification with opening) for each of the first C triples in $\boldsymbol{D}$, and remove them from $\boldsymbol{D}$. If a party did not output accept in every execution, it sends $\perp$ to the other parties and outputs $\perp$.
     (c) The remaining NB triples in $\boldsymbol{D}$ are divided into N sets of triples $\boldsymbol{D}_1,\ldots,\boldsymbol{D}_N$, each of size B. For $i = 1,\ldots,N$, the bucket $\boldsymbol{D}_i$ contains the triples $([a_{(i-1)\cdot B+1}],[b_{(i-1)\cdot B+1}],[c_{(i-1)\cdot B+1}]),\ldots,([a_{i\cdot B}],[b_{i\cdot B}],[c_{i\cdot B}])$.
  4. *Check buckets:* The parties initialize a vector $\boldsymbol{d}$ of length N. Then, for $i = 1,\ldots,N$:
     (a) Denote the triples in $\boldsymbol{D}_k$ by $([a_1],[b_1],[c_1]),\ldots,([a_B],[b_B],[c_B])$.
     (b) For $j = 2,\ldots,B$, the parties run Protocol 2.24 (triple verification using another without opening) on input $([a_1],[b_1],[c_1])$ and $([a_j],[b_j],[c_j])$, to verify $([a_1],[b_1],[c_1])$.
     (c) If a party did not output accept in every execution, it sends $\perp$ to the other parties and outputs $\perp$.
     (d) The parties set $\boldsymbol{d}_i = ([a_1],[b_1],[c_1])$; i.e., they store these shares in the ith entry of $\boldsymbol{d}$.
- **Output:** The parties output $\boldsymbol{d}$.

The idea behind the protocol is as follows. The parties first use $\mathcal{F}_{\text{rand}}$ to generate many shares of *pairs* of random values $[a_i],[b_i]$. Next, they run the semi-honest multiplication protocol of Section 2.2 to compute shares of $c_i = a_i \cdot b_i$. However, since the multiplication protocol is only secure for semi-honest parties, a malicious adversary can cheat in this protocol. We therefore utilize the fact that even if the malicious adversary cheats, the resulting shares $[c_i]$ is *consistent*, but it may be the case that $c_i = a_i b_i \oplus 1$ instead of $c_i = a_i b_i$ (see Claim 2.4). We therefore use *cut-and-choose* to check that the triples are indeed correct. We do this by opening C triples using Protocol 2.22; this protocol provides a full guarantee that the parties detect any incorrect triple that is opened. Next, the parties randomly divide the remaining triples into "buckets" of size B and use Protocol 2.24 to verify that the first triple in the bucket is correct. Recall

that in Protocol 2.24, one triple is used to check another without revealing its values. Furthermore, by Lemma 2.25, if the first triple is not correct and the second is, then this is detected by the honest parties. Thus, the only way an adversary can successfully cheat is if **(a)** no incorrect triples are opened, and **(b)** there is no bucket with both correct and incorrect triples. Stated differently, the adversary can only successfully cheat if there exists a bucket where all triples are incorrect. By appropriately choosing the bucket-size B and number of triples C to be opened, the cheating probability can be made negligibly small.

Proposition 3.3. *Let N, B, C be such that $N = CB^2$ and $(B-1)\log_2 C \geq \sigma$. Then, Protocol 3.2 securely computes $\mathcal{F}_{\text{triples}}$ with abort in the $(\mathcal{F}_{\text{rand}}, \mathcal{F}_{\text{perm}})$-hybrid model, with statistical error $2^{-\sigma}$ and in the presence of one malicious party.*

Proof Sketch: Intuitively, the triples generated are to random values since $\mathcal{F}_{\text{rand}}$ is used to generate $[a_i], [b_i]$ for all i (note that in $\mathcal{F}_{\text{triples}}$ the adversary chooses its shares in $[a_i], [b_i]$; this is inherited from its capability in $\mathcal{F}_{\text{rand}}$). Then, Protocol 2.22 is used to ensure that the first C triples are all correct (recall that by Lemma 2.4, $[a_i], [b_i], [c_i]$ are all consistent sharings, and thus by Lemma 2.23 the honest parties output $\perp$ if $c_i \neq a_i b_i$). Finally, Protocol 2.24 is used to verify that all of the triples in $\boldsymbol{d}$ are correct multiplication triples. By Lemma 2.25, if $([a_1], [b_1], [c_1])$ in any of the buckets is not a correct multiplication triple, and there exists a $j \in \{2, \ldots, B\}$ for which $([a_j], [b_j], [c_j])$ *is* a correct multiplication triple, then the honest parties output $\perp$ (note that once again by Lemma 2.4, all of the shares are guaranteed to be consistent). Thus, the only way that $\boldsymbol{d}_i$ for some $i \in [N]$ contains an incorrect multiplication triple is if *all* of the C opened triples were correct and the entire bucket $\boldsymbol{D}_i$ contains incorrect multiplication triples. Denote the event that this happens for some i by bad. By choosing B and C so that $\Pr[\mathsf{bad}]$ is negligible, the protocol is secure. Observe that the triples are all generated and fixed before $\mathcal{F}_{\text{perm}}$ is called, and thus the probability that bad occurs is equal to the balls-and-buckets game of [18, 5], where the adversary wins only if there exists no "mixed bucket" (containing both good and bad balls). In [5, Proof of Lemma 12], it is shown that $\Pr[\mathsf{bad}] < 2^{-\sigma}$ when $(B-1)\log_2 C \geq \sigma$ (these parameters are derived from their notation by setting $C = \ell$, $B = b$ and $N = \ell b^2$). For any σ, we therefore choose B and C such that $N = CB^2$ and $(B-1)\log_2 C \geq \sigma$, and the appropriate error probability is obtained.

Observe that simulation is easy; $\mathcal{S}$ receives N triples from the trusted party and simulates the $2(NB+C)$ calls to $\mathcal{F}_{\text{rand}}$. $\mathcal{S}$ places the appropriate values from the N triples in random places, and ensures that they will all be the first triple in each bucket (by setting the output of $\mathcal{F}_{\text{perm}}$ appropriately). Then, $\mathcal{S}$ sends $\mathsf{continue}$ to the trusted party if and only if $\mathcal{A}$ did not cheat in any multiplication. The only difference between the protocol execution with $\mathcal{A}$ and the simulation with $\mathcal{S}$ is in the case that bad occurs, which happens with negligible probability. ■

Concrete parameters. In our protocol, generation of triples is highly efficient. Thus, we can generate a very large number of triples at once (unlike [18]) which

yields better parameters. In [18, Proof of Theorem 8] it was shown that when the probability of a triple being incorrect is 1/2, the adversary can cheat with probability at most $2^{-\sigma}$ when $(1 + \log_2 N)(B - 1) \geq \sigma$. This implies that for $N = 2^{20}$ and $\sigma = 40$, we can take $B = 3$ because $(1+\log_2 N)(B-1) > 21 \cdot 2 > 40$. In order to make the probability of a triple being incorrect be (close to) 1/2, we can set $C = 3 \cdot 2^{20}$. This implies that the overall number of triples required is $6 \cdot 2^{20}$.

An improved combinatorial analysis is provided in [5]. They show that when setting $N = CB^2$, the adversary can cheat with probability at most $2^{-\sigma}$ when $(B - 1) \log_2 C \geq \sigma$ (in [5], they write ℓ instead of C and b instead of B). In order to minimize the number of triples overall, B must be kept to a minimum. For $N = 2^{20}$ and $\sigma = 40$, one can choose $B = 4$ and $C = 2^{16}$. It then follows that $(B - 1) \log_2 C = 3 \cdot 16 > 40$. Thus, the overall number of triples required is $NB + C = CB^3 + C = 2^{22} + 2^{16}$. Observe that these parameters derived from the analysis of [5] yield approximately 2/3 the cost of $6 \cdot 2^{20}$ as required by the analysis of [18]. (This follows because $6 \cdot 2^{20} = \frac{3}{2} \cdot 2^{22}$.)

In Section 5 we provide a *new analysis* showing that it suffices to generate $3 \cdot 2^{20}+3$ triples. This is approximately 25% less than the analysis of [5]. Concretely, to generate 1 million validated triples, the analysis of [5] requires generating 4,065,536 triples initially, whereas we require only 3,000,003.

Deferring compareview**.** In the calls to functionalities $\mathcal{F}_{\text{rand}}$ and $\mathcal{F}_{\text{perm}}$ and in the execution of Protocol 2.24, the parties use the compareview subprocedure. As explained before, the parties actually compare only at the end of the entire triple-generation protocol, and compare a hash of the view instead of the entire view, which reduces communication significantly.

Using pseudorandomness in $\mathcal{F}_{\text{perm}}$. In practice, in order to reduce the communication, the calls to $\mathcal{F}_{\text{coin}}$ inside $\mathcal{F}_{\text{perm}}$ are only used in order to generate a short seed. Each party then applies a pseudorandom generator to the seed in order to obtain all of the randomness needed for computing the permutation. This protocol actually no longer securely computes $\mathcal{F}_{\text{perm}}$ since the permutation is not random, and the corrupted party actually knows that it is not random since it has the seed. Nevertheless, by the proof of Proposition 3.3, we have that the only requirement from $\mathcal{F}_{\text{perm}}$ is that the probability of bad happening is negligible. Now, since the triples are fixed before $\mathcal{F}_{\text{perm}}$ is called, the probability of bad happening is simply the probability that a specific subset of permutations occur (that map all of the incorrect triples into the same bucket/s). If this occurs with probability that is non-negligibly higher than when a truly random permutation is used, then this can be used to distinguish the result of the pseudorandom generator from random.

4 Secure Computation of Any Functionality

In this section, we we show how to securely compute any three party functionality f. The idea behind the protocol is simple. The parties first use $\mathcal{F}_{\text{triples}}$ to generate

a vector of valid multiplication triples. Next, the parties compute the circuit using the semi-honest multiplication protocol of Section 2.2 for each AND gate. Recall that by Lemma 2.4, the result of this protocol is a triple of consistent shares $([a],[b],[c])$ where $c = ab$ or $c = ab \oplus 1$, even when one of the parties is malicious. Thus, it remains to verify that for each gate it holds that $c = ab$. Now, utilizing the valid triples generated within $\mathcal{F}_{\text{triples}}$, the parties can use Protocol 2.24 to verify that $([a],[b],[c])$ is a correct multiplication triple (i.e., that $c = ab$) without revealing anything about a, b or c. See Protocol 4.2 for a full specification.

We prove the following theorem:

Theorem 4.1. *Let f be a three-party functionality. Then, Protocol 4.2 securely computes f with abort in the $(\mathcal{F}_{\text{triples}}, \mathcal{F}_{\text{share}}, \mathcal{F}_{\text{reconst}})$-hybrid model, in the presence of one malicious party.*

Proof Sketch: Intuitively, the security of this protocol follows from the following. If the adversary cheats in any semi-honest multiplication, then by Lemma 2.25 the honest parties output $\perp$. This holds because by Lemma 2.4 all shares $[x],[y],[z]$ are consistent (but $z \neq xy$), while $([a_k],[b_k],[c_k])$ is guaranteed to be a correct multiplication triple since it was generated by $\mathcal{F}_{\text{triples}}$. Thus, the adversary must behave honestly throughout, and the security is reduced to the proof of security for the semi-honest case, as proven in [1].

The simulator for $\mathcal{A}$ works by playing the role of the trusted party for $\mathcal{F}_{\text{triples}}$ and $\mathcal{F}_{\text{share}}$, and then by simulating the semi-honest multiplication protocol in the circuit emulation phase for every AND gate. The verification stage involving executions of Protocol 2.24 is then simulated by $\mathcal{S}$ internally handing random ρ, σ values to $\mathcal{A}$ as if sent by P_{i-1}. Since $\mathcal{S}$ plays the trusted party in $\mathcal{F}_{\text{triples}}$ and $\mathcal{F}_{\text{share}}$, it knows all of the values held and therefore can detect if $\mathcal{A}$ tries to cheat. If yes, then it simulates the honest parties aborting, and sends $\perp$ to the trusted party as P_i's input. Otherwise, it sends the input of P_i sent by $\mathcal{A}$ in $\mathcal{F}_{\text{share}}$. Finally, after receiving P_i's output from the trusted party computing f, $\mathcal{S}$ plays the ideal functionality computing $\mathcal{F}_{\text{reconst}}$. If $\mathcal{A}$ sends abort to $\mathcal{F}_{\text{reconst}}$ then $\mathcal{S}$ sends abort to the trusted party computing f; otherwise, it sends continue. We stress that the above simulation works since the semi-honest multiplication protocol is *private in the presence of a malicious adversary*, meaning that its view can be simulated before any output is revealed. This was shown in [1, Section 4]. Thus, the view of a malicious $\mathcal{A}$ in the circuit-emulation phase is simulated as in [1], and then the verification phase is simulated as described above. This completes the proof sketch. ■

Generating many triples in the offline. In many cases, the circuit being computed is rather small. However, the highest efficiency in $\mathcal{F}_{\text{triples}}$ is achieved when taking a very large N (e.g., $N = 2^{20}$). We argue that $\mathcal{F}_{\text{triples}}$ can be run once, and the triples used for multiple different computations. This is due to the fact that the honest parties abort if *any* multiplication is incorrect, and this makes no difference whether a single execution utilizing N gates is run, or multiple executions.

PROTOCOL 4.2 (Securely Computing a Functionality f)

- **Inputs:** Each party P_j where $j \in \{1,2,3\}$ holds an input $x_j \in \{0,1\}^\ell$.
- **Auxiliary Input:** The parties hold a description of a boolean circuit C that computes f on inputs of length ℓ. Let N be the number of gates in C.
- **The protocol – offline phase:** The parties call $\mathcal{F}_{\text{triples}}$ with input N and obtain a vector $\boldsymbol{d}$ of sharings.
- **The protocol – online phase:**
  1. *Sharing the inputs:* For each input wire, the parties call $\mathcal{F}_{\text{share}}$ with the dealer being the party whose input is associated with that wire.
  2. *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:
     - If G_k is a XOR gate: Given shares $[x]$ and $[y]$ on the input wires, the parties compute $[x] \oplus [y]$ and define the result as their share on the output wire.
     - If G_k is a NOT gate: Given shares $[x]$ on the input wire, the parties compute $\overline{[x]}$ and define the result as their share on the output wire.
     - If G_k is an AND gate: Given shares $[x]$ and $[y]$ on the input wires, the parties run the semi-honest multiplication protocol of Section 2.2.
  3. *Verification stage: Before* the secrets on the output wires are reconstructed, the parties verify that all the multiplications were carried out correctly, as follows. For $k = 1, \ldots, N$:
     (a) Denote by $([x], [y])$ the shares of the input wires to the kth AND gate, and denote by $[z]$ the shares of the output wire of the kth AND gate.
     (b) The parties run Protocol 2.24 (triple verification using another without opening) on input $([x], [y], [z])$ and $([a_k], [b_k], [c_k])$ to verify $([x], [y], [z])$.
     (c) If a party did not output accept in every execution, it sends $\perp$ to the other parties and outputs $\perp$.

     Observe that all executions of Protocol 2.24 can be run in parallel. In addition, compareview can be run once at the end of all checks, and using a hash of the view as described in Section 3.
  4. If any party received $\perp$ in any call to any functionality above, then it outputs $\perp$ and halts.
  5. *Output reconstruction:* For each output wire of the circuit, the parties call $\mathcal{F}_{\text{reconst}}$ with input $([v], j)$ where $[v]$ is the sharing of the value on the output wire, and P_j is the party whose output is on the wire.
  6. If a party received $\perp$ in any call to $\mathcal{F}_{\text{reconst}}$ 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.

5 Improved Combinatorial Analysis

In this section we provide a tighter analysis of the probability that the adversary succeeds in circumventing the computation without being caught. This analysis

allows us to reduce both the overall number of triples needed and the number of triples that are opened in the cut-and-bucket process, compared to [18] and [5]. In our specific protocol, reducing the number of triples to be opened is of great importance since generation of a triple requires 3 bits of communication, while each opening requires 9 bits.

Loosely speaking, the adversary can succeed if the verification of an incorrect AND gate computation is carried using an incorrect multiplication triple. This event can only happen if no incorrect triples were opened and if the *entire bucket* from which the incorrect triple came contained only incorrect triples. Otherwise, the honest parties would abort in the triples-generation protocol, when running the check phase. Since the triples are randomly assigned to buckets, the probability that this event occurs is small (which is what we need to prove). Clearly, increasing the number of triples checked and the bucket size reduces the success probability of the adversary. However, increasing these parameters raises the computation and communication cost of our protocol. Thus, our goal is to minimize these costs by minimizing the number of triples generated (and opened). We denote by σ the statistical parameter, and our aim to guarantee that the adversary succeeds with probability at most $2^{-\sigma}$. Recall that C is the number of triples opened in the cut-and-bucket process and B is the size of the bucket. We start by defining the following balls-and-buckets game, which is equivalent to our protocol (in the game, a "bad ball" is an incorrect multiplication triple). We say that the adversary "wins", if the output of the game is 1.

$\text{Game}_1(\mathcal{A}, N, B, C)$**:**

1. The adversary $\mathcal{A}$ prepares $M = NB + C$ balls. Each ball can be either **bad** or **good**.
2. C random balls are chosen and opened. If one of the C balls is **bad** then output 0. Otherwise, the game proceeds to the next step.
3. The remaining NB balls are randomly divided into N buckets of equal size B. Denote the buckets by $B_1, ..., B_N$. We say that a bucket is **fully bad** if all balls inside it are **bad**. Similarly, a bucket is **fully good** if all balls inside it are **good**.
4. The output of the game is 1 if and only if there exists i such that bucket B_i is fully bad, and all other buckets are either **fully bad** or **fully good**.

Note that the condition in the last step forces the adversary to choose at least one bad ball if it wishes to win. We first show that for $\mathcal{A}$ to win the game, the number of bad balls $\mathcal{A}$ chooses must be a multiple of B, the size of a bucket.

Lemma 5.1. *Let T be the number of bad balls chosen by the adversary $\mathcal{A}$. Then, a necessary condition for $Game_1(\mathcal{A}, N, B, C) = 1$ is that $T = B \cdot t$ for some $t \in \mathbb{N}$.*

Proof: This follows immediately from the fact that the output of the game is 1 only if no bad balls are opened and all buckets are fully bad or good. Thus, if $T \neq B \cdot t$, then either a bad ball is opened or there must be some bucket that is mixed, meaning that it has both bad and good balls inside it. Therefore, in this case, the output of the game will be 0. ■

Following Lemma 5.1, we derive a formula for the success probability of the adversary in the game. We say that the adversary $\mathcal{A}$ has chosen to "corrupt" t buckets if $t = \frac{T}{B}$, where T is the number of bad balls generated by the adversary.

Theorem 5.2. *Let t be the number of buckets $\mathcal{A}$ has chosen to corrupt. Then, for every $0 < t \leq N$ it holds that*

$$\Pr[Game_1(\mathcal{A}, N, B, C) = 1] = \binom{N}{t}\binom{NB+C}{tB}^{-1}.$$

Proof: Assume $\mathcal{A}$ has chosen to corrupt t buckets. i.e., $\mathcal{A}$ has generated tB bad balls. Let E_c be the event that no bad balls were detected when opening C random balls. We have:

$$\Pr[E_c] = \frac{\binom{NB+C-tB}{C}}{\binom{NB+C}{C}} = \frac{(NB+C-tB)!(NB)!}{(NB-tB)!(NB+C)!}.$$

Next, let E_B the event that the bad balls are in exactly t buckets after permuting the balls (and so there are t fully bad buckets and all other buckets are fully good). There are $(NB)!$ ways to permute the balls, but if we require the tB bad balls to fall in exactly t buckets then we first choose t buckets out of N, permute the tB balls inside them, and finally permute the other $NB - tB$ balls in the other buckets. Overall, we obtain that

$$\Pr[E_B] = \frac{\binom{N}{t}(tB)!(NB-tB)!}{(NB)!}.$$

Combining the above, we obtain that

$$\begin{aligned}\Pr[\text{Game}_1(\mathcal{A}, N, B, C) = 1] &= \Pr[E_c \wedge E_B] = \Pr[E_c] \cdot \Pr[E_B] \\ &= \binom{N}{t}\frac{(NB+C-tB)!(tB)!}{(NB+C)!} \\ &= \binom{N}{t}\binom{NB+C}{tB}^{-1}.\end{aligned}$$

■

Next, we show that if $C \geq B$, then the best strategy of the adversary is to corrupt exactly one bucket. This allows us to derive an upper bound of the success probability of the adversary.

Theorem 5.3. *If $C \geq B$, then for every adversary $\mathcal{A}$, it holds that*

$$\Pr[Game_1(\mathcal{A}, N, B, C) = 1] \leq N\binom{NB+C}{B}^{-1}.$$

Proof: Following Theorem 5.2, we need to show that for *every* $t \geq 1$

$$\binom{N}{t}\binom{NB+C}{tB}^{-1} \leq N\binom{NB+C}{B}^{-1}.$$

First, observe that when $t = 1$, the left side of the inequality is exactly the same as its right side, and thus the theorem holds.

Next, assume that $t \geq 2$; we need to show that:

$$\binom{N}{t}\binom{NB+C}{tB}^{-1} < \binom{NB+C}{B}^{-1}$$

which is equivalent to proving that

$$\binom{N}{t}\frac{(tB)!(NB+C-tB)!}{(NB+C)!} < \frac{B!(NB+C-B)!}{(NB+C)!}$$

which is in turn equivalent to proving that

$$\binom{N}{t}\frac{(tB)!}{B!} < \frac{(NB+C-B)!}{(NB+C-tB)!}.$$

By multiplying both sides of the inequality with $\frac{1}{(tB-B)!}$ we obtain that in order to complete the proof, it suffices to show that

$$\binom{N}{t}\binom{tB}{tB-B} < \binom{NB+C-B}{tB-B}. \tag{1}$$

Using the assumption that $C \geq B$, we obtain that instead of proving Eq. (1), it is sufficient to prove that

$$\binom{N}{t}\binom{tB}{tB-B} < \binom{NB}{tB-B}. \tag{2}$$

To see that Eq. (2) holds, consider the following two combinatorial processes:

1. Choose t buckets out of N. Then, choose $tB - B$ out of the tB balls in these buckets.
2. Choose $tB - B$ out of NB balls.

Note that since $t \geq 2$, it holds that $tB - B > 0$, and the processes are well defined. Next, observe that both processes end with holding $tB - B$ balls that were chosen from an initial set of NB balls. However, while in the second process we do not place any restriction on the selection process, in the first process we require that t buckets will be chosen first and then the $tB - B$ balls are allowed to be chosen only from the t buckets. Thus, the number of choice options in the second process is strictly larger than in the first process. Finally, since the first process describes exactly the left side of Eq. (2), whereas the second process describes exactly the right side of Eq. (2), we conclude that the inequality indeed holds. ■

Corollary 5.4. *If $C = B$ and B, N are chosen such that $\sigma \leq \log\left(\frac{(N\cdot B+B)!}{N\cdot B!\cdot(N\cdot B)!}\right)$, then for every adversary $\mathcal{A}$ it holds that* $\Pr[Game_1(\mathcal{A}, N, B, C) = 1] \leq 2^{-\sigma}$.

Proof: This holds directly from Theorem 5.3, which holds if $C \geq B$. Since this holds for any $C \geq B$, we set $C = B$ in the bound of Theorem 5.3, and have that the bound is fulfilled if

$$N\binom{NB+B}{B}^{-1} \leq 2^{-\sigma} \quad \text{and so} \quad \frac{1}{N}\cdot\binom{NB+B}{B} \geq 2^{\sigma}.$$

Taking log of both sides yields the result. ■

Corollary 5.4 provides us a way of computing the bucket size B for every possible N and σ. For example, setting $N = 2^{20}$ and $\sigma = 40$ (meaning that we want to output 2^{20} triples from the pre-processing protocol with error probability less than 2^{-40}), we obtain that $B = 3$ suffices and we need to generate $NB + C = 3 \cdot 2^{20} + 3$ triples, of which only 3 triples are opened. We performed this computation for $N = 2^{20}, 2^{30}$ and $\sigma = 40, 80, 128$ and compared the results with [5] (recall that according to their analysis, when setting $N = CB^2$, the adversary can cheat with probability at most $2^{-\sigma}$ when $(B-1)\log_2 C \geq \sigma$). The comparison is presented in Tables 1, 2 and 3. In the tables, we use "M" to denote the number of triples that are initially generated, i.e., $M = NB + C$ in our work whereas $M = CB^3 + C$ in [5].

	$N = 2^{20}$			$N = 2^{30}$		
	B	M	C	B	M	C
[5]	4	4,259,840	65,536	3	3,340,530,119	119,304,647
Our work	3	3,145,731	3	3	3,221,225,475	3

Table 1. Parameter comparison for $\sigma = 40$

	$N = 2^{20}$			$N = 2^{30}$		
	B	M	C	B	M	C
[5]	7	7,361,432	21,400	5	5,411,658,793	42,949,673
Our work	5	5,242,885	5	4	4,294,967,300	4

Table 2. Parameter comparison for $\sigma = 80$

	$N = 2^{20}$			$N = 2^{30}$		
	B	M	C	B	M	C
[5]	10	10,496,246	10,486	6	6,427,277,106	29,826,162
Our work	7	7,340,039	7	5	5,368,709,125	5

Table 3. Parameters comparison for $\sigma = 120$

As can be seen, in all cases, our combinatorial analysis yields a significant improvement, both in the number of generated triples and in the number of triples needed to be opened. Specifically, although only few triples are opened, we reduce the overall number of triples generated by up to 25%, compared to [5]. Recall that both improvements are important, as each triple less to generate means 1 bit less to send for each party, and each triple less to open means 3 bit less to send for each party.

6 Efficiency and Comparison

In this section, we describe the communication and computation complexity of (the computationally secure variant of) our protocol. We compare our protocol to that of [17], since this is the most efficient protocol known for our setting of three parties, malicious adversaries, and an honest majority. The complexity of the protocol in [17] is close to Yao's two-party semi-honest protocol, and thus its communication complexity is dominated by the size of the garbled circuit and its computation complexity is dominated by the amount of work needed to prepare a garbled circuit and evaluate it. The comparison summary is shown in Table 4; for our protocol, the complexity is based on a bucket size of $B = 3$. A detailed explanation appears below.

	Communication (bits)	**Number of AES computations**
Our protocol	$10N$	$\frac{20N+3N\log(3N))}{128} \approx \frac{N}{5}$
The protocol of [17]	$85N$	$3N$

Table 4. Average cost per party (N = number of AND gates in the circuit)

Communication Complexity. We count the number of bits sent by each party for each AND gate. The semi-honest multiplication protocol requires sending a single bit, and verifying a triple using another without opening requires sending 2 bits (only very few triples are checked with opening and so we ignore this). Now, a single multiplication and a single verification is used for each AND gate (3 bits). Furthermore, each triple is generated from B triples (generated using B multiplications) and $B-1$ verifications, thereby costing $B+2(B-1)$ bits. The overall cost per gate is therefore $B+2B-2+3 = 3B+1$. For $B = 3$ (which suffices with $N = 2^{20}$ and error 2^{-40}), we conclude that only 10 bits are sent by each party per gate.

In contrast, in [17], the communication is dominated by a single garbled circuit. When using the half-gates optimization of [23], such a circuits consists of two ciphertexts per AND gate with a size of 256 bits. Thus, on average, each party sends $256/3 \approx 85$ bits per AND gate.

Number of AES computations. The computations in our protocol are very simple, and the computation complexity is therefore dominated by the AES computations needed to generated randomness (in the multiplication and to compute correlated randomness). Two bits of pseudorandomness are needed for each call to $\mathcal{F}_{\text{cr}}$ to generate correlated randomness. In order to generate a triple, 2 calls to $\mathcal{F}_{\text{cr}}$ are required and an additional call for the multiplication, for a total of 6 bits. For every AND gate, B triples are generated and one additional multiplication is carried out for the actual gate, resulting in a total of $6B+2$ bits. In addition, $\mathcal{F}_{\text{perm}}$ requires an additional $NB\log(NB)$ bits of pseudorandomness. Thus, the total number of pseudorandom bits for N gates equals $(6B+2)N+NB\log(NB)$; taking $B = 3$ as above, we have $20N + 3N(\log 3N)$. Noting that 128-bits of

pseudorandomness are generated with a single AES computation, this requires $\frac{20N+3N(\log 3N)}{128} \approx \frac{N}{5}$ calls to AES.

In contrast, in the protocol of [17], two parties need to garble the circuit and one needs to evaluate it. Garbling a circuit requires 4 AES operations per AND gate and evaluating requires 2 operations per AND gate. Thus, the average number of AES operations is $\frac{10}{3} \approx 3$ per party per AND gate.

References

1. T. Araki, J. Furukawa, Y. Lindell, A. Nof and K. Ohara. High-Throughput Semi-Honest Secure Three-Party Computation with an Honest Majority. To appear in the 23*rd ACM CCS*, 2016.
2. D. Beaver. Efficient Multiparty Protocols Using Circuit Randomization. In *CRYPTO 1991*, Springer (LNCS 576), pages 420–432, 1992.
3. D. Beaver, S. Micali, and P. Rogaway. The round complexity of secure protocols. In the *22nd STOC*, pages 503–513, 1990.
4. M. Ben-Or, S. Goldwasser, A. Wigderson. Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation. *STOC 1988*: 1-10
5. 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.
6. R. Canetti. Security and Composition of Multiparty Cryptographic Protocols. *Journal of Cryptology*, 13(1):143–202, 2000.
7. R. Canetti. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In 42*nd FOCS*, pages 136–145, 2001.
8. D. Chaum, C. Crépeau and I. Damgård. Multi-party Unconditionally Secure Protocols. In 20*th STOC*, pages 11–19, 1988.
9. I. Damgård, V. Pastro, N.P. Smart and S. Zakarias. Multiparty Computation from Somewhat Homomorphic Encryption. In *CRYPTO 2012*, Springer (LNCS 7417), pages 643–662, 2012.
10. R.A. Fisher and F. Yates. *Statistical Tables for Biological, Agricultural and Medical Research* (3rd ed.). Oliver & Boyd. pages 26-27, 1938.
11. O. Goldreich. *Foundations of Cryptography: Volume 2 – Basic Applications*. Cambridge University Press, 2004.
12. O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. *19th STOC*, 218–229, 1987.
13. S. Goldwasser and Y. Lindell. Secure Computation Without Agreement. In the *Journal of Cryptology*, 18(3):247-287, 2005.
14. E. Kushilevitz, Y. Lindell and T. Rabin. Information-Theoretically Secure Protocols and Security Under Composition. In the *SIAM Journal on Computing*, 39(5):2090–2112, 2010.
15. Y. Lindell and B. Pinkas. Secure Two-Party Computation via Cut-and-Choose Oblivious Transfer. In the *8th TCC*, Springer (LNCS 6597), 329–346, 2011.
16. Y. Lindell and B. Riva. Blazing Fast 2PC in the Offline/Online Setting with Security for Malicious Adversaries. In *ACM Conference on Computer and Communications Security*, pages 579–590, 2015.
17. P. Mohassel, M. Rosulek and Y. Zhang. Fast and Secure Three-party Computation: The Garbled Circuit Approach. In *ACM Conference on Computer and Communications Security*, pages 591–602, 2015.

18. J.B. Nielsen, P.S. Nordholt, C. Orlandi and S.S. Burra. A New Approach to Practical Active-Secure Two-Party Computation. In *CRYPTO 2012*, Springer (LNCS 7417), pages 681–700, 2012.
19. T. Rabin and M. Ben-Or. Verifiable Secret Sharing and Multi-party Protocols with Honest Majority. In 21*st STOC*, pages 73–85, 1989.
20. P. Rindal and M. Rosulek. Faster Malicious 2-Party Secure Computation with Online/Offline Dual Execution. In *USENIX Security Symposium*, pages 297–314, 2016.
21. Sharemind, Cybernetica. `https://sharemind.cyber.ee`.
22. A. Yao. How to Generate and Exchange Secrets. In the 27*th FOCS*, pages 162–167, 1986.
23. S. Zahur, M. Rosulek and D. Evans: Two Halves Make a Whole – Reducing Data Transfer in Garbled Circuits Using Half Gates. In *EUROCRYPT 2015*, Springer (LNCS 9057), pages 220–250, 2015.

A Definition of Security

The security parameter is denoted n; negligible functions and computational indistinguishability are defined in the standard way, with respect to non-uniform polynomial-time distinguishers.

Ideal versus real model definition. We use the ideal/real simulation paradigm in order to define security, where an execution in the real world is compared to an execution in an ideal world where an incorruptible trusted party computes the functionality for the parties [6, 11]. For the sake of clarity, we present our definition specifically for the three-party case with one party corrupted. We define *security with abort* (and without fairness), meaning that the corrupted party may receive output while the honest parties do not. Our definition does *not* guarantee *unanimous abort*, meaning that some honest party may receive output while the other does not. It is easy to modify our protocols so that the honest parties unanimously abort by running a single (weak) Byzantine agreement at the end of the execution [13]; we therefore omit this step for simplicity.

Note that with an honest majority, it is possible to achieve fairness (assuming a broadcast channel). Nevertheless, our protocol does not guarantee this, and we do not know how to modify it to guarantee fairness without significantly sacrificing efficiency,

The real model. In the real model, a three-party protocol π is executed by the parties. For simplicity, we consider a synchronous network that proceeds in rounds and a rushing adversary, meaning that the adversary receives its incoming messages in a round before it sends its outgoing message.[3] The adversary $\mathcal{A}$ can

[3] This modeling is only for simplicity, since in our protocol, all parties receive and send messages in each round. Thus, by instructing each party to only send their round $i+1$ messages after receiving all round-i messages, we have that an execution of the protocol in an asynchronous network is the same as for a rushing adversary in a synchronous network. Note that we do not guarantee output delivery, so "hanging" of the protocol is also allowed.

be malicious; it sends all messages in place of the corrupted party, and can follow any arbitrary strategy. The honest parties follow the instructions of the protocol.

Let $\mathcal{A}$ be a non-uniform probabilistic polynomial-time adversary controlling party P_i for $i \in \{1,2,3\}$. Let $\text{REAL}_{\pi,\mathcal{A}(z),i}(x_1,x_2,x_3,n)$ denote the output of the honest parties and $\mathcal{A}$ in an real execution of π, with inputs x_1,x_2,x_3, auxiliary-input z for $\mathcal{A}$, and security parameter n.

The ideal model. We define the ideal model, for any (possibly reactive) functionality $\mathcal{F}$, receiving inputs from P_1,P_2,P_3 and providing them outputs. Let $i \in \{1,2,3\}$ be the index of the corrupted party controlled by the adversary. The ideal execution proceeds as follows:

- **Send inputs to the trusted party:** Each honest party P_j sends its specified input x_j to the trusted party. The corrupted party P_i controlled by the adversary may either send its specified input x_i, some other x_i' or an abort message.
- **Early abort option:** If the trusted party received abort from the corrupted P_i, it sends $\perp$ to all messages and terminates. Otherwise, it proceeds to the next step.
- **Trusted party sends output to the adversary:** The trusted party computes each party's output as specified by the functionality $\mathcal{F}$ based on the inputs received; denote the output of P_j by y_j. The trusted party then sends y_i to the corrupted party.
- **Adversary instructs trusted party to continue or halt:** For each $j \in \{1,2,3\}$ with $j \neq i$, the corrupted party sends the trusted party either abort_j or $\mathsf{continue}_j$. For each $j \neq i$:
  - If the trusted party received abort_j then it sends P_j the abort value $\perp$ for output.
  - If the trusted party received $\mathsf{continue}_j$ then it sends P_j its output value y_j.
- **Outputs:** The honest parties always output the output value they obtained from the trusted party, and the corrupted party outputs nothing.

Let $\mathcal{S}$ be a non-uniform probabilistic polynomial-time adversary controlling party P_i for $i \in \{1,2,3\}$. Let $\text{IDEAL}_{\mathcal{F},\mathcal{S}(z),i}(x_1,x_2,x_3,n)$ denote the output of the honest parties and $\mathcal{S}$ in an ideal execution with the functionality $\mathcal{F}$, inputs x_1,x_2,x_3 to the parties, auxiliary-input z to $\mathcal{S}$, and security parameter n.

Security. Informally speaking, the definition says that protocol π securely computes f if adversaries in the ideal world can simulate executions of the real world protocol. In some of our protocols there is a statistical error that is not dependent on the computational security parameter. As in [15], we formalize security in this model by saying that the distinguisher can distinguish with probability at most this error *plus* some factor that is negligible in the security parameter. This is formally different from the standard definition of security since the statistical error does not decrease as the security parameter increases.

Definition A.1. *Let $\mathcal{F}$ be a 3-party functionality, and let π be a 3-party protocol. We say that π* securely computes f with abort in the presence of one malicious party*, if for every non-uniform probabilistic polynomial-time adversary $\mathcal{A}$*

in the real world, there exists a non-uniform probabilistic polynomial-time simulator/adversary $\mathcal{S}$ in the ideal model with $\mathcal{F}$, such that for every $i \in \{1,2,3\}$,

$$\left\{\text{IDEAL}_{F,\mathcal{S}(z),i}(x_1,x_2,x_3,n)\right\} \stackrel{\text{c}}{\equiv} \left\{\text{REAL}_{\pi,\mathcal{A}(z),i}(x_1,x_2,x_3,n)\right\}$$

where $x_1,x_2,x_3 \in \{0,1\}^*$ under the constraint that $|x_1| = |x_2| = |x_3|$, $z \in \{0,1\}^*$ and $n \in \mathbb{N}$. We say that π securely computes f with abort in the presence of one malicious party with statistical error $2^{-\sigma}$ *if there exists a negligible function $\mu(\cdot)$ such that the distinguishing probability of the adversary is less than $2^{-\sigma} + \mu(n)$.*

The hybrid model. We prove the security of our protocols in a hybrid model, where parties run a protocol with real messages and also have access to a trusted party computing a subfunctionality for them. The modular sequential composition theorem of [6] states that one can replace the trusted party computing the subfunctionality with a real secure protocol computing the subfunctionality. When the subfunctionality is g, we say that the protocol works in the g-hybrid model.

Universal Composability [7]. Protocols that are proven secure in the universal composability framework have the property that they maintain their security when run in parallel and concurrently with other secure and insecure protocols. In [14, Theorem 1.5], it was shown that any protocol that is proven secure with a black-box non-rewinding simulator and also has the property that the inputs of all parties are fixed before the execution begins (called input availability or start synchronization in [14]), is also secure under universal composability. Since the input availability property holds for all of our protocols and subprotocols, it is sufficient to prove security in the classic stand-alone setting and automatically derive universal composability from [14]. We remark that this also enables us to call the protocol and subprotocols that we use in parallel and concurrently (and not just sequentially), enabling us to achieve more efficient computation (e.g., by running many executions in parallel or running each layer of a circuit in parallel).

Conditional security. In one of our constructions (securely computing $\mathcal{F}_{\text{reconst}}$ in Section 2.7), we only prove security on the condition that the honest parties' inputs fulfill some condition. Then, in our protocols, whenever $\mathcal{F}_{\text{reconst}}$ is called, it is always guaranteed that the condition is fulfilled. Thus, intuitively this suffices. We now explain formally why this is sufficient. The reason is that it is always possible to define the ideal functionality so that the trusted party first checks if the condition holds. If yes, then it computes the output as specified. Otherwise, it provides the adversary with the honest parties' inputs and lets it determine their outputs. Now, if a protocol securely computes the original functionality when the condition holds, then it securely computes the modified functionality *without any condition*. The reason for this is as follows: if the condition holds, then the protocol is secure by assumption and the original simulator can be used; if the condition does not hold, then the simulator receives the honest parties' inputs and so can perfectly simulate an execution by running the honest parties and then sending the functionality the output they would receive in a real execution.

Finally, observe that as long as the protocol is only used when the condition is fulfilled, this addition to the functionality makes no differences (since it is never called). We therefore conclude that it is possible to define conditional security and use the functionality with all the benefits of modular composition and so on.

B Proof of Proposition 2.7

Proposition B.1 (Proposition 2.7 – restated). *If F is a pseudorandom function, then Protocol 2.5 securely computes functionalities $\mathcal{F}^1_{\text{cr}}$ and $\mathcal{F}^2_{\text{cr}}$, respectively, with abort in the presence of one malicious party.*

Proof: The proof of this proposition is similar to the proof of Theorem 3.9 in [1]. Note that [1] considers the semi-honest model; however, since the protocol is non-interactive (beyond the setup phase), there is almost no difference.

We first consider $\mathcal{F}^1_{\text{cr}}$. Let $\mathcal{A}$ be the real-model adversary controlling party P_i. We construct a simulator $\mathcal{S}$ who interacts externally with the $\mathcal{F}_{cr}$ functionality and internally with $\mathcal{A}$, as follows:

1. $\mathcal{S}$ invokes $\mathcal{A}$, chooses a random key $k_{i-1} \in \{0,1\}^\kappa$ and internally hands it to $\mathcal{A}$ as if received from P_{i-1} in the setup phase.
2. Next, $\mathcal{S}$ receives the key k_i that it sends to P_{i+1} in the setup phase of Protocol 2.5. (Since $\mathcal{A}$ is rushing, $\mathcal{S}$ first provides $\mathcal{A}$ with k_{i-1} and then receives its message k_i.)
3. If $\mathcal{A}$ does not send a valid $k_i \in \{0,1\}^\kappa$, then $\mathcal{S}$ externally sends abort to $\mathcal{F}^1_{\text{cr}}$. Else, $\mathcal{S}$ externally sends $(k, k') = (k_{i-1}, k_i)$ to $\mathcal{F}^1_{\text{cr}}$.
4. From this point on (when computing the "generating randomness" phase) upon input id, $\mathcal{S}$ just outputs whatever $\mathcal{A}$ outputs.

We now show that for each input id, the joint output distribution of $\mathcal{S}$ and the honest parties in an ideal execution with $\mathcal{F}^1_{\text{cr}}$ is computationally indistinguishable from the joint output distribution of $\mathcal{A}$ and the honest parties in a real execution of Protocol 2.5 with adversary $\mathcal{A}$. In order to see this, observe that in the $\mathcal{F}^1_{cr}$ functionality the respective outputs of the honest parties P_{i-1}, P_{i+1} are random values $\alpha_{i-1}, \alpha_{i+1}$ under the constraint that $F_k(id) \oplus F_{k'}(id) \oplus \alpha_{i-1} \oplus \alpha_{i+1} = 0$, whereas in the real-world protocol the output of the honest parties is $F_k(id) \oplus F_{k''}(id)$ and $F_{k'}(id) \oplus F_{k''}(id)$ for a random key k'' unknown to the adversary (chosen by P_{i+1} and sent to P_{i-1}). By a straightforward reduction, the pseudorandom function $F_{k''}$ in the real execution can be replaced by a truly random function, implying that $F_{k''}(id)$ can be replaced by a truly random value α. Now, define $\alpha_{i-1} = F_k(id) \oplus \alpha$ and define $\alpha_{i+1} = F_{k'}(id) \oplus \alpha$. Observe that $\alpha_{i-1} \oplus \alpha_{i+1} = F_k(id) \oplus F_{k'}(id)$, and that by the way they are defined $\alpha_{i-1}, \alpha_{i+1}$ are random under this constraint. We therefore have that for any *fixed* k, k', the outputs of the honest parties are computationally indistinguishable in both executions. Since the entire view of $\mathcal{A}$ consists of k, k' and these are *identically distributed* in both cases, we have that the joint distributions are computationally indistinguishable, as required.

The proof for functionality $\mathcal{F}_{\text{cr}}^2$ is almost identical (the only difference being that the α values are sent separately instead of their XOR, but this makes no difference). ■

C Proof of Proposition 2.20

Proposition C.1 (Proposition 2.20 – restated). *Protocol 2.19 securely computes $\mathcal{F}_{\text{share}}$ with abort in the $(\mathcal{F}_{\text{rand}}, \mathcal{F}_{\text{reconst}})$-hybrid mode, in the presence of one malicious party.*

Proof: Let $\mathcal{A}$ be the adversary corrupting party P_i. We construct a simulator $\mathcal{S}$ separately for the following two cases:

- *Case 1 – P_j is honest (and so $i \neq j$)*: In this case, simulator $\mathcal{S}$ invokes $\mathcal{A}$ and obtains the value (t_i, s_i) that it intends to send to $\mathcal{F}_{\text{rand}}$ in the protocol. Then, $\mathcal{S}$ chooses a random bit $b \in \{0,1\}$. Next, $\mathcal{S}$ simulates the call to $\mathcal{F}_{\text{reconst}}$. Recall that the corrupted party has no input to $\mathcal{F}_{\text{reconst}}$ and only receives output. Furthermore, since $[v]$ was generated by $\mathcal{F}_{\text{rand}}$, the honest parties shares are guaranteed to be consistent. Thus, $\mathcal{S}$ simulates P_j's output by simply sending it the same pair (t_i, s_i), as $\mathcal{F}_{\text{reconst}}$ works in the case of consistent input (as proven in Proposition 2.17). Next, if $\mathcal{A}$ sends abort to $\mathcal{F}_{\text{reconst}}$, then $\mathcal{S}$ sends abort to $\mathcal{F}_{\text{share}}$ and simulates P_j sending $\perp$ to P_i. Else, $\mathcal{S}$ simulates P_j sending b to the corrupted P_i. Finally, $\mathcal{S}$ externally sends $\mathcal{F}_{\text{share}}$ the share of the corrupted party to be $(t_i, s_i \oplus b)$.
  We show that the joint distribution of $\mathcal{A}$'s view and the honest parties' outputs is identical in the real and ideal executions. $\mathcal{A}$'s view is clearly identical in both cases; we show that the output of the honest parties' given $\mathcal{A}$'s view are also identical. In the ideal simulation, the corrupted party's output share is $(t_i, s_i \oplus b)$, and the honest parties' output shares are those defined by $(t_i, s_i \oplus b)$ and the dealer's bit v. Specifically, we have that $t_{i-1} = s_i \oplus b \oplus v$ and $t_{i+1} = t_{i-1} \oplus t_i = s_i \oplus t_i \oplus b \oplus v$, and $s_{i-1} = t_{i-2} \oplus v = t_{i+1} \oplus v$ and $s_{i+1} = t_i \oplus v$; see Claim 2.1.
  In a real execution, the honest parties' outputs from $\mathcal{F}_{\text{rand}}$ are those defined by (t_i, s_i) and a random a. The dealer P_j then sends $b = a \oplus v$ and the honest parties set their output as $[v] = [a] \oplus b$. The t-parts of the shares held by the honest parties after $\mathcal{F}_{\text{rand}}$ are $t_{i-1} = s_i \oplus a$ and $t_{i+1} = s_i \oplus t_i \oplus a$. Furthermore, $s_{i-1} = t_{i+1} \oplus a$ and $s_{i+1} = t_i \oplus a$. After the dealer sends $b = v \oplus a$, the honest parties XOR a into the "s" part of their share. Thus, the *output* of the honest parties after the entire protocol are $t_{i-1} = s_i \oplus a$ and $s_{i-1} = (t_{i+1} \oplus a) \oplus b = t_{i+1} \oplus v$, and $t_{i+1} = s_i \oplus t_i \oplus a$ and $s_{i+1} = (t_i \oplus a) \oplus b = t_i \oplus v$.
  Observing that $a = b \oplus v$, we have that these are the *exact* same values as in the ideal execution.
- *Case 2 - P_j is corrupt (and so $i = j$):* In this case, as above, $\mathcal{S}$ invokes $\mathcal{A}$ and obtains the value (t_i, s_i) that it intends to send to $\mathcal{F}_{\text{rand}}$. Next, $\mathcal{S}$ chooses a random $a \in \{0,1\}$ and simulates the call to $\mathcal{F}_{\text{reconst}}$ by internally

handing $\mathcal{A}$ the bit a (and the pair (t_i, s_i) which the corrupted party always receives). Finally, $\mathcal{S}$ receives bits b, b' that $\mathcal{A}$ intends to send to the honest parties P_{i+1} and P_{i-1}, respectively. If $b \neq b'$, then $\mathcal{S}$ simulates the honest parties aborting, and sends abort to $\mathcal{F}_{\text{share}}$. Otherwise, $\mathcal{S}$ sends (t_i, s_i, v) to $\mathcal{F}_{\text{share}}$, where $v = a \oplus b$.

As above, it is clear that $\mathcal{A}$'s view is identical in both the real and ideal executions. In addition, since (t_i, s_i, v) fully determine the honest parties' shares, and in both the real and ideal executions the parties hold shares of v defined by (t_i, s_i), we have that the joint distribution of the adversary's view and honest parties' outputs is identical in both cases.

This concludes the proof. ■