From Passive to Covert Security at Low Cost

Ivan Damgård, Martin Geisler, and Jesper Buus Nielsen

Dept. of Computer Science, University of Aarhus, Denmark
{ivan,mg,jbn}@cs.au.dk

Abstract. Aumann and Lindell defined security against *covert attacks*, where the adversary is malicious, but is only caught cheating with a certain probability, where the idea is that in many real-world cases, a large probability of being caught is sufficient to prevent the adversary from trying to cheat. In this paper, we show how to compile a passively secure protocol for honest majority into one that is secure against covert attacks, again for honest majority and catches cheating with probability 1/4. The cost of the modified protocol is essentially twice that of the original plus an overhead that only depends on the number of inputs.

1 Introduction

When studying cryptographic protocols, the behavior of the adversary has traditionally been categorized as being either *semi-honest* (passive) or *malicious* (active). A semi-honest adversary will only listen in on the network communication and spy passively on the internal state of the corrupted protocol participants. At the other end of the spectrum, a malicious adversary can make corrupted parties behave arbitrarily and will try to actively disrupt the computation in order to gain extra information and/or cause incorrect results.

Aumann and Lindell [2] introduces a third type of adversary called a *covert* adversary. This is intuitively an adversary which is able to do an active attack, but will behave correctly if the chance of being caught is sufficiently large—even if that chance is not essentially 1. The argument for studying covert adversaries is that there are many real world situations where the consequences of being caught out-weights the benefit of cheating—even a small but non-negligible risk of being caught is a deterrent. An example could be companies that agree to conduct an auction using secure multiparty computation. If a company is found to be cheating it may be subject to fines and it will hurt its long-term relationships with customers and other companies.

In the standard simulation-based definition of secure multiparty computation a protocol is said to *securely evaluate a function* f if no attack against the protocol can do better than an attack on an ideal process where an ideal functionality evaluates f and hands the result to the parties.

Aumann and Lindell [2] give three different models of what a covert adversary can do by defining two different ideal functionalities that may compute f as usual, but may also act differently, depending on what the adversary does. They also define what it means for a protocol to implement an ideal functionality

securely, this is a fairly standard simulation-based definition for sequentially composable protocols.

Thus, the special ingredient in the model that allows to accommodate covert attacks is only in the definition of the functionalities, which correspond to different levels of security, which are called *Explicit Cheat Formulation* (ECF) and *Strong Explicit Cheat Formulation* (SECF).[1] The basic idea in both cases is that the adversary may decide to try to cheat and must inform the functionality about this. The functionality then decides if the cheating is detected which happens with probability ε, where ε is the *deterrence factor*. In this case all parties are informed that some specific corrupt party cheated. Otherwise, with probability $1-\varepsilon$, the cheating is undetected, and there is no security guarantee anymore, the functionality gives all inputs to the adversary and lets him decide the outputs. The difference between the two variants is that for ECF, the adversary gets the inputs of honest parties and decides their outputs immediately when he decides to cheat. For SECF, this only happens if the cheat is not detected.

Thus, with ECF, the adversary is caught with probability ε, but will learn the honest parties' inputs even if he is caught. With SECF, he must try to cheat *and succeed* to learn anything he was not supposed to.

1.1 Our Contribution

In this paper we propose a new construction that "compiles" a passively secure protocol into a new protocol with covert security. The approach is generic, but for concreteness we describe the idea starting from the classical BGW protocol [6] for evaluating arithmetic circuits, and only give the full compiler in the appendix.

We assume honest majority and synchronous communication with secure point-to-point channels. We also assume a poly-time adversary, as we use cryptographic tools.

The basic idea is to first use a protocol with full active security to do a small amount of computation. Here, we will prepare two sets of (secret-shared) inputs to the passively secure protocol. However, only one set of sharings contains the actual inputs, while the other—the *dummy* shares—contain only zeros. Initially, it is unknown which set is the dummy one. Then we run the passively secure protocol on both sets of inputs until parties hold shares of the outputs, which they must commit to. Now we reveal which sharings contained dummy values, and everything concerning the dummy execution can be then made available to check that no cheating occurred here. If no cheating was detected, we open the outputs of the real execution.

The intuition is that the adversary has to decide whether to cheat without knowing which execution is the dummy one, and therefore we can catch him with probability $\frac{1}{2}$ if he cheats at all, so one would expect this to give a deterrence factor of $\frac{1}{2}$.

[1] They also have a so called Failed Simulation definition which is weaker and which we do not use here.

However, while the intuition is straightforward, there are several non-trivial technicalities to take care of to make this work. We need parties to be able to prove that they really sent/received a given message earlier, and we have to do the check at the end without introducing too much overhead. After solving these problems, we obtain a protocol with deterrence factor $\frac{1}{4}$ whose complexity is essentially twice that of the passive protocol plus the overhead involved in preparing the inputs (which does not depend on the size of the computation).

We note that we focus on the complexity we get when there is no deviation from the protocol. In our construction, the adversary can slow things down by a factor linear in the number of parties by deviating, but the protocol is still secure, the adversary can only make it fail if he runs the risk of actually cheating and hence of being caught. Now, the spirit of covert security is that the adversary is to some extent rational, he does not cheat because it does not pay off to do so. It seems to us that there is little benefit in practice for the adversary in only slowing things down, while he cannot learn extra information or influence the result. So we believe that the complexity in practice can be expected to be what we get when there is no deviation.

We show our protocol is secure by showing that it implements Aumann and Lindell's functionality *in the UC model* [9], i.e., we do not use their simulation notion. The only difference this makes is that we get a stronger composition property for our protocol.

We show that the classical passively secure protocol by Ben-Or et al. [6] can be compiled to give a protocol with SECF security. Our approach can be used in a more general way, to compile any passively secure protocols into a covert protocol, if the original protocol satisfies certain reasonable conditions. The conditions are essentially as follows. The protocol should be based on secret sharing and consist of a computation phase and a reconstruction phase.

Computation phase: The computation phase starts from sharings of the inputs and produces sharings of the outputs, where the view of $t < n/2$ passively corrupted parties is independent of the inputs being computed on.

Reconstruction phase: The reconstruction phase consists of a single message from each party to each other party—i.e., it is non-interactive.

Passive security: Suppose uniformly random sharings of the inputs are dealt by an ideal functionality. Consider the protocol that executes the computation phase on these sharings and then the reconstruction phase. This protocol should be passively secure against $t < n/2$ statically corrupted parties.

The approach to obtaining covert security is basically the same as described above. The details are described in Appendix A. If the computation phase leaks no information, even under active attack (as is the case for the BGW protocol), we get SECF security, otherwise ECF security is obtained.

1.2 Related Work and Discussion

Goyal et al. [14] improve Aumann and Lindell's 2-party protocol and also give a general multiparty computation protocol with covert security for the case of dishonest majority.

Our work focuses instead on honest majority. The skeptical reader may ask whether this is really interesting: the motivation for covert security is to settle for less than full robustness in return for more efficient protocols, and it may seem that we already know how to have great efficiency with honest majority and full active security. For instance, in [5, 10], it is shown that unconditionally secure evaluation of a circuit $\mathcal{C}$ for n parties and $t < n/3$ corruptions can be done in complexity $O(|\mathcal{C}|n)$ plus an overhead that only depends on the depth of the circuit, and in [12], it is shown under a computational assumption that this can be reduced to $O(|\mathcal{C}|)$ except for logarithmic factors plus an overhead that is independent of the circuit. Here, the security threshold can be arbitrarily close to $\frac{1}{2}$.

How could we hope to be better than that? There are two answers to this: First, the previous protocols are not as efficient as it may seem: the result from [12] only works asymptotically for a large number of parties and very large computations, it makes non black-box use of a pseudo-random function and is, in fact, very far from being practical. [5, 10] use only cheap information theoretic primitives, but the security threshold in non-optimal and there is an overhead implying that deep circuits are expensive.

However, these protocols can all become much simpler and more practical if we assume the adversary is passive. For instance, when the adversary is passive the protocols from [5, 10] can tolerate $t < n/2$ and no longer have an overhead that depends on the circuit depth. Our compiler works for any "reasonable" protocol that is based on secret sharing, so we can use it on these simpler passively secure protocols and get a protocol with covert security, but with efficiency and security threshold similar to the passively secure solutions.

The second answer is that general circuit evaluation is not the only application. There are many special purpose protocols that are designed for a passive adversary but where obtaining active security comes at a significant cost. One example is the protocol by Algesheimer et al. [1] for distributed RSA key generation. Another is the auction application described in [8]. In both cases the protocols do not go via evaluation of a circuit for the desired function, but gets significant optimizations by taking other approaches. We can use our construction here to get covert security at a cost essentially a factor of two.

2 Preliminaries

Aumann and Lindell [2] present three successively stronger notions of security in the presence of covert adversaries, of which we consider the two strongest ones, where the adversary is forced to decide whether to cheat without knowledge of the honest parties' inputs. As mentioned, these are called ECF and SECF and are defined by specifying two (very similar) ideal functionalities.

For convenience, we give the ECF and SECF functionalities here. The only differences from [2] is that we do not include an option for the adversary to abort the protocol, and also, if no cheating is detected, the adversary cannot stop the

functionality from giving outputs to the honest parties. This gives a stronger notion of security, and we can obtain it as we assume an honest majority.

Another difference is that we relax the requirements on the detection mechanism slightly. In [2] it is required that only one corrupted party is detected and that the honest parties agree on that party. We allow that several corrupted parties are detected and allow that different honest parties detect different sets of corrupted parties. The only requirement is that there is at least one corrupted party which is detected by all honest parties. In the presence of an honest majority, the stronger detection requirement in [2] can then be implemented using a Byzantine agreement at the end of the protocol on who should take the blame. We prefer to see this negotiation as external to the protocol and thus allow the more relaxed detection. See Fig. 1.

Let f be a function with n inputs and n outputs, where n is the number of parties. The ECF functionality $\mathcal{F}_{\text{ECF}}^{f}$ for function f with deterrence factor ε works as follows:

Inputs: Any honest party P_i sends input x_i to $\mathcal{F}_{\text{ECF}}^{f}$, while the adversary $\mathcal{A}$ sends input on behalf of the corrupted parties.

Cheat detection: Let $C \subset \{1, \ldots, n\}$ denote the indices of the corrupted parties and let $H = \{1, \ldots, n\} \setminus C$ be the honest parties. The adversary can at any time instruct $\mathcal{F}_{\text{ECF}}^{f}$ to give outputs of the form $(\texttt{corrupt}, j)$ for $j \in C$ to P_i with $i \in H$. For $i \in H$, let $J_i \subset C$ be the set of j for which P_i output $(\texttt{corrupt}, j)$.

Attempted cheat: If $\mathcal{F}_{\text{ECF}}^{f}$ receives $(\texttt{cheat})$ from $\mathcal{A}$, it will send $(x_1, \ldots, x_n)$ to $\mathcal{A}$. It then decides randomly if the cheating was detected or not:

- **Undetected:** With probability $1 - \varepsilon$, $\mathcal{F}_{\text{ECF}}^{f}$ sends $(\texttt{undetected})$ to the adversary. Then $\mathcal{A}$ specifies for each $i \in H$ an output y_i and $\mathcal{F}_{\text{ECF}}^{f}$ outputs y_i to P_i for $i \in H$.
- **Detected:** With probability ε, $\mathcal{F}_{\text{ECF}}^{f}$ sends $(\texttt{detected})$ to $\mathcal{A}$. In this case $\mathcal{A}$ also gets to decide the output y_i for $i \in H$, but must ensure that $\cap_{i \in H} J_i \neq \emptyset$ at the end of the execution.

Output generation: If $\mathcal{A}$ did not attempt to cheat, $\mathcal{F}_{\text{ECF}}^{f}$ computes outputs $(y_1, \ldots, y_n) = f(x_1, \ldots, x_n)$ and gives y_i to P_i.

Fig. 1. Functionality $\mathcal{F}_{\text{ECF}}^{f}$

The functionality $\mathcal{F}_{\text{SECF}}^{f}$ is defined exactly as $\mathcal{F}_{\text{ECF}}^{f}$, except that when the adversary sends a cheat message, the functionality does not send the inputs of honest parties to the adversary. This only happens if the cheating is undetected. We can now define security:

Definition 1. *Protocol π computes f with ε-ECF (SECF) security and threshold t if it implements $\mathcal{F}_{\text{ECF}}^{f}$ ($\mathcal{F}_{\text{SECF}}^{f}$) in the UC model, securely against poly-time adversaries corrupting at most t parties.*

This definition naturally extends to a hybrid UC model where certain functionalities are assumed to be available. By the UC composition theorem and given implementations of the auxiliary functionalities, a protocol follows that satisfy the above definition without auxiliary functionalities.

In the following, we will consider secure evaluation of an arithmetic circuit $\mathcal{C}$ over some finite field K. We assume that each input and output of $\mathcal{C}$ is assigned

to some party, whence $\mathcal{C}$ induces in a natural way a function $f_{\mathcal{C}}$ of the form considered above. In the following, "computing $\mathcal{C}$ securely" will mean computing $f_{\mathcal{C}}$ securely in the sense of the above definition.

We will denote the participants in the protocol by $P_1, \ldots, P_n$ for a total of n parties. Shamir secret sharing of $a \in K$ with threshold t results in a set of shares denoted by $[a]_t$ or simply $[a]$ when the threshold is clear from the context. The share held by P_i is denoted a_i.

3 Auxiliary Functionalities

We will employ some ideal functionalities in order to make the presentation more clear. They represent sub-protocols which we show how to implement in Section 5.

Message Transmission Functionality We will make use of a functionality $\mathcal{F}_{\text{TRANSMIT}}$ that is an enhancement of the standard model for secure point-to-point channels. It essentially allows to prove to third parties which messages one received during the protocol, and to further transfer such revealed messages. It does not commit the corrupted parties to what they sent to each other. See Fig. 2 for details.

The ideal functionality $\mathcal{F}_{\text{TRANSMIT}}$ works with message identifiers mid encoding a sender $s(mid) \in \{1, \ldots, n\}$ and a receiver $r(mid) \in \{1, \ldots, n\}$. We assume that no mid is used twice. The functionality works as follows:

Secure transmit: When receiving $(\texttt{transmit}, mid, m)$ from $P_{s(mid)}$ and receiving $(\texttt{transmit}, mid)$ from all (other) honest parties, store (mid, m), mark it as *undelivered*, and output $(mid, |m|)$ to the adversary. If P_s does not input a $(\texttt{transmit}, mid, m)$ message, then output $(\texttt{corrupt}, s(mid))$ to all parties.

Synchronous delivery: At the end of each round, deliver each undelivered (mid, m) to $P_{r(mid)}$ and mark (mid, m) as delivered.

Reveal received message: On input $(\texttt{reveal}, mid, i)$ from a party P_j which at any point received the output (mid, m), output (mid, m) to P_i.

Do not commit corrupt to corrupt: If both P_j and P_s are corrupt, then the adversary can ask $\mathcal{F}_{\text{TRANSMIT}}$ to output (mid, m') to any honest P_i for any m' and any mid with $s(mid) = s$.

Fig. 2. Ideal Functionality $\mathcal{F}_{\text{TRANSMIT}}$

This functionality will be used for all private communication in the following, and provides a way to reliably show what was received at any earlier point in the protocol. This is used when the dummy execution is checked for consistency.

Input Functionality For notational convenience we assume that each P_i has one input $x_i \in K$. The input functionality runs as in Fig. 3. Note that we let the adversary pick the dummy inputs, which is done simply not to decide at this abstract level on any specific set of dummy inputs, like $x^{(i,d)} = 0$.

The ideal functionality $\mathcal{F}_{\text{INPUT}}$ is parametrized by a secret sharing scheme, sss, and works as follows.

1. Receive an input x_i from each P_i and an input $(d_1, \ldots, d_n)$ from the adversary. The adversary also inputs x_i for $i \in C$.
2. Flip a uniformly random bit $d \in_{\mathsf{R}} \{0, 1\}$.
3. Let $e = 1 - d$. Let $x^{(i,d)} = d_i$ be the *dummy* inputs and let $x^{(i,e)} = x_i$ be the *enriched* inputs.
4. For $j = 1, \ldots, n$ and $c = 0, 1$, sample $[x^{(j,c)}] \leftarrow \text{sss}(x^{(j,c)})$.
5. Output $(x_i^{(j,0)})_{j=1}^n$ and $(x_i^{(j,1)})_{j=1}^n$ to P_i.
6. On a later input $(\texttt{reveal}, i, k)$, output d and $(x_i^{(j,d)})_{j=1}^n$ to P_k.

Fig. 3. Ideal Functionality $\mathcal{F}_{\text{INPUT}}$

Commitment Functionality We use a flavor of commitment where the committer cannot avoid that a commitment is revealed. The details are given in Fig. 4.

The functionality $\mathcal{F}_{\text{COMMIT}}$ uses commitment identifiers encoding the sender $s(cid)$ of the commitment. It works as follows:

Commit: On input $(\texttt{commit}, cid, m)$ from $P_{s(cid)}$ and input $(\texttt{commit}, cid)$ from all (other) honest parties, store (cid, m) and output $(\texttt{commit}, cid, |m|)$ to the adversary.

Reveal: On input $(\texttt{reveal}, cid, r)$ from all honest parties, where (cid, m) is stored, give (cid, m) to P_r.

Fig. 4. Ideal Functionality $\mathcal{F}_{\text{COMMIT}}$

Coin-Flip Functionality We use the coin-flip functionality given in Fig. 5.

The functionality $\mathcal{F}_{\text{FLIP}}^B$ is parametrized by a positive integer B and works as follows:

1. Sample a uniformly random $k \in_{\mathsf{R}} \{0, \ldots, B-1\}$.
2. When the first honest party inputs $(\texttt{flip})$, output k to the adversary.
3. If in the round where the first honest party inputs $(\texttt{flip})$ there is some party P_i which does not input $(\texttt{flip})$, then output $(\texttt{corrupt}, i)$ to all parties.

Fig. 5. Ideal Functionality $\mathcal{F}_{\text{FLIP}}^B$

4 Protocol

Having defined the necessary ideal functionalities, we will now describe how we use them to compile the classical passively secure protocol by Ben-Or et al. [6] based on Shamir secret-sharing into one with covert security. This protocol computes an arithmetic circuit $\mathcal{C}$ with passive security. Assuming the inputs to the arithmetic circuit have been secret shared, the protocol does addition by having parties add their shares locally, and multiplication by local multiplication of shares followed by a re-sharing by each parties of the local products. Due to space constraints, we assume the details are known to the reader.

The protocols in this section use the auxiliary functionalities we defined. Thus the actual complexity of our construction depends on the implementation of those auxiliary functionalities. It turns out that the overhead incurred includes a contribution coming from the cryptographic primitives we use, this overhead does not depend on the communication complexity of protocol we compile. In addition, the adversary can choose to slow down $F_{Transmit}$ by a factor of n, but since he cannot make it fail, a covert adversary is unlikely to make such a choice as discussed in the introduction.

We begin with a simple construction which has a rather poor computational complexity. Following that, we show how the simple protocol can be adapted to yield a better complexity.

Theorem 1. *The protocol in Fig. 6 computes $\mathcal{C}$ with $\frac{1}{2}$-SECF security and threshold $t < n/2$ in the $(\mathcal{F}_{\text{TRANSMIT}}, \mathcal{F}_{\text{INPUT}}, \mathcal{F}_{\text{COMMIT}}, \mathcal{F}_{\text{FLIP}})$-hybrid world against a static adversary.*

Proof. Initially $\mathcal{S}$ is given the inputs of the corrupt parties. It passes them on to $\mathcal{A}$ and simulates the protocol execution up until the point where the bit d is revealed and it is determined which of the two executions were the dummy execution. $\mathcal{S}$ does this by inventing random shares whenever $\mathcal{A}$ would expect to see a share from an honest party. $\mathcal{A}$ will always see only t shares and any subset of size t look completely random in the real protocol execution. $\mathcal{S}$ can therefore simulate them perfectly by giving $\mathcal{A}$ random values.

During the protocol, $\mathcal{A}$ is observed by $\mathcal{S}$ and it can thus be determined if $\mathcal{A}$ ever sends an incorrect intermediate result to one of the honest parties.

- If $\mathcal{A}$ did not cheat at all, or if $\mathcal{A}$ cheated in both executions, then $\mathcal{S}$ simply follows the protocol. In the first case $\mathcal{F}_{\text{SECF}}^{fc}$ will give $\mathcal{S}$ the outputs for the corrupt parties, which $\mathcal{S}$ can pass along to $\mathcal{A}$ unchanged. In the second case, $\mathcal{A}$ will be caught with certainty before seeing anything which depend on the honest parties inputs. $\mathcal{S}$ can therefore simulate the protocol execution towards $\mathcal{A}$ using random shares only.
- If $\mathcal{A}$ cheats in execution d' (first or second execution), $\mathcal{S}$ will send (`cheat`) to $\mathcal{F}$. The functionality then determines if the cheat was successful:

  Detected: The simulator must now ensure that $\mathcal{A}$ believes he cheated in the dummy execution.

  $\mathcal{A}$ will want to query $\mathcal{F}_{\text{INPUT}}$ for the value of d and the shares of the dummy inputs. In response, $\mathcal{S}$ sends a response with $d = d'$, which means that $\mathcal{A}$ cheated in the dummy execution. $\mathcal{S}$ must also send back shares of the inputs $\{x^{(j,d)} = d_j\}_{j=1}^n$ consistent with the shares $\mathcal{A}$ has already seen. At this point $\mathcal{A}$ has only seen random values for an *non-qualified* subset of the parties in response to its call to $\mathcal{F}_{\text{INPUT}}$. $\mathcal{S}$ can therefore choose polynomials that agree with these values and at the same time correspond to a sharing of the inputs d_j, and compute consistent shares of the honest parties using these polynomials.

  If P_j were the first corrupt party who send an incorrect message to an honest party, $\mathcal{S}$ will send $(\texttt{corrupt}, j)$ to $\mathcal{F}_{\text{SECF}}^{fc}$.

In general, if any of the ideal functionalities output $(\texttt{corrupt}, j)$ to P_i, then P_i also outputs $(\texttt{corrupt}, j)$. Not mentioning this further, the protocol proceeds in five steps:

1. All parties provide input to $\mathcal{F}_{\text{INPUT}}$. In return they obtain shares of secret sharings $[x^{(j,0)}]$ and $[x^{(j,1)}]$ for $j = 1, \ldots, n$. Nobody knows which sharings are dummy at this point.
2. Each party P_i generates random keys K_i^0 and K_i^1 and commit to them using $\mathcal{F}_{\text{COMMIT}}$ twice.
3. The passively secure protocol is run on both input sets $\{[x^{(j,0)}]\}_{j=1}^n$ and $\{[x^{(j,1)}]\}_{j=1}^n$. This evaluates the circuit $\mathcal{C}$ twice. The parties use $\mathcal{F}_{\text{COMMIT}}$ to commit to their shares of the output. All randomness used in the first and second protocol run come from pseudo-random generators seeded by K_i^0 and K_i^1, respectively.
4. The parties query $\mathcal{F}_{\text{INPUT}}$ for the random bit d and the shares of $\{[x^{(j,d)}]\}_{j=1}^n$. They then use $\mathcal{F}_{\text{COMMIT}}$ to reveal the key K_i^d used for the pseudo-random generator for all P_i. Knowing the initial inputs and the seed for the pseudo-random generator used, the entire message trace of all parties is fixed. The parties also open the commitments to the dummy output shares.
5. Each party locally simulates the entire dummy execution to determine if any cheating took place. This amounts to checking for each party whether his input shares of $[x^{(j,d)}]$ (revealed by $\mathcal{F}_{\text{INPUT}}$) and seed K_i^d (revealed by $\mathcal{F}_{\text{COMMIT}}$) together lead to the shares he claims to have obtained of the output (revealed by $\mathcal{F}_{\text{COMMIT}}$) if he follows the passively secure protocol on the messages that other parties would have sent if they followed the protocol on their shares and expanded randomness. If no discrepancies are found, the output shares of the real execution are opened.

   Otherwise, the honest parties must determine who cheated [a].

   The parties have already locally simulated the dummy execution so they know the correct message trace. It is therefore simple to match this against the actual message trace revealed by $\mathcal{F}_{\text{TRANSMIT}}$ and pinpoint the first deviation. If P_j made the first mistake, the honest parties output $(\texttt{corrupt}, j)$ and halt.

[a] Note that it is possible for a corrupt party to "frame" an honest party by sending him wrong intermediate results. The honest party cannot tell the difference and will produce incorrect output. $\mathcal{F}_{\text{TRANSMIT}}$ is there to safeguard honest parties against this form of attack. The parties call it to reveal all messages that were received in the dummy execution.

Fig. 6. Simple version

Undetected: In this case the functionality responded with $(\texttt{undetected})$ together with the honest parties' inputs. The simulator must therefore make it look as if $\mathcal{A}$ cheated in the execution that was not opened, i.e., the real execution. As above, $\mathcal{S}$ can compute polynomials that will give a correct sharing of inputs based on what $\mathcal{A}$ already know and with $d = 1 - d'$.

Using these inputs together with the corrupt parties' inputs and outputs, $\mathcal{S}$ can now compute the consequence of $\mathcal{A}$'s cheating, i.e., the altered outputs of the honest parties. It passes these outputs to $\mathcal{F}_{\text{SECF}}^{fc}$ as the honest parties' outputs.

It is clear that the above simulation matches the output of $\mathcal{A}$ in the hybrid world perfectly when $\mathcal{A}$ did not cheat and when $\mathcal{A}$ was foolish enough to cheat in both executions.

When $\mathcal{A}$ cheats in just one execution, $\mathcal{S}$ will make the honest parties output $(\texttt{corrupt}, j)$ for some corrupt P_j (if $\mathcal{A}$ was detected) or output normal outputs (if $\mathcal{A}$ was undetected). Each of these two cases are picked with probability exactly $\frac{1}{2}$ by the random choice made by $\mathcal{F}_{\text{SECF}}^{fc}$. We get the same probability distribution in the hybrid world where $\mathcal{F}_{\text{INPUT}}$ picks the bit d uniformly at random.

In total, we can now conclude that the protocol in Fig. 6 computes $f_{\mathcal{C}}$ with $\frac{1}{2}$-SECF security.

The above protocol has each party execute the passively secure protocol twice after which each party simulates the actions of all other parties in the dummy execution. In the standard BGW protocol [6], each party has a computational complexity of $\mathcal{O}(n)$ per gate. By asking every party to simulate every other party, we increase the computational complexity to $\mathcal{O}(n^2)$ per gate.

The communication complexity is doubled by running the passively secure protocol twice. In the normal case where the dummy execution is found to contain no errors, the communication complexity is increased no further. When errors *are* detected, every party is sent the messages communicated by every other party. This will again introduce a quadratic blowup, now in the communication complexity. We argued in the introduction that even a small fixed chance of catching misbehavior is enough to deter the parties. Because of that, we expect to find no discrepancies most of the time, and thus obtain the same *communication* complexity as the original protocol within a constant factor. We still have a quadratic blowup in the *computational* complexity. However, local computations are normally considered free compared to the communication, i.e., the network is expected to be the bottleneck. So for a moderate number of parties, this simple protocol can still be quite efficient.

Still, we would like to lower the complexity when errors are detected. Below we propose a slightly more complex protocol which has only a constant overhead in both computation and communication both when no errors are detected and when the parties are forced to do a more careful verification.

If no errors are detected, each party does two protocol executions followed by a check of the input/output behavior of one other party. This is clearly a constant factor overhead compared to the passively secure protocol. When a party is accused, all other parties must check this party. This adds only a linear overhead to the overall protocol, and thus the protocol in Fig. 7 has a linear overhead compared to the passively secure protocol.

It might seem as an overkill in the protocol in Fig. 7 to use $\mathcal{F}_{\text{TRANSMIT}}$ for communication and then also have the parties commit to their communication using $\mathcal{F}_{\text{COMMIT}}$. The reason for the commitments is to commit the corrupted parties to what they sent among each other before it is revealed which parties check which parties. If we do not do that, they might decide on which of them was the deviator after the revelation of d and k and thus always pick the deviator

This is a modification of the protocol in Fig. 6. After running Step 1–3 unchanged, it continues with:

1. All P_i use $\mathcal{F}_{\text{COMMIT}}$ to commit to their view of the protocol, i.e., all messages exchanged between P_i and P_j for all j. This results in commitments $\text{comm}_{\{i,j\}}^{(i,0)}$ for the first execution and $\text{comm}_{\{i,j\}}^{(i,1)}$ for the second, where $\text{comm}_{\{i,j\}}^{(m,c)}$ is the view of P_m of what was sent between P_i and P_j in execution number c.
2. The parties query $\mathcal{F}_{\text{INPUT}}$ for the random bit d and the shares of the dummy inputs. They then use $\mathcal{F}_{\text{FLIP}}^{n-1}$ to flip a uniformly random $k \in \{1, \ldots, n-1\}$ that will be used when checking. $\mathcal{F}_{\text{COMMIT}}$ is used by all parties to reveal the key K_i^d used for the pseudo-random generator for all P_i. Finally, the commitments to shares in the output from the dummy execution are opened.
3. Each party P_i checks P_l, where $l = (i - 1 + k \bmod n) + 1$, i.e., he checks P_{i+k} with wraparound from P_n back to P_1.
   The commitments $\text{comm}_{\{l,j\}}^{(j,d)}$ and $\text{comm}_{\{l,j\}}^{(l,d)}$ are opened to P_i, i.e., the committed views of P_l and P_j of what was exchanged between them. If there is a disagreement, then P_i broadcasts a complaint and P_l and P_j must decommit to all parties and use $\mathcal{F}_{\text{TRANSMIT}}$ to show which messages they received from the other. This will clearly detect at least one corrupt party among P_l and P_j if P_i was honest, or reveal P_i as corrupt if the commitments were equal after all, i.e., if P_i made a false accusation.
   If all committed views agree, then P_i simulates the local computations done by P_l and checks whether this leads to the shares of the dummy output opened by P_l and the messages sent according to $\text{comm}_{\{l,j\}}^{(l,d)}$. If a deviation is found, P_i broadcasts an accusation against P_l, and all parties check P_l as P_i did. If they verify the deviation they output $(\texttt{corrupt}, l)$, otherwise they output $(\texttt{corrupt}, i)$.
4. If no accusations were made, the output of the real execution is opened.

Fig. 7. Efficient version

to be one which is checked by a corrupted party. For an example of what can go wrong without the commitments the interested reader can refer to Appendix B.

Theorem 2. *The protocol in Fig. 7 computes $\mathcal{C}$ with $\frac{1}{4}$-SECF security and threshold $t < n/2$ in the $(\mathcal{F}_{\text{TRANSMIT}}, \mathcal{F}_{\text{INPUT}}, \mathcal{F}_{\text{COMMIT}}, \mathcal{F}_{\text{FLIP}})$-hybrid world.*

Proof. The simulator for the protocol in Fig. 7 runs like the simulator for the protocol in Fig. 6, except that it must now only output $(\texttt{corrupt}, i)$ to $\mathcal{F}$ if it determines that a message trace for a corrupt party P_i was checked by an honest party, and it must do while maintaining the same probability distribution as in the hybrid world.

As before, $\mathcal{S}$ will simulate $\mathcal{A}$ and observe the messages sent to honest parties. As soon as an incorrect message is observed in execution d' and all parties committed to their communication with the other parties, we know there exists an offset $k' \in \{1, \ldots, n-1\}$ for which an honest P_i would catch a corrupt P_l, where $l = (i - 1 + k' \bmod n) + 1$ in execution d':

- If two parties P_l and P_j committed to $\text{comm}_{\{l,j\}}^{(l,d')} \neq \text{comm}_{\{l,j\}}^{(j,d')}$, then one of them is corrupted, P_l say, and we pick k' such that P_l is checked by an honest P_i.

– If $\text{comm}_{\{l,j\}}^{(l,d')} = \text{comm}_{\{l,j\}}^{(j,d')}$ for all pairs of parties, then the wrong message sent to an honest party in execution d' implies that some party P_l is committed to values which are not consistent with an execution of the protocol, and we pick k' to ensure that P_l is checked by an honest party.[2]

The simulator sends (`cheat`) to $\mathcal{F}_{\text{SECF}}^{fc}$. We have two outcomes:

Detected: Set $d = d'$ and sample k at random such that P_l is checked by an honest party.

Undetected: Set $d = d'$ with probability $\frac{1}{3}$, and $d = 1 - d'$ otherwise. Sample $k \in \{1, \ldots, n-1\}$ such that P_l is checked by an honest party with probability $\alpha = \frac{4}{3}(\frac{n-t}{n-1} - \frac{1}{4})$.

If $\mathcal{A}$ did not cheat, $\mathcal{S}$ selects d and k as in the hybrid protocol. The simulation continues as in the hybrid world with these choices for d and k.

If $\mathcal{A}$ did not cheat, the ideal world output clearly match the hybrid world. When $\mathcal{A}$ did cheat, we will show that d and k are picked with the correct distribution. First note that $\mathcal{S}$ pick $d = d'$ with probability $\frac{1}{4} \cdot 1 + \frac{3}{4} \cdot \frac{1}{3} = \frac{1}{2}$, as in the hybrid world.

For the selection of k, note that a cheating party will always have a unique distance to every honest party. These distances make up a subset of $\{1, \ldots, n-1\}$ of size $n - t$. The cheater is caught exactly when the offset is picked within this subset. This happens with probability $\frac{n-t}{n-1}$ in the hybrid world. The simulator picks k among the indices of honest parties with the same probability: $\frac{1}{4} + \frac{3}{4}\alpha = \frac{n-t}{n-1}$. We conclude that $\mathcal{S}$ will simulate the hybrid world.

5 Implementation of Sub-Protocols

In this section we sketch how to implement the sub-protocols described above.

Detection In all sub-protocols we will need a tool for stopping the protocol "gracefully" when corruption is detected This is done by all parties running the following rules in parallel.

1. If a party P_i sees that a party P_d deviates from the protocol, then P_i signs $(\texttt{corrupt}, d)$ to get signature γ_i and sends the signature to all parties. Then P_i outputs $(\texttt{corrupt}, d)$.
2. If P_k received a signature γ_i on $(\texttt{corrupt}, d)$ from $t + 1$ distinct parties P_i, it considers these as a *proof* that P_d is corrupted, sends this proof to all parties, outputs $(\texttt{corrupt}, d)$, waits for one round and then terminates all protocols.

[2] Note that P_l need not be the one who sent the incorrect message to the honest party—P_l may have behaved locally consistent given its inputs—but $\mathcal{S}$ will be able to find a first deviator, and it will clearly not be one of the honest parties.

3. If P_k receives a proof that P_d is corrupt from any party, it relays this proof to all parties, outputs $(\texttt{corrupt}, d)$, waits for one round and then terminates all protocols.

If the signature scheme are unforgeable and only corrupted parties deviate from the protocol, then the protocol has the following two properties, except with negligible probability.

Detection soundness: If an honest party outputs $(\texttt{corrupt}, d)$, then P_d is corrupt.

Common detection: If an honest party terminates the protocol prematurely, then there exists P_d such that *all* honest parties have output $(\texttt{corrupt}, d)$.

The reason why the relayer P_r waits for one round before terminating is that P_r wants all other parties to have seen a proof that P_i is corrupt before it terminates itself. Otherwise the termination of P_r would be considered a deviation and an honest P_r could be falsely detected. In the following we do not always mention explicitly that the detection sub-protocol is run as part of all protocols.

Transmission Functionality The transmission protocol can run in two modes. In *cheap mode* $\mathcal{F}_{\text{TRANSMIT}}$ is implemented as follows.

1. On input $(\texttt{transmit}, mid, m)$ party $P_{s(mid)}$ signs (mid, m) to obtain signature σ_s and sends (mid, m, σ_s) to $P_{r(mid)}$.
2. On input $(\texttt{transmit}, mid)$ party $P_{r(mid)}$ waits for one round and then expects a message (mid, m, σ_s) from $P_{s(mid)}$, where σ_s is a valid signature from P_s on (mid, m). If it receives it, it outputs (mid, m).
3. On input $(\texttt{reveal}, mid, i)$ party P_j, if it at some point output (mid, m), sends (mid, m, σ_s) to P_i, which outputs (mid, m) if σ_s is valid.

It is easy to check that this is a UC secure implementation under the following restrictions:

Synchronized input from honest parties: If some honest party receives input $(\texttt{transmit}, mid)$, then all honest parties $P_i \neq P_{s(mid)}$ receives the same input $(\texttt{transmit}, mid)$. Furthermore, if $P_{s(mid)}$ is honest, it receives input $(\texttt{transmit}, mid, m)$ for some m.

Signatures: Even corrupted P_s send along the signatures σ_s.

The restriction *synchronized input from honest* can be enforced by the way the ideal functionality is used by an outer protocol, i.e., by ensuring that the honest parties agree on which message identifiers are used for which message in which rounds. This is the case for the way we use $\mathcal{F}_{\text{TRANSMIT}}$. The restriction *signatures* is unreasonable, and we show how to get rid of it below. We need the rule **Do not commit corrupt to corrupt** in $\mathcal{F}_{\text{TRANSMIT}}$ as we cannot prevent a corrupt P_s from providing a corrupt P_i with signatures on arbitrary messages, i.e., we cannot commit the corrupted parties to what they have sent among themselves.

As mentioned, the above implementation only works if all senders honestly send the needed signatures. If at some point some P_r does not receive a valid signature from P_s, it publicly accuses P_s of being corrupted and the parties switch to the below *expensive mode* for transmissions from P_s to P_r.

1. On input $(\texttt{transmit}, mid, m)$ party $P_{s(mid)}$ signs (mid, m) to obtain signature σ_s and sends (mid, m, σ_s) to all $P_i \neq P_s$.
2. On input $(\texttt{transmit}, mid)$ parties $P_i \neq P_{s(mid)}$ wait for one round and then expects a message (mid, m, σ_s) from $P_{r(mid)}$, where σ_s is a valid signature of $P_{s(mid)}$ on (mid, m). If P_i receives it, it sends (mid, m, σ_s) to $P_{r(mid)}$. Otherwise, it sends a signature γ_i on $(\texttt{corrupt}, i)$ to all parties.
3. On input $(\texttt{transmit}, mid)$ party $P_{r(mid)}$ waits for two rounds and then expects a message (mid, m, σ_s) from each P_i, where σ_s is a valid signature of $P_{s(mid)}$ on (mid, m). If it arrives from some P_i, then P_r outputs (mid, m).

Note that now each round of communication on $\mathcal{F}_{\text{TRANSMIT}}$ takes two rounds on the underlying network. Between two parties where there have been no accusations, messages are sent as before (Step 1 in the above protocol) and the extra round is used for silence—it is necessary that also non-accusing parties use two rounds to not lose synchronization.

If P_s sends a valid signature to just one honest party, then P_r gets its signature and can proceed as in optimistic mode. If P_s does not send a valid signature to any honest party, then all $n - t$ honest P_i send γ_i to all parties and hence all honest parties output $(\texttt{corrupt}, s)$ in the following round, meaning that P_s was detected. Using these observations it can easily be shown that the above protocol is a UC implementation of $\mathcal{F}_{\text{TRANSMIT}}$ against covert adversaries with deterrence 1. Note that it is not a problem that we send m in cleartext through all parties, as an accusation of P_s by P_r means that P_s or P_r is corrupt, and hence m need not be kept secret.

We skipped the details of how the accusations are handled. We could in principle handle accusations by using one round of broadcast after each round of communication to check if any party wants to make an accusation. After broadcasting the accusations, the appropriate parties can then switch to expensive model. To avoid using a Byzantine agreement primitive in each round, we use a slightly more involved, but much cheaper technique which communicates less than n^2 bits in each round and which only uses a BA primitive when there are actually some accusations to be dealt with. The details are given in the next section.

In cheap mode, using $\mathcal{F}_{\text{TRANSMIT}}$ adds an overhead $N\kappa$ bits compared to plain transmission, where κ is the length of a signature and N is the number of messages sent. In expensive mode this overhead is a factor n larger.

Cheap Exception Handling Consider a protocol consisting of two protocols π_{MAIN} and π_{EXCEPT}, both for the authenticated, synchronous point-to-point model. Initially the parties run π_{MAIN}. The goal is to allow any party to raise a flag, which stops π_{MAIN} and starts π_{EXCEPT}. With some details left out for now, this is handled as follows.

– If a party P_i wants to stop the main protocol, it sends (`stop`) to all parties and stops the execution of π_{MAIN}. It records the round R_i in which it stopped running π_{MAIN}.
– If a party P_i receives (`stop`) from any party while running π_{MAIN}, it sends (`stop`) to all parties and stops the execution of π_{MAIN}. It records the round R_i in which it stopped running π_{MAIN}.
– After all parties stopped they resynchronize and then run π_{EXCEPT}.
– After having run π_{EXCEPT}, the parties agree on a round C of π_{MAIN} which was executed completely, i.e., $R_i > C$ for all honest P_i, and then they rerun from round $C+1$. If a party P_r already received a message from P_s for one of the rounds that are now rerun, then P_r ignores any new message sent by P_s for that round. This is to avoid that corrupted parties can change their mind on what they sent in a previous round.

The resynchronization is needed as honest parties might stop in different rounds—though at most with a staggering of one round.

The resynchronization uses a sub-protocol where the input of P_i is the round R_i in which it stopped. The output is some common R such that it is guaranteed that $R_i = R$ for some honest P_i, i.e., at least one honest party stopped in round R. Since the honest parties stop within one round of each other, it follows that all honest parties stopped in round $R-1$, R or $R+1$. In particular, no honest party stopped in round $R-2$. The parties can therefore safely set $C = R-2$, i.e., rerun from round $R-1$.

The protocol used to agree on the round R proceeds as follows:

1. Each P_i has input $R_i \in \mathbb{N}$ and it is guaranteed that $|R_i - R_j| \leq 1$ for all honest P_i and P_j.
2. Let $r_i = R_i \bmod 4$ and make 4 calls to the BA functionality—name the calls BA_0, BA_1, BA_2 and BA_3. The input to BA_c is 1 if $c = r_i$ or $c = r_i - 1 \bmod 4$ and the input to BA_c is 0 if $c = r_i + 1 \bmod 4$ or $c = r_i + 2 \bmod 4$.
3. Let $o_c \in \{0,1\}$ for $c = 0,1,2,3$ denote the outcome of BA_c. Now P_i finds the largest $R \in \{R_i - 1, R_i, R_i + 1\}$ for which $o_{R \bmod 4} = 1$ and outputs R.

It is fairly straight forward to see that the honest parties output the same R and that R was always the input of some honest party. Look at two cases.

– If there exists ρ such that $R_i = \rho$ for all honest P_i, then all honest parties input the same to the BA functionalities, and then trivially $o_{\rho-1 \bmod 4} = 1$, $o_{\rho \bmod 4} = 1$, $o_{\rho+1 \bmod 4} = 0$ and $o_{\rho+2 \bmod 4} = 0$. Consequently, at honest parties outputs $R = \rho$.
– If there exists ρ such that $R_i = \rho$ for some honest P_i and $R_j = \rho+1$ for some honest P_j, then $R_k \in \{\rho, \rho+1\}$ for all honest P_k, and thus all honest P_k input 1 to $BA_{\rho \bmod 4}$, and so $o_{\rho \bmod 4} = 1$. Furthermore, all honest parties input 0 to $BA_{\rho+2 \bmod 4}$, so $o_{\rho+2 \bmod 4} = 0$. It follows that all honest parties output $R = \rho$ if $o_{\rho+1 \bmod 4} = 0$ and that all honest parties output $R = \rho + 1$ if $o_{\rho+1 \bmod 4} = 1$. Both outputs are valid.

The above protocol is an improved version of a protocol by Bar-Noy et al. [3], which in turn uses techniques from Berman et al. [7]). The protocol in [3] uses $\log(B)$ calls to the BA functionality, where B is an upper bound on the input of the parties. We use just 4.

Note that at the point where the four BAs are run, the honest parties might still be desynchronized by one round. We handle this using a technique from [15] which simulates each round in the BA protocols by three synchronous rounds in the authenticated channel model.

Commitment Functionality The protocol uses a one-round UC commitment scheme with a constant overhead (commit to κ bits using $\mathcal{O}(\kappa)$ bits), which can be realized with static security in the PKI model [4] given any mixed commitment scheme [11] with a constant overhead. Concretely we can instantiate such a scheme under Paillier's DCR assumption. Note that opposed to Barak et al. [4] we do not need a setup assumption: We assume honest majority and can thus, once and for all, use an active secure MPC to generate the needed setup [13]. The protocol also uses an error-correcting code (ECC) for n parties which allows to compute the message from any $n - t$ correct shares.

If one is willing to use the random oracle model, UC commitment can instead be done by calling the oracle on input the message to commit to, followed by some randomness. In practice, this translates to a very efficient solution based on a hash function.

The protocol proceeds as follows.

1. On input $(\texttt{commit}, cid, m)$, $P_{s(cid)}$ computes an ECC $(m_1, \ldots, m_n)$ of m. The sender then computes $c_i \leftarrow \text{commit}_{pk_i}(m_i)$ and sends c_i to P_i via $\mathcal{F}_{\text{TRANSMIT}}$.
2. On input $(\texttt{reveal}, cid, r)$, P_i opens each c_i to P_i. The opening is sent via $\mathcal{F}_{\text{TRANSMIT}}$. If any P_i receives an invalid opening, it transfers c_i and m_i to all parties and P_s is detected as a cheater. Otherwise, P_i transfers c_i and the opening to P_r.
3. Then P_r collects validly opened c_i. Let I be the index of these and let m_i be the opening of c_i for $i \in C$. If $|I| < n - t$, then P_r waits for one round and terminates.[3] If $(m_i)_{i \in I}$ is not consistent with a codeword in the ECC, then P_r transfers $(c_i)_{i \in I}$ and the valid openings to the other parties which detect P_s as corrupted. Otherwise, P_r uses $(m_i)_{i \in I}$ to determine m and outputs (cid, m).

Assuming that a commitment to ℓ bits have bit-length $\mathcal{O}(\max(\kappa, \ell))$, where κ is the security parameter, the complexity of a commitment to ℓ bits followed by an opening is $\mathcal{O}(n \max(\kappa, \ell/n)) = \mathcal{O}(n(\kappa + \ell/n)) = \mathcal{O}(\ell + n\kappa)$. This is assuming that there are no active corruptions, such that $\mathcal{F}_{\text{TRANSMIT}}$ has constant overhead.

[3] Since we assume that at most t parties are corrupted, we can assume that either P_s is detected or P_r receives $n - t$ commitments with corresponding valid decommitments.

Flip Functionality To implement $\mathcal{F}_{\text{FLIP}}^{B}$ the parties proceed as follows.

1. On input (`flip`), all P_i commit to a uniformly random $k_i \in [0, B-1]$.
2. In the next round all P_i reveal k_i to all parties.
3. All parties output $k = \sum_{i=1}^{n} k_i \bmod B$.

Under the condition that the protocol is used by the honest parties in a way that guarantees that they input (`flip`) in the same round, the argument that the protocol implements the functionality against a covert adversary (with deterrence 1) is straight forward.

Input Functionality The input functionality can be implemented using a VSS with a multiplication protocol active secure against $t < n/2$ corruptions. The VSS should have the property that it is possible to verifiable reconstruct the secret and the share of all parties given the shares of the honest parties—standard bivariate sharing has this property. We sketch the protocol.

1. Each P_i deals a VSS $[\![x_i]\!]$ of its input x_i.
2. The parties use standard techniques to compute a VSS $[\![d]\!]$ of a uniformly random $d \in_{\mathsf{R}} \{0,1\} \subset K$.
3. For each input $[\![x_i]\!]$ the parties use an actively secure multiplication protocol to compute $[\![x^{(i,0)}]\!] = [\![d_i \cdot x_i]\!]$ and $[\![x^{(i,1)}]\!] = [\![(1-d_i) \cdot x_i]\!]$.
   Each P_i takes its output to be $(x_i^{(j,0)})_{j=1}^{n}$ and $(x_i^{(j,1)})_{j=1}^{n}$, where $x_i^{(j,c)}$ is its point on the polynomial used by the sharing $[\![x^{(j,c)}]\!]$. The other values of the VSS are internal to the implementation of $\mathcal{F}_{\text{INPUT}}$ and only used for the below command.
4. On input $(\texttt{reveal}, i, k)$ the parties reconstruct $[\![d]\!]$ and all $[\![x^{i,d}]\!]$ towards P_k and P_k computes the points $x^{(j,d)}$ of P_j in all sharings and output $(x_i^{(j,d)})_{j=1}^{n}$.

Bibliography

[1] J. Algesheimer, J. Camenisch, and V. Shoup. Efficient computation modulo a shared secret with application to the generation of shared safe-prime products. In *CRYPTO*, pages 417–432, 2002.

[2] Y. Aumann and Y. Lindell. Security against covert adversaries: Efficient protocols for realistic adversaries. In S. P. Vadhan, editor, *TCC*, volume 4392 of *Lecture Notes in Computer Science*, pages 137–156. Springer, 2007.

[3] A. Bar-Noy, X. Deng, J. A. Garay, and T. Kameda. Optimal amortized distributed consensus. *Information and Computation*, 120(1):93–100, 1995.

[4] B. Barak, R. Canetti, J. B. Nielsen, and R. Pass. Universally composable protocols with relaxed set-up assumptions. In *FOCS*, pages 186–195. IEEE Computer Society, 2004.

[5] Z. Beerliová-Trubíniová and M. Hirt. Perfectly-secure mpc with linear communication complexity. In *TCC*, pages 213–230, 2008.

[6] M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *STOC*, pages 1–10. ACM, 1988.
[7] P. Berman, J. A. Garay, and K. J. Perry. Optimal early stopping in distributed consensus. In A. Segall and S. Zaks, editors, *WDAG*, volume 647 of *Lecture Notes in Computer Science*, pages 221–237. Springer, 1992.
[8] P. Bogetoft, D. L. Christensen, I. Damgård, M. Geisler, T. Jakobsen, M. Krøigaard, J. D. Nielsen, J. B. Nielsen, K. Nielsen, J. Pagter, M. I. Schwartzbach, and T. Toft. Secure multiparty computation goes live. In *Financial Cryptography*, pages 325–343, 2009.
[9] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, pages 136–145. IEEE, 2001.
[10] I. Damgård and J. B. Nielsen. Scalable and unconditionally secure multiparty computation. In *CRYPTO*, pages 572–590, 2007.
[11] I. Damgård and J. B. Nielsen. Perfect hiding and perfect binding universally composable commitment schemes with constant expansion factor. In M. Yung, editor, *CRYPTO*, volume 2442 of *Lecture Notes in Computer Science*, pages 581–596. Springer, 2002.
[12] I. Damgård, Y. Ishai, M. Krøigaard, J. B. Nielsen, and A. Smith. Scalable multiparty computation with nearly optimal work and resilience. In *CRYPTO*, pages 241–261, 2008.
[13] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game – a completeness theorem for protocols with honest majority. In *STOC*, pages 218–229. ACM, 1987.
[14] V. Goyal, P. Mohassel, and A. Smith. Efficient two party and multi party computation against covert adversaries. In *EUROCRYPT*, pages 289–306, 2008.
[15] Y. Lindell, A. Lysyanskaya, and T. Rabin. Sequential composition of protocols without simultaneous termination. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 203–212. ACM Press, 2002.

A General Transformation

In this section we describe the general transformation. We start by giving one set of requirements which allows to do the transformation, and which allows an easy analysis. After giving the requirements we discuss some relaxations. Then we give the protocol, the simulator and the analysis of the simulator. Throughout the section we let H denote the set of indices of honest parties and C the indices of corrupted parties.

A.1 Requirements

We can transform protocols consisting of the following two parts.

Secret sharing scheme: The first part is a secret sharing scheme, sss. Given a secret s and a randomizer r it produces n shares $(s_1, \ldots, s_n) = \text{sss}(s; r)$; We use the notation $[s] = (s_1, \ldots, s_n)$ to denote n-vectors. We write $[s] \in \text{sss}$ to mean that $[s] = \text{sss}(s; r)$ for some r.

Secret shared computation: A protocol π_f which computes on shared values. In the input configuration the parties hold shares of $[x^{(j)}] \in \text{sss}$ for $j = 1, \ldots, \ell$. The protocol ends by the parties holding share of $[y^{(j)}] \in \text{sss}$ for $j = 1, \ldots, \ell$—for notational convenience we assume ℓ inputs and ℓ outputs.

We have the following requirements:

Reconstruction: We need that s can be computed efficiently from the shares of the honest parties in $[s] \in \text{sss}$.

Patching of secret sharing scheme: In words, it is possible to explain the shares of any non-qualified set as the shares of any secret.

Technically, there exists a poly-time function, patch, with the following property. Let C denote a non-qualified set. Compute $(s_1, \ldots, s_n) \leftarrow \text{sss}(s; r)$ for a uniformly random randomizer r for the secret sharing scheme and compute $r' = \text{patch}(s, r, C, s')$. Then r' is uniformly random given s' and $s'_i = s_i$ for all $i \in C$ when $(s'_1, \ldots, s'_n) \leftarrow \text{sss}(s'; r')$.

Randomness minimality: In words, the shares of honest parties are fixed by the secret and the shares of the corrupted parties.

Technically, there exists a poly-time function, complete, with the following property. Let C denote a maximally non-qualified set and let $[s] = (s_1, \ldots, s_n) \in \text{sss}$. Then $\text{complete}(s, (s_i)_{i \in C}) = (s_i)_{i \in H}$.

Correctness: If $[x^{(j)}] \in \text{sss}$ for $j = 1, \ldots, \ell$ and if $[y^{(j)}]$ for $j = 1, \ldots, \ell$ are computed as $([y^{(j)}])_{j=1,\ldots,\ell} = \pi_f(([y^{(j)}])_{j=1,\ldots,\ell})$, where all parties follow the protocol, then $[y^{(j)}] \in \text{sss}$ for $j = 1, \ldots, \ell$ and $(y^{(j)})_{j=1,\ldots,\ell} = f((x^{(j)})_{j=1,\ldots,\ell})$.

Patching of protocol: In words, it is possible to explain the view of any non-qualified set as being consistent with any inputs of the honest parties. Technically we require that there exists a poly-time function, patch, with the property defined below.

Let C be any non-qualified set and let π_i denote the code of P_i in π_f; It takes inputs $(x_i^{(j)})_{j=1}^n$ and a randomizer s_i and produces outputs $(y_i^{(j)})_{j=1}^n$.

Let $([x^{(j)}])_{j=1}^n$ and $([x^{(j)\prime}])_{j=1}^n$ be two inputs for the protocol π_f. Denote the shares of P_i in $([x^{(j)}])_{j=1}^n$ by $(x_i^{(j)})_{j=1}^n$ and denote the shares of P_i in $([x^{(j)\prime}])_{j=1}^n$ by $(x_i^{(j)\prime})_{j=1}^n$. Assume that $[x^{(j)}] \in \text{sss}$ and $[x^{(j)\prime}] \in \text{sss}$ for $j = 1, \ldots, n$ and assume that $(x_i^{(j)})_{j=1}^n = ((x_i^{(j)})')_{j=1}^n$ for $i \in C$.

For $i \in H$ sample a uniformly random randomizer s_i for π_i and for $i \in C$, let s_i be arbitrary values. Compute

$$(s'_j)_{j=1}^n = \text{patch}(([x^{(j)}])_{j=1}^n, (s_j)_{j=1}^n, C, ([x^{(j)\prime}])_{j=1}^n).$$

Execute the n parties $\pi_i((x_i^{(j)})_{j=1}^n; s_i)$ together and record the view, view_i, of each P_i. Then execute the n parties $\pi_i((x_i^{(j)\prime})_{j=1}^n; s'_i)$ together and record

the view, view'_i, of each P_i. Then the s'_i of $i \in H$ are uniformly random (and independent of s_i for $i \in C$ and the inputs $([x^{(j)'}])_{j=1}^n$) and $\text{view}_i = \text{view}'_i$ for $i \in C$, in particular $s'_i = s_i$ for $i \in C$.

The properties *correctness* and *reconstruction* are needed for the overall correctness of the protocol which runs π_f and then reconstructs. The property *patching of secret sharing scheme* ensures that the shares of the corrupted parties hold no information on s. This ensures the security of sharing the inputs using sss. The property *randomness minimality* ensures that if the shares on the honest parties are added to the shares of the corrupted parties, it leaks no information extra to s.[4] This ensures that it is secure to compute the outputs by revealing the honest shares to the corrupted parties. The property *patching of protocol* ensures that the view of the corrupted parties in an execution of π_f leaks no information whatsoever.

An example of a secret sharing scheme with the desired properties is Shamir's secret sharing scheme. An example of a protocol with the desired properties is the passive secure protocol of [13] and the passive secure protocol in [1].

Relaxations The requirements can be relaxed in several ways, which all, however, complicate the analysis sufficiently that we have chosen not to go to further generality.

First of all, we require **Randomness minimality** simply to by sure that it is secure to reconstruct the outputs by simply making the shares of the honest parties public. If the shares of the honest parties were not fixed by the shares of the corrupted parties and the secret, they might in principle encode more information than the output. We can relax this requirement to allow arbitrary secret sharing scheme, like ramp schemes, simply by requiring that the computation phase followed by a reconstruction phase consisting of making the shares of the output public is secure against $t < n/2$ corrupted parties.

We can also deal with protocols which leak partial information about the outputs during the computation, as opposed to the strict compute-reveal paradigm imposed above. Such protocol are broken into phases which leak no information and each phase is transformed as below, and in each phase a new dummy execution is selected.

Finally we can in a straight forward manner generalize to deal with schemes where the secret sharing schemes used to share the inputs and the outputs are not the same. This might be convenient for schemes converting between different secret sharing scheme, where circuit based techniques are not a convenient way to get security against deviations.

A.2 Protocol

The protocol π_f is given below. For notational simplicity we assume that there are $\ell = n$ inputs and that each P_i has one input $x_i \in K$. The protocol runs in

[4] In the view of the corrupted parties the entropy of the honest shares given s is 0.

the $(\mathcal{F}_{\text{INPUT}}, \mathcal{F}_{\text{COMMIT}}, \mathcal{F}_{\text{TRANSMIT}}, \mathcal{F}_{\text{FLIP}}^{n-1})$-hybrid model with parties $P_1, \ldots, P_n$ and proceeds as follows:

1. Each P_i inputs x_i to $\mathcal{F}_{\text{INPUT}}$ to receive shares $(x_i^{(j,0)})_{j=1}^n$ and $(x_i^{(j,1)})_{j=1}^n$.
2. Each P_i samples uniformly random seeds $K_i^{(0)}$ and $K_i^{(1)}$ for a pseudo-random generator, prg, and commits to these values using $\mathcal{F}_{\text{COMMIT}}$.
3. Each P_i computes $s_i^{(0)} = \text{prg}(K_i^{(0)})$ and $s_i^{(1)} = \text{prg}(K_i^{(1)})$ and for $c = 0, 1$ runs π_f with input $(x_i^{(j,c)})_{j=1}^n$ and randomness $s_i^{(c)}$. All messages are sent via $\mathcal{F}_{\text{TRANSMIT}}$.
4. For $j = 1, \ldots, n$ and $c = 0, 1$ party P_i uses $\mathcal{F}_{\text{COMMIT}}$ to commit to the its outputs $y_i^{(j,c)}$ and the messages $\text{comm}_{\{i,j\}}^{(i,c)}$ sent between P_i and P_j in the execution of $\pi_f^{(c)}$ (as seen by P_i).
5. Each P_i inputs (`flip`) to $\mathcal{F}_{\text{FLIP}}$ to receive $k \in [0, n-2]$. For $i = 1, \ldots, n$, let $l_i = (i + k \bmod n) + 1$.
6. All parties instruct $\mathcal{F}_{\text{INPUT}}$ to reveal d and $(x_{l_i}^{(j,d)})_{j=1}^n$ to P_i, each P_{l_i} instructs $\mathcal{F}_{\text{COMMIT}}$ to reveal $K_{l_i}^{(d)}$ to P_i, and P_i computes $s_{l_i}^{(d)} = \text{prg}(K_{l_i}^{(d)})$.
7. Each P_j instructs $\mathcal{F}_{\text{COMMIT}}$ to reveal $\text{comm}_{\{j,l_i\}}^{(j,d)}$ to P_i, and P_i checks that $\text{comm}_{\{j,l_i\}}^{(j,d)} = \text{comm}_{\{j,l_i\}}^{(l_i,d)}$ for all P_j.
   If $\text{comm}_{\{j,l_i\}}^{(j,d)} = \text{comm}_{\{j,l_i\}}^{(l_i,d)}$, then denote their common value by $\text{comm}_{\{j,l_i\}}^{(d)}$.
   If $\text{comm}_{\{j,l_i\}}^{(j,d)} \neq \text{comm}_{\{j,l_i\}}^{(l_i,d)}$, then P_i broadcasts a complaint and transfers $\text{comm}_{\{j,l_i\}}^{(j,d)}$ and $\text{comm}_{\{j,l_i\}}^{(l_i,d)}$ to the other parties who checks that $\text{comm}_{\{j,l_i\}}^{(j,d)} \neq \text{comm}_{\{j,l_i\}}^{(l_i,d)}$. Then P_j and P_i must instruct $\mathcal{F}_{\text{TRANSMIT}}$ to reveal the communication between them. A party failing to do so is detected as corrupted. Otherwise, the one which revealed $\text{comm}_{\{j,l_i\}}^{(\cdot,d)}$ inconsistent with the communication revealed by $\mathcal{F}_{\text{TRANSMIT}}$ is detected as corrupted.
8. Each P_{l_j} reveals to P_j the values $(y_{l_j}^{(i,d)})_{i=1,\ldots,n}$.
9. Each P_i runs the program of P_{l_i} in π_f on inputs $(x_{l_i}^{(j,d)})_{j=1}^n$, randomizer $s_{l_i}^{(b)}$ and the incoming messages from $\text{comm}_{\{j,l_i\}}^{(d)}$ $(j = 1, \ldots, n)$.

   Let $(y_{l_i}^{(j,d)'})_{j=1,\ldots,n}$ be the shares of the output computed by the program. If the messages sent by the program deviates from the outgoing messages in $\text{comm}_{\{j,l_i\}}^{(d)}$ $(j = 1, \ldots, n)$ or $(y_{l_i}^{(j,d)'})_{j=1,\ldots,n} \neq (y_{l_i}^{(j,d)})_{j=1,\ldots,n}$, then P_i broadcasts (`complaint`) and transfers the appropriate values to let the other parties verify the inconsistency. If the inconsistency is confirmed, the other parties output $(\texttt{corrupt}, l_i)$. If P_i broadcasts a complaint and does not demonstrate an inconsistency, the parties output $(\texttt{corrupt}, i)$.
10. If no party broadcast a complaint, then the parties instruct $\mathcal{F}_{\text{COMMIT}}$ to reveal $y_i^{(j,1-d)}$ towards P_j. Then P_j collects $n - t$ values $y_i^{(j,1-d)}$, reconstructs $y^{(j,1-d)}$ and outputs $y_j = y^{(j,1-d)}$.

A.3 Simulator

The first tool we need is that we can predict what each party *should* send in each round by inspecting $\mathcal{F}_{\text{INPUT}}$ and $\mathcal{F}_{\text{COMMIT}}$ in Step 2. This is done as follows.

1. For each P_i, let $(x_i^{(j,0)})_{j=1}^n$ and $(x_i^{(j,1)})_{j=1}^n$ be the outputs to P_i in Step 1.
2. Let $K_i^{(0)}$ and $K_i^{(1)}$ be the values input to $\mathcal{F}_{\text{COMMIT}}$ in Step 2 and let $s_i^{(0)} = \text{prg}(K_i^{(0)})$ and $s_i^{(1)} = \text{prg}(K_i^{(1)})$.
3. For $i = 1, \ldots, n$ and $c = 0, 1$, let $\pi_i^{(c)}$ denote the code of P_i in π_f initialized with inputs $(x_i^{(j,c)})_{j=1}^n$ and randomizer $s_i^{(c)}$. Now each $\pi_i^{(c)}$ is ready to send the first round of messages.
4. As long as not all $\pi_i^{(c)}$ terminated, run the $\pi_i^{(c)}$ to produce the messages that it should send in the next round, and input these message the corresponding $\pi_j^{(c)}$.
5. Let $\text{pcomm}_{\{i,j\}}^{(c)}$ be the messages sent between $\pi_i^{(c)}$ and $\pi_j^{(c)}$ above, and let $\text{pshare}_i^{(c)} = (y_i^{(j,c)})_{j=1}^n$ be the shares computed by $\pi_i^{(c)}$.

We now define some different types of deviations.

1. Inspect $\mathcal{F}_{\text{COMMIT}}$ to get the values $\text{comm}_{\{i,j\}}^{(i,c)}$ committed to by all parties. If $\text{comm}_{\{i,j\}}^{(i,c)} \neq \text{comm}_{\{i,j\}}^{(j,c)}$, then say that an *inter-deviation* involving P_i and P_j occurred in track c. If P_i is honest, then define P_j to be the *deviator*. If P_j is honest, then define P_i to be the deviator. If P_i and P_j are both corrupted, then define the lower indexed to be the deviator.
2. If there are no inter-deviations, then define $\text{comm}_{\{i,j\}}^{(c)}$ to be the common value of $\text{comm}_{\{i,j\}}^{(i,c)}$ and $\text{comm}_{\{i,j\}}^{(j,c)}$. Now, if $\text{comm}_{\{i,j\}}^{(c)} \neq \text{pcomm}_{\{i,j\}}^{(c)}$ for some P_i and P_j, then say that an *intra-deviation* occurred in track c and define the deviator $P_k \in \{P_i, P_j\}$ be the lowest indexed party to send a message different from $\text{pcomm}_{\{i,j\}}^{(c)}$ to some other party.
3. If there are no intra-deviations either, then all parties followed the passive protocols, which defines consistent output shares $\text{pshare}_i^{(c)} = (y_i^{(j,c)})_{j=1}^n$ for $i = 1, \ldots, n$. If any P_i commits to a value different from the predicted $y_i^{(j,c)}$, say that a *commitment deviation* occurred in track c and define P_i to be the *deviator*.

It is straight forward to see that if there are no deviations of any type in any track, then all parties followed the passive protocols and committed to correct shares. Furthermore, if any deviation occurred in the dummy track and the associated deviator P_i is checked by a honest party, then some corrupted P_j will be detected, though not necessarily exactly P_i. Finally, it can be seen that by the end of Step 4 we can define which deviations occurred and who are the deviators, and at this point the bit d and the index k are uniformly random and independent of the view of t corrupted parties. Putting these observations together, we see that if it is not the case that all parties followed the passive

protocols and committed to correct shares, then a corrupted party is detected with probability $p/2$, where p is the probability that a deviator is checked by an honest party. If there is just one deviator, then $p = (n-t)/(n-1) > 1/2$.

The above intuitively shows that the deterrence factor is $1/4$. We now show that we can actually simulate the protocol. The simulator is given in below:

1. First $\mathcal{S}$ receives $(x_i)_{i\in C}$ from $\mathcal{F}_{\text{EVAL}}$ and outputs x_i to $\mathcal{A}$ as if coming from a corrupted P_i in π_f.
2. In Step 1 $\mathcal{S}$ inspects the inputs of $\mathcal{A}$ to $\mathcal{F}_{\text{INPUT}}$ and records the inputs x'_i it gives on behalf of P_i, $i \in C$. It then inputs $\{x'_i\}_{i\in C}$ to $\mathcal{F}_{\text{EVAL}}$ and gets back $\{y_i\}_{i\in C}$, where $(y_1, \ldots, y_n) = f(x'_1, \ldots, x'_n)$, where $x'_i = x_i$ for $i \in H$ is the x_i received by $\mathcal{F}_{\text{EVAL}}$ from the environment.
3. In Step 1 $\mathcal{S}$ inspects the inputs of $\mathcal{A}$ to $\mathcal{F}_{\text{INPUT}}$ and records $(d_1, \ldots, d_n)$. Then it samples $[x^{(j,0)}] \leftarrow \text{sss}(d_j)$ and $[x^{(j,1)}] \leftarrow \text{sss}(d_j)$ for $j = 1, \ldots, n$ and outputs $(\{x_i^{(j,0)}\}_{j=1}^n, \{x_i^{(j,1)}\}_{j=1}^n)$ to P_i as if coming from $\mathcal{F}_{\text{INPUT}}$.
4. Then $\mathcal{S}$ runs Step 2, using random $K_i^{(c)}$ for $i \in H$ and inspecting the inputs of $\mathcal{A}$ to $\mathcal{F}_{\text{COMMIT}}$ to learn $K_i^{(c)}$ for $i \in C$. It then computes $s_i^{(c)} = \text{prg}(K_i^{(c)})$ for $i = 1, \ldots, n$.
5. Then $\mathcal{S}$ runs Step 3 and Step 4 honestly.
   Cheating is detected and handled as follows. For $i = 1, \ldots, n$ and $c = 0, 1$ the simulator computes the predicted values $\text{pcomm}_{\{i,j\}}^{(c)}$ and $\text{pshare}_i^{(c)}$ from the values $\{x_i^{(j,c)}\}_{j=1}^n$ and $s_i^{(c)}$ computed above. During the simulation, $\mathcal{S}$ inspects the inputs from $\mathcal{A}$ to $\mathcal{F}_{\text{TRANSMIT}}$ and $\mathcal{F}_{\text{INPUT}}$. If some P_i, $i \in C$, makes any type of deviation in any track, then $\mathcal{S}$ pauses the simulation and lets $\delta \in \{0, 1\}$ denote the index of the execution in which the deviation occurred.
   (a) Transmits a value to P_j, $j \in H$ which is different from the one predicted by the pcomm values.
   (b) Commits to any comm value not consistent with the pcomm values.
   (c) Commits to a share $y_i^{(j,c)}$ different from the one predicted by the pshare values.
6. If there were deviations, then $\mathcal{S}$ inputs (`cheat`) to $\mathcal{F}_{\text{EVAL}}$ and gets back $(x_i)_{i=1}^n$ and a notification of whether it was caught or not.
   - If it was caught, then it lets $d = \delta$ and samples k uniformly at random from $i - 1 - H \bmod n$
   - If it was not caught, it samples $d \in \{0, 1\}$ with $\Pr[d = \delta] = \frac{1}{3}$ and $\Pr[d = 1 - \delta] = \frac{2}{3}$, and samples $k \in [0, \ldots, n-2]$ with k uniformly random in $i - 1 - H \bmod n$ with probability $\alpha = \frac{4}{3}(\frac{n-t}{n-1} - \frac{1}{4})$ and k uniformly random in $[0, n-2] \setminus i - 1 - H \bmod n$ with probability $1 - \alpha$.

   It then patches the simulated enriched execution, $e = 1 - d$, and finishes the simulation as follows:
   (a) The enriched execution used inputs computed as $[x^{(j,e)}] \leftarrow \text{sss}(d_j; r_j)$. Now $\mathcal{S}$ computes $r'_j = \text{patch}(d_j, r_j, C, x_j)$ for the $(x_i)_{i=1}^n$ received from $\mathcal{F}_{\text{EVAL}}$ and $[x^{(j,e)'}] \leftarrow \text{sss}(x_j; r'_j)$. This defines new shares $(x_i^{(j,e)'})_{j=1}^n$ for $i \in H$.

(b) Then $\mathcal{S}$ computes

$$(s_j^{(e)\prime})_{j=1}^n \leftarrow \text{patch}(([x^{(j,e)}])_{j=1}^n, (s_j^{(e)})_{j=1}^n, C, ([x^{(j,e)\prime}])_{j=1}^n).$$

(c) Then $\mathcal{S}$ finishes the execution of Step 3 using the patched $(x_i^{(j,e)\prime})_{j=1}^n$ as input to $P_i^{(e)}$ and using the patched $s_i^{(e)\prime}$ as randomizer for $P_i^{(e)}$.
(d) In Step 5 of π_f, $\mathcal{S}$ lets $\mathcal{F}_{\text{FLIP}}$ return the k that $\mathcal{S}$ computed in Step 5.
(e) In Step 6 of π_f, $\mathcal{S}$ lets $\mathcal{F}_{\text{INPUT}}$ return the d that $\mathcal{S}$ computed in Step 5.
(f) Then $\mathcal{S}$ finishes the simulation by simply following the protocol. For each P_i it instructs $\mathcal{F}_{\text{EVAL}}$ to output whatever P_i outputs in the simulation.

7. If there were no deviations in the simulation of Step 3 and Step 4, then $\mathcal{S}$ patches the enriched shares of corrupted outputs committed to by honest parties as follows.
   For each output $y^{(j)} = y_j$, $j \in C$, received from $\mathcal{F}_{\text{EVAL}}$ it finds the share $y_i^{(j,e)}$ committed to by P_i, $i \in C$. Then it computes

$$(y_i^{(j,e)})_{i\in H} = \text{complete}(y^{(j)}, (y_i^{(j,e)})_{i\in C})$$

   and changes the internal state of $\mathcal{F}_{\text{COMMIT}}$ to be consistent with P_i, $i \in H$, having committed to $y_i^{(j,e)}$ in Step 3.
   Then it finishes the simulation by honestly following the protocol. When some honest P_i outputs y_i in the simulation, $\mathcal{S}$ instructs $\mathcal{F}_{\text{EVAL}}$ to give the output of P_i. If some honest P_i outputs $(\texttt{corrupt}, j)$ in the simulation, $\mathcal{S}$ instructs $\mathcal{F}_{\text{EVAL}}$ to output $(\texttt{corrupt}, j)$ on behalf of P_i.

A.4 Analysis

In this section we show that the view produced by the simulator is indistinguishable from the view of the protocol. We look at two cases.

Case I There is a deviation in Step 3 or Step 4.
Case II There are no deviations in Step 3 and Step 4.

We start with Case II. Here there are the following differences between the simulation and the execution.

1. In the simulation the enriched execution is run with the inputs $(d_1, \ldots, d_n)$. In the execution it is run with $(x_1, \ldots, x_n)$.
2. In the simulation the outputs from P_i, $i \in H$ seen by the environment comes from $\mathcal{F}_{\text{EVAL}}$. In the execution they are the values computed by P_i by running the protocol.
3. In the simulation the shares $(y_i^{(j,e)})_{i\in H}$ for $j \in C$ are input to $\mathcal{F}_{\text{COMMIT}}$. They are computed as $\text{complete}(y^{(j)}, (y_i^{(j,e)})_{i\in H})$ where $y^{(j)}$ is output by $\mathcal{F}_{\text{EVAL}}$.

The view of the adversary and environment of the first difference can be shown to be negligible using the security of prg and the *patching* properties. First we change the simulation to use uniformly random s_i^e for honest P_i, which changes the view of the corrupted parties negligibly by the security of prg and the fact that the view of the enriched execution is not opened. Then we can change the simulation to use the correct input x_i for each honest P_i instead of d_i in the enriched execution. This will not change the view of the corrupted parties as we in Case II assume that they follow the protocol π_f correctly, which in particular allows us to apply he *patching* properties, which ensure that the view of the corrupted parties are independent of the inputs of the honest parties in π_f when all parties follow the protocol.

We then consider the second difference. In Case II we have no deviations in the execution of the enriched execution $\pi_f^{(e)}$. By the *correctness* property and the fact that $x^{(j,e)}$ is the value input to $\mathcal{F}_{\text{INPUT}}$ by P_i, it follows that in the protocol the sharing $[y^{(j,e)}]$ is a sharing of the valued obtained by computing $(y^{(j,e)})_{j=1}^n = f((x_j)_{j=1}^n)$ for the inputs x_j to $\mathcal{F}_{\text{INPUT}}$. In the simulation, $\mathcal{S}$ takes the inputs to $\mathcal{F}_{\text{INPUT}}$ from the corrupted parties and give these to $\mathcal{F}_{\text{EVAL}}$. As a result $\mathcal{F}_{\text{EVAL}}$ computes $(y^{(j,e)})_{j=1}^n = f((x_j)$, and hence the output to P_i will be the same as in the protocol.

We then consider the third difference. As we argued above, the value $y^{(j)}$, $j \in C$, received by $\mathcal{S}$ from $\mathcal{F}_{\text{EVAL}}$ in the simulation will be the same as the value $y^{(j,e)}$ of which $\pi_f^{(e)}$ computes a sharing $[y^{(j,e)}]$ in the execution. Furthermore, by the *correctness* property and the fact that all parties follow $\pi_f^{(e)}$ in Case II, we have that $[y^{(j,e)}] = (y_i^{(j,e)})_{i=1}^n \in \text{sss}$. By the *randomness minimality* it follows that $(y_i^{(j,e)})_{i\in H} = \text{complete}(y^{(j)}, (y_i^{(j,e)})_{i\in C})$ in the execution, i.e., $(y_i^{(j,e)})_{i\in H}$ have the exact same distribution as in the simulation.

We then consider Case I. If we ignore that the honest parties use pseudo-random s_i instead of truly random ones, which can be handled as above, it follows directly from the patching properties that the view seen by the corrupted parties have the same distribution as in an execution of the protocol with the d and k used by $\mathcal{S}$, the only difference being that in the execution $x^{(j,e)} = x_j$ from a beginning and in the simulation we patch from $x^{(j,e)} = d_j$ to $x^{(j,e)} = x_j$ after d is determined. Furthermore, in Case I $\mathcal{S}$ determines all outputs of $\mathcal{F}_{\text{EVAL}}$ and specify them to be the same as those in the simulation. This ensures that the view of the simulation by $\mathcal{A}$ and the environment is indistinguishable from their view of an execution of the protocol with the d and k used by $\mathcal{S}$. All that remains to be checked is thus that

1. When $\mathcal{S}$ is detected by $\mathcal{F}_{\text{EVAL}}$, there exists a corrupted party P_j for which all honest parties output $(\texttt{corrupt}, j)$.
2. If an honest party outputs $(\texttt{corrupt}, j)$, then P_j is corrupted.
3. The distribution of d and k are the same in the simulation and the execution.

The first property follows from picking $d = \delta$ and $k \in i - 1 - H \bmod n$ when $\mathcal{S}$ is caught. Picking $d = \delta$ ensures that the execution in which P_i deviated is

made the dummy one. Picking $k = i - 1 - h \bmod n$ for $h \in H$ ensures that $h + k \bmod n + 1 = i$, which ensure that the honest P_h will check P_i. This ensures that at least one corrupted party is caught.

The second property follows by inspection of the protocols.

We then consider the third property. We first argue that d is negligibly close to the right distribution and then argue that k has the correct distribution.

When $\mathcal{S}$ is caught we let $d = \delta$ with probability 1. When $\mathcal{S}$ is not caught we let $d = \delta$ with probability $\frac{1}{3}$. This means that we let $d = \delta$ with probability $\frac{1}{4} \cdot 1 + \frac{3}{4} \cdot \frac{1}{3} = \frac{1}{2}$. In the protocol the view of $\mathcal{A}$ is computationally independent of d up until it makes a deviation—it is perfectly independent if we let the honest parties use a uniformly random s_i instead of a pseudo-randomly generated one. This means that $\delta = d$ with probability negligibly close to $\frac{1}{2}$, or the bit δ, which is known by $\mathcal{A}$ would be a distinguisher for prg.

When $\mathcal{S}$ is caught we sample k uniformly random in $i - 1 - H \bmod n$ with probability 1. When $\mathcal{S}$ is caught we sample k uniformly random in $i - 1 - H \bmod n$ with probability α. This means that we pick k uniformly at random in $i - 1 - H \bmod n$ with probability $\frac{1}{4} + \frac{3}{4}\alpha$. Hence an element in $i - 1 - H$ is picked with probability

$$\left(\frac{1}{4} + \frac{3}{4}\alpha\right)\frac{1}{n-t} = \left(\frac{1}{4} + \frac{3}{4}\frac{4}{3}\left(\frac{n-t}{n-1} - \frac{1}{4}\right)\right)\frac{1}{n-t} = \frac{1}{n-1} \ .$$

Since $i - 1 - H \subset [0, n-2]$ and the remaining probability mass is distributed uniformly over $[0, n-2] \setminus i - 1 - H$, it follows that also an element in $[0, n-2] \setminus i - 1 - H$ is picked with probability $\frac{1}{n-1}$, so k is uniform over $[0, n-2]$ in the simulation as well as in the protocol.

B Chinese Whispers

In our protocol we have all parties commit to the messages they exchanged before the revelation of d and k. This might seem superfluous, as $\mathcal{F}_{\text{TRANSMIT}}$ was already used to commit the parties to their messages. The difference is that $\mathcal{F}_{\text{COMMIT}}$ also binds the corrupted parties to what they have exchanged among themselves, whereas $\mathcal{F}_{\text{TRANSMIT}}$ allows the corrupted parties to change their mind on what they exchanged among themselves. We give an example demonstrating that it makes a difference whether the corrupted parties are committed to was sent among them before we reveal who checks who.

Consider the following passively secure protocol called Chinese-Whispers for computing the identity function $(x_1, \ldots, x_n) \mapsto (x_1, \ldots, x_n)$.

1. In round 1 party P_1 should send 0 to P_2.
2. For $i = 2, \ldots, n-1$, in round i party P_i should take the bit sent from P_{i-1} and send it to P_{i+1}.
3. In round n all parties $i = 2, \ldots, n$ should behave as follows: If they received a 0 from P_{i-1} in round i, then they should output x_i. If they received a 1 then they should broadcast x_i and then output x_i.

CHINESE-WHISPERS is clearly passively secure. Consider now a setting where $P_1, \ldots, P_t$ are corrupted and where P_t sends 1 to P_{t+1} in round t, a serious deviation which lets the corrupted parties learn all the inputs of the honest parties. If we now let the honest parties check a corrupted party at random, then with probability going to 1, as n grows, there will be a corrupted party P_i, $i \in \{2, \ldots, t-1\}$, which is checked by a corrupted party. The adversary can then behave as if $P_1, \ldots, P_{i-1}$ all received and sent 0, that P_i received 0 and sent 1, and $P_{i+1}, \ldots, P_t$ all received and sent 1. Now all the parties except P_i followed the protocol, and since the local consistency of P_i is not checked this goes unnoticed. If we had forced the adversary to picked the party P_i having received 0 and sent 1 before the assignment of who checks who was made public, then P_i would have been checked by an honest party with probability at least $\frac{1}{2}$, which is what we need.

CHINESE-WHISPERS itself does not fit as a protocol we can transform, but can be embedded into any such protocol to render it vulnerable to the above attack.