# The IPS Compiler: Optimizations, Variants and Concrete Efficiency*

Yehuda Lindell† Eli Oxman† Benny Pinkas†

August 12, 2011

**Abstract**

In recent work, Ishai, Prabhakaran and Sahai (CRYPTO 2008) presented a new compiler (hereafter the IPS compiler) for constructing protocols that are secure in the presence of malicious adversaries without an honest majority from protocols that are only secure in the presence of semi-honest adversaries. The IPS compiler has many important properties: it provides a radically different way of obtaining security in the presence of malicious adversaries with no honest majority, it is black-box in the underlying semi-honest protocol, and it has excellent asymptotic efficiency.

In this paper, we study the IPS compiler from a number different angles. We present an efficiency improvement of the "watchlist setup phase" of the compiler that also facilitates a simpler and tighter analysis of the cheating probability. In addition, we present a conceptually simpler variant that uses protocols that are secure in the presence of covert adversaries as its basic building block. This variant can be used to achieve more efficient asymptotic security, as we show regarding black-box constructions of malicious oblivious transfer from semi-honest oblivious transfer. In addition, it deepens our understanding of the model of security in the presence of covert adversaries. Finally, we analyze the IPS compiler from a *concrete efficiency* perspective and demonstrate that in some cases it can be competitive with the best efficient protocols currently known.

**Keywords:** secure computation, malicious, semi-honest and covert adversaries, asymptotic and concrete efficiency, the IPS compiler

*Research generously supported by the European Research Council as part of the ERC project LAST. The first author was also supported by the ISRAEL SCIENCE FOUNDATION (grant No. 781/07).

†Dept. of Computer Science, Bar-Ilan University, Israel. Email: lindell@biu.ac.il, eli.oxman@gmail.com, benny@pinkas.net.

# 1 Introduction

In the setting of secure multiparty computation, a set of parties wish to jointly compute some function of their inputs while preserving security properties such as *privacy*, *correctness*, *independence of inputs*, and more. These properties must be preserved in the face of adversarial behavior. In this paper, we consider security in the presence of three types of adversaries. The two classic adversary models are those of *semi-honest* adversaries that follow the protocol specification exactly but attempt to learn more than they should, and *malicious* adversaries that can behave as they wish and as such can arbitrarily deviate from the protocol specification. A more recent model, called security in the presence of *covert* adversaries, guarantees that if a malicious adversary behaves in a way that enables it to break the protocol in some way, then it will be caught cheating by the honest parties with some probability $\epsilon$ [1].

There are two rather distinct settings for studying this problem. In the first, it is assumed that a majority of the parties are honest. In such a case, it is possible to securely compute any efficient functionality with information-theoretic security [4, 7, 34, 2] assuming private channels (and broadcast, for the case of $n/3 \leq t < n/2$ corrupted parties). In the second setting, any number of the parties may be corrupted; this case includes the important two-party setting where one party may be corrupted. In this case of no guaranteed honest majority, information-theoretic security cannot be achieved. Nevertheless, assuming the existence of oblivious transfer, which can be constructed from enhanced trapdoor permutations and homomorphic encryption, it has been shown that any efficient functionality can be securely computed without an honest majority [36, 19].

Beyond proving an important theorem stating that any functionality can be securely computed without an honest majority, the construction of [19] shows how to *compile* any protocol that is secure in the presence of semi-honest adversaries into a protocol that is secure in the presence of malicious adversaries, using one-way functions alone. This result is therefore often referred to as the GMW compiler. Recently, a new compiler was presented by Ishai, Prabhakaran and Sahai (IPS) [27]. This compiler works in a completely different way to that of the GMW compiler. First, unlike GMW, it does not compile an $m$-party protocol for securely computing a functionality $f$ in the presence of semi-honest adversaries into an $m$-party protocol for securely computing the same $f$ in the presence of malicious adversaries. Rather, the IPS compiler uses $m$-party protocols that securely compute some basic operations (like addition and multiplication of shares in a finite field) in the presence of semi-honest adversaries with no honest majority, in order to transform a multiparty protocol that securely computes $f$ in the presence of malicious adversaries with an honest majority, to an $m$-party protocol that securely computes $f$ in the presence of malicious adversaries with *no honest majority*. As a specific example, note that by setting $m = 2$ the IPS compiler can generate a two-party protocol for computing $f$ that is secure against malicious adversaries, from two-party protocols for computing basic functionalities (secure against semi-honest adversaries), and a multiparty protocol for computing $f$ (secure against malicious adversaries which can corrupt only a minority of the parties). Intriguingly, the IPS compiler utilizes the world of information-theoretic secure computation in order to achieve security in the setting of no honest majority, since the basic protocol computing $f$, to which the compiler is applied, can be defined in an information-theoretic setting.

The IPS compiler has a number of important properties. First, it is black box in the underlying constructions; see [22] as to why this is of importance. Second, it provides a uniform approach to both the two-party and multiparty settings, like the GMW compiler. Third, in some settings, it has excellent asymptotic efficiency. This is due to the fact that in some cases (e.g., when

an arithmetic circuit computing $f$ is significantly smaller than a Boolean circuit computing $f$), information-theoretic protocols for the setting of an honest majority are much more efficient than computational protocols for the setting of no honest majority. The compiler can be applied to these more efficient protocols. This property has already been utilized to present protocols that have excellent theoretical, asymptotic efficiency [28]. Despite the above, it is unclear as to whether this approach can be used to achieve *concrete* efficiency for functionalities of interest [24].

### 1.1 The IPS Compiler

The compiler of [27] utilizes the following components in order to achieve the secure $m$-party computation of a functionality $f$, with no honest majority.

- A multiparty information-theoretic protocol $\pi$ computing $f$ with $m$ clients who provide input and $n = O(m^2 k)$ servers who carry out the computation (where $k$ is a security parameter analyzed in our work), which is secure in the presence of a malicious adversary corrupting a minority of the *servers*. This is called the *outer protocol* by [27].
- $m$-party subprotocols and local client instructions for simulating the server computation in $\pi$, that are secure as long as the adversary behaves in a semi-honest manner. Given the known techniques for information-theoretic secure multiparty computation, it suffices for example to use $m$-party subprotocols for securely computing additive shares of the product of shares (all other steps can be carried using local client instruction for generating shares, adding shares, and so on). We stress that these subprotocols need only be secure in the presence of semi-honest adversaries. These are called the *inner protocols* by [27].

The way that the compiler works is for the $m$ real parties to run the information-theoretic protocol $\pi$ by emulating the operations of the $n$ servers in $\pi$. This emulation is carried out using the secure $m$-party subprotocols, run between the $m$ real parties (clients), to compute the next step of all parties in $\pi$. Thus, the $n$ parties running $\pi$ are virtual and are emulated by the $m$ real parties running the protocol. Observe that if the real adversary were to behave in a semi-honest manner in each subprotocol, then the overall computation would clearly be secure. This is due to the fact that the emulation of the $n$ parties in $\pi$ is carried out using a protocol that is *secure* in the presence of semi-honest adversaries. Thus, $f$ is securely computed, as guaranteed by $\pi$. However, the adversary here may be *malicious*.

The magic in the IPS compiler is how to leverage the semi-honest security of the subprotocols in order to achieve security in the presence of malicious adversaries. The central observation is that in order for a malicious adversary to cheat, it must cheat in at least $n/2$ of the subprotocols. This is due to the fact that $\pi$ is secure unless a *majority* of the servers, namely at least $n/2$ of them, behave maliciously. In order to prevent such cheating, the IPS compiler sets up *watchlists*, which enable the honest parties to verify that malicious parties are not cheating. These watchlists are generated as follows. Each real party (client) chooses the randomness that it will use when running the semi-honest subprotocol for each virtual server. Then, using oblivious transfer, each other client obtains $k$ of the random strings of all other clients. Now, given the randomness that a party is supposed to use in the semi-honest subprotocol, it is possible to check that it is indeed behaving honestly. Furthermore, since oblivious transfer is used to obtain these strings, no client can know which semi-honest executions are being "watched" (this is why these $k$ strings obtained are called a *watchlist*). It is important to note, however, that it is not possible to make the watchlists too

large since corrupted parties view the same number of servers via the watchlists as honest parties. Now, each time the randomness of a client (used with respect to some server) is watched by a corrupted party, the internal state of the server is seen and that server is essentially corrupted. It is shown in [27] that $n = O(k^2 m)$ servers are required in order to obtain security with a probability of cheating that is negligible in $k$. This number is important because it determines the number of servers in the information-theoretic protocol and thus its complexity.

In our description of our results below, we assume that the reader is familiar with the details of the IPS construction. For those not so familiar, Section 2 contains a detailed description and tutorial-like explanation of the IPS compiler and how it works.

## 1.2 Our Results

### 1.2.1 Optimizations

As will become clear in our concrete analysis of the efficiency of the compiler, see Sections 1.2.3 and 6, the number $n$ of servers can become very large for some choices of parameters. This can have a severe effect on the efficiency of the protocol in a number of ways, one of those being the watchlist setup phase where $m^2 n$ executions of Rabin oblivious transfers must be carried out (each of these costing $\log n$ regular oblivious transfers; see Section 4 for a detailed explanation). This can therefore quickly become the bottleneck of the protocol. We therefore first devise a method for reducing the required number of servers to $n = O(mk)$ rather than $n = O(m^2 k)$, and second for setting up the watchlists in a way that costs an equivalent of $O(mn)$ regular oblivious transfers, rather than $O(m^2 n \log n)$ oblivious transfers. These optimizations are significant since they enable a better choice of parameters for the outer information-theoretic protocol. Specifically, the best efficiency is obtained by using a protocol with many servers and a small fraction of corrupted parties; this enables the heavy use of the packed or multi-secret sharing methodology of [14]. Using our method, a smaller number of virtual servers can be corrupted and so a more efficient protocol can be used. We stress that when measuring concrete complexity, our new watchlist setup protocol is substantially more efficient, even for the two-party case where $m = 2$.

In addition to the above, we observe that it is possible to use an outer information-theoretic protocol that provides hybrid security in the presence of some malicious parties and some semi-honest parties. This is due to the fact that the corruptions that are due to a malicious party actively cheating in a semi-honest protocol without getting caught are "malicious corruptions", whereas corruptions that are due to a malicious party watching a server through its legitimate watchlists are "semi-honest corruptions". This provides more flexibility in constructing the outer protocol and can yield better efficiency, as was shown in [13].

### 1.2.2 Variants

**A simple compiler from covert to malicious adversaries.** We present an analog of the IPS compiler that uses subprotocols that are secure in the presence of covert adversaries instead of protocols that are secure in the presence of semi-honest adversaries. Recall that a protocol is secure in the presence of covert adversaries, with a deterrent parameter $\epsilon$, if any cheating by an adversary is detected by the honest parties with probability at least $\epsilon$ [1]. (For our purposes, it is convenient to assume that $\epsilon = 1/2$.) The use of subprotocols with this level of security fits naturally with the IPS paradigm: the information-theoretic outer protocol is emulated using protocols that are secure in the presence of covert adversaries with deterrent $\epsilon = 1/2$. Then, if the adversary tries to cheat in

$k$ of the subprotocol executions, it will be caught except with probability $2^{-k}$. Observe that there is no need for any watchlists. In addition, the analysis and proof of security of this compiler is extraordinarily straightforward, since the cheating probability can be measured exactly with ease.

Beyond being a significant conceptual simplification, the usage of covert protocols also enables us to use an information theoretic protocol with just $n = m + 2k$ parties (tolerating up to $k$ corruptions), rather than $O(m^2k)$ or $O(mk)$ parties when using semi-honest security. Relying on the fact that protocols that are secure in the presence of covert adversaries with deterrent $\epsilon = 1/2$ are only about 2–3 times the cost of semi-honest protocols, this results in an *asymptotic* efficiency improvement over the original compiler. We stress, though, that by our concrete analysis, the original compiler of [27] will typically be more efficient for *concrete* parameters since the use of watchlists means that local computation by a party can be checked directly and need not be distributed.

**Covert security from semi-honest security.** We observe that the IPS compiler with some minor modifications can be used to obtain, in a black-box manner, security in the presence of covert adversaries from protocols that are secure in the presence of semi-honest adversaries. We show that it suffices to use watchlists of size $O(m)$ and an oblivious transfer protocol that is secure in the presence of covert adversaries to set up those watchlists. (We also show that covert oblivious transfer can be constructed at the cost of only a constant number of semi-honest oblivious transfers.) This answers a major open question left by the work of [10]. They show how to construct protocols that are secure in the presence of covert adversaries from protocols that are secure in the presence of semi-honest adversaries, in the information-theoretic setting with an honest majority. However, their techniques do not extend at all to the setting of no honest majority. We stress that our result is not a GMW-type compiler that takes a semi-honest protocol computing $f$ and generates a covert protocol computing $f$. Rather, it is a black-box reduction that works exactly like IPS, combining semi-honest protocols for related subprotocols with an information-theoretic protocol computing $f$.

**IPS compilation and covert adversaries.** Based on the above, we have that the IPS paradigm significantly contributes to our understanding of security in the presence of covert adversaries, and enables us to position covert adversaries in their natural place between semi-honest and malicious adversaries with respect to protocol constructions.

**Black-box malicious OT from semi-honest OTs with linear overhead.** We demonstrate the usefulness of working through covert security as in the above two IPS variants by showing that there is a *quantitative* advantage to be gained. Specifically, we show how to construct malicious oblivious transfer from one-way functions and semi-honest oblivious transfer in a black-box way, using only a *linear* number of invocations of the underlying semi-honest oblivious transfer protocol. The previously known construction for this problem required a *quadratic* number of invocations of the underlying semi-honest protocol [22]. (We remark that this result is of interest when a single or small number of oblivious transfers are being computed. When many oblivious transfers are being run, the original compiler of [27] suffices for obtaining constant asymptotic overhead.) The reason that our new techniques are able to obtain this asymptotic efficiency improvement is due to the fact that the watchlist setup phase of IPS requires oblivious transfer that is already secure in the presence of malicious adversaries, and the best known black-box construction of this requires a

quadratic number of semi-honest oblivious transfers. In contrast, our compilation using the covert model requires only oblivious transfer that is secure in the presence of covert adversaries.

### 1.2.3 Concrete Efficiency

On an *abstract level*, the IPS compiler provides an elegant and conceptually simple way of constructing protocols that are secure in the presence of malicious adversaries. However, the actual instantiation of a protocol using the IPS approach is dependent on many different parameters and choices, all having a significant effect on the concrete efficiency of the result. To start with, appropriate inner and outer protocols must be chosen, and these choices are interdependent. This is due to the fact that the most efficient information-theoretic outer protocol may require more invocations of the inner protocol for computing multiplications (which may be more expensive than other operations) than a less efficient protocol, when judging efficiency in the standard information-theoretic setting. Thus, the cost of running the inner multiplication protocol must be traded off with the cost of other operations in the outer protocol. In addition, there may be other outer protocols that can utilize different inner protocols and obtain higher efficiency.

Another parameter that must be chosen is the exact number of servers $n = O(mk)$. It is clear that $n > mk + O(k)$ since $m - 1$ corrupted parties have already effectively corrupted $(m-1)k$ servers since they see $(m-1)k$ servers in their watchlists of the honest parties. However, how large should $n$ be? A naive approach which is to take $n$ to be the smallest possible function of $k$ actually may have the opposite effect. For example, if $m = 2$ and $n = 4k$ then the corrupt party, which corrupts $k$ servers through its watchlist, needs to control $k$ more servers, or a fraction of $1/4$ of the servers, in order to cheat in the outer protocol. If $n = 3k$ then the adversary needs to control $k/2$ more servers, namely a fraction of $1/6$ of the servers. The probability of it being caught in the latter case is smaller, and therefore $k$, and subsequently $n$, might have to be larger in that case than when $n = 4k$.

In order to be concrete, let the number of clients $m$ equal 2, and consider an outer protocol that tolerates any $t < n/2$ corruptions. We briefly analyze the difference between setting $n = 4k$ and $n = 3k$, where $k$ is the number of servers in the watchlist. In the case of $n = 4k$, the adversary needs to corrupt an additional $k$ servers in order to have a dishonest majority of $2k$ servers, and this requires cheating in the simulation of at least $k$ servers in the inner semi-honest protocols. The honest party does not detect this if its watchlist contains only servers in the other $3k$ (out of $4k$) servers. A rough analysis gives that the probability that the adversary succeeds in this case is $(3/4)^k$. In contrast, if $n = 3k$ then the adversary needs to corrupt an additional $k/2$ servers and so it successfully cheats with probability only $(2.5/3)^k = (5/6)^k$. Setting a fixed error of $2^{-40}$, we have that when $n = 4k$ we need to set $k = 97$ and so $n = 388$, and when $n = 3k$ we need to set $k = 152$ and so $n = 456$. We therefore conclude that it is better to take $n = 4k$ than $n = 3k$ since this results in a lower number of servers $n$ relative to the same cheating probability of $2^{-40}$.

The above analysis relates to an outer protocol that tolerates any $t < n/2$ corruptions. However, the best protocols for this setting [11] use the packed secret sharing methodology of [14]. This methodology, described in Section 6, enables the effective multiplication of an entire block of shares using a single multiplication protocol, as well as other efficiency improvements. Thus, large blocks can significantly lower the complexity of the protocol. However, the outer protocol must have the property that the number of corrupted parties that can be tolerated is upper bounded by the difference between the secret sharing threshold (which for [11] must be less than $n/4$) and the block size. For example, if we take a block of size $n/5$ then the number of corrupted parties in the

outer protocol must be at most $n/4 - n/5 = n/20$. However, the calculation that we carried out above regarding the size of $n$ assumed that $n/2$ corruptions can be tolerated. Thus, many more servers are needed, and we need to recalculate the required number from scratch. For example, if we simply take $3,880$ servers (10 times the above) and use the same $k = 97$, then we have that at most $3880/20 = 194$ parties can be corrupted and the adversary needs to corrupt an additional 97 parties. Thus, it successfully cheats with probability approximately $(3783/3880)^{97}$ which equals 0.08. Thus, the adversary can cheat successfully with probability that is way too high and different parameters must be chosen. This demonstrates the complexity of choosing good parameters since they are all interdependent. Observe that in our concrete analysis, we use $k$ as the size of the watchlists and not an independent security parameter; in the above asymptotic treatment these were the same.

We remark finally that additional complications that arise are due to the specific structure of the circuit being used (some circuits can utilize the packed secret sharing optimization more than others), and the fact that the number of servers needed here is typically much more than that considered in the standard information-theoretic setting. In order to see why this is significant, we consider the concrete example of secure computation of the AES function. A small arithmetic circuit of only 2,400 gates was constructed for this function [12]. Since the size of the circuit has a huge impact on the complexity of the protocol, the fact that such a small circuit can be designed for this function is significant (note that an analogous Boolean circuit is of size 33,000 gates [31]). However, the circuit design relies inherently on the fact that the field used is $GF[2^8]$. In the IPS setting, the number of servers will almost always be considerably larger than this, and so either this circuit cannot be used, or secret sharing based on algebraic-geometric codes must be used [8], resulting in a different inner protocol.

## 2 A Description the IPS Compiler

In this section we describe the IPS compiler in detail. The aim of the presentation here is to present the compiler and explain how and why it works on an intuitive level. For a formal presentation and proofs of security, we refer the reader to the original IPS papers [27, 28].

**The basic idea.** The main idea behind the IPS construction is to have a number of parties, amongst which any number may be corrupted, simulate the execution of an information-theoretic protocol that is secure when a majority of the parties are honest. From here on we call the real parties running the protocol (with no honest majority) *clients*, and we call the simulated parties for the information-theoretic protocol *servers*. In Figure 1, two real clients $P_1$ and $P_2$ simulate an information-theoretic protocol with $n$ servers $S_1, \ldots, S_n$. This simulation is carried out by *distributing* the computation of each server amongst the clients, using appropriate subprotocols $\pi_1, \ldots, \pi_n$ for this task. That is, for each simulated server $S_i$, the clients $P_1$ and $P_2$ run a secure protocol $\pi_i$ computing the next message that $S_i$ should send. These subprotocols must be secure for any number of corrupted clients, since the clients run these subprotocols and no honest majority is guaranteed for them. In [27] the real subprotocols run by the clients are called the *inner protocols*, and the simulated protocol run by the simulated servers is called the *outer protocol*. We remark that the outer protocol is actually a protocol involving clients and servers, where the clients provide input and receive output and the servers just aid in the computation. The security requirement is that the protocol be secure for any number of corrupted clients as long as a majority of the servers are honest. Standard information-theoretic protocols can be cast in this model by simply having

the clients share their inputs (via VSS) with the servers, who then carry out the computation and return the output shares to the clients for reconstruction.

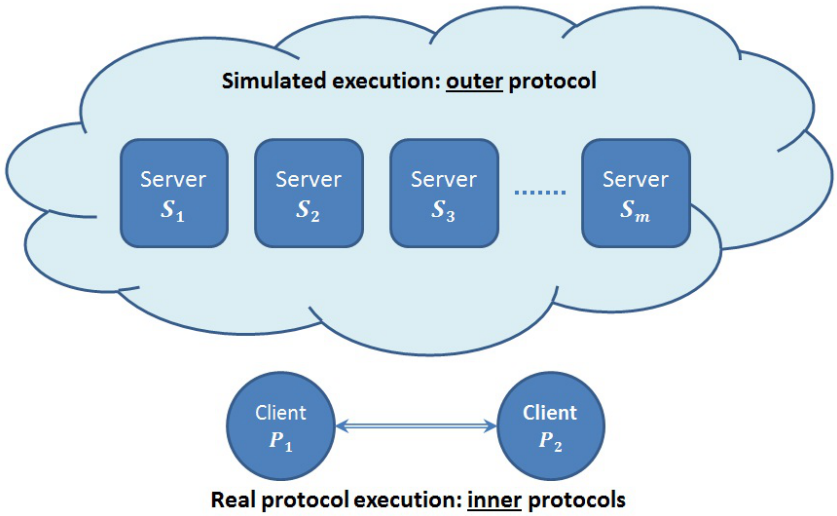


Figure 1: The IPS compiler – two parties simulate an $n$-party honest-majority protocol

It is clear that if the clients run inner protocols that are secure in the presence of a malicious adversary, then the simulation of the outer protocol will emulate an execution with an honest majority of servers. In fact, this simulation will emulate an execution where no servers are corrupted at all (this is because the the inner protocols are secure in the presence of malicious adversaries and so no malicious adversary can cause any server emulation to deviate from the protocol specification). However, the aim of the IPS compiler is to construct a protocol that is secure in the presence of malicious adversaries from inner protocols that are only secure in the presence of semi-honest adversaries (the outer protocol does need to be secure in the presence of malicious adversaries, but this can be information theoretic and so in some sense "simpler"). However, if the clients run inner protocols that are only secure in the presence semi-honest adversaries, then it is possible for a malicious client to actively cheat and possibly completely break the outer protocol. Thus, a mechanism is needed to prevent the clients from cheating "too much" in the inner protocol (we say "too much" because by the resilience of the outer protocol to a dishonest minority, we can tolerate cheating in some of the inner protocols). This mechanism uses a novel technique called *watchlists*.

**The inner protocol and the watchlist mechanism.** The servers in the outer protocol have no input, and therefore their actions in every step are a function of their random tapes and the incoming messages that they see (equivalently, their actions are a probabilistic function of the messages sent in the protocol). For each server $S_i$, the clients run "inner protocols" that compute the actions of $S_i$, with security in the presence of semi-honest adversaries. In order to explain how cheating is prevented in the protocol that simulates $S_i$, denote by $\pi_i$ the semi-honest protocol used to simulate $S_i$. This protocol is *reactive*, meaning that it is invoked in stages where each stage represents a round in the outer protocol. Furthermore, in each stage, its input is its state after the previous invocation and the vector of messages sent by all other servers in the previous round; of course, these inputs are shared between the parties (this sharing can be simple additive sharing). The protocol then outputs a sharing of the state after this execution, and a sharing of the messages

sent by server $S_i$ to all other servers in this round.

Now, in order to prevent a client from cheating in $\pi_i$, we must verify two things. First, we must ensure that it runs the instructions of the honest party in every invocation of $\pi_i$, relative to some random tape.[1] Second, we must ensure that it uses the correct inputs in every invocation. That is, we must verify that it inputs its share of the state after the previous invocation and its shares of *all* of the messages of the previous round that are intended for server $S_i$. If both of the above are enforced, then by the security of $\pi_i$ the simulation of server $S_i$ yields an "honest" server in the outer protocol. We now describe how each type of check is carried out:

1. *Watchlist for protocol instructions:* The purpose of this watchlist is to ensure that each client runs the instructions of the honest party in every invocation of $\pi_i$, relative to some random tape. In order to see how this works, observe that each client's actions in $\pi_i$ are *fully determined* by its random tape and its input in this invocation. Thus, if one client (say, $P_2$) is given the random tape of another client (say, $P_1$) in protocol $\pi_i$, then $P_2$ can verify that $P_1$ follows the instructions exactly by just comparing the messages sent to those that should be sent. We stress that $P_2$ can compute these messages exactly because they are a deterministic function of the random tape and input. (Recall that there are two types of input: the sharing of the state from the previous invocation and the sharing of all the messages sent to the server from the previous round. The former is known since $P_2$ has the random tape of $P_1$ and so can compute $P_1$'s output from the previous invocation, which includes the sharing of the state; the latter is dealt with below by the second type of watchlist.) See Figure 2 for this type of watchlist.

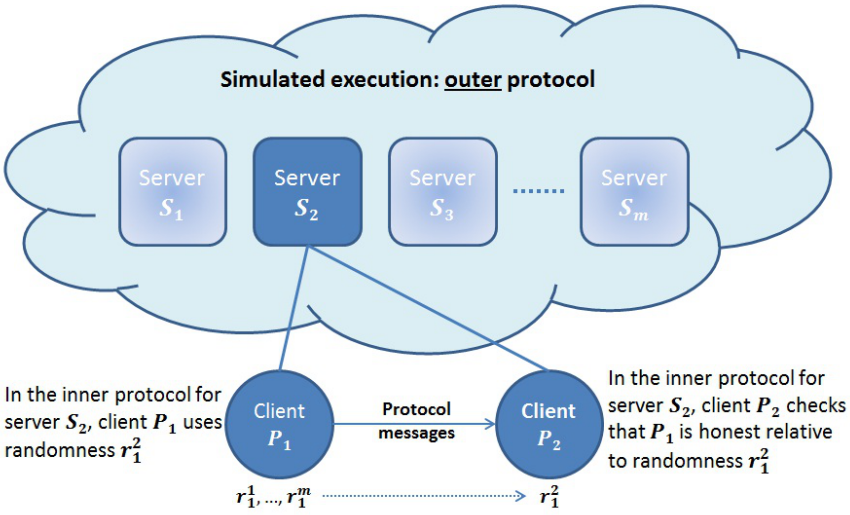


Figure 2: Protocol-instruction watchlist – client $P_2$ checks $P_1$ in the simulation of server $S_2$

In order to set up this watchlist, $P_1$ chooses random tapes $r_1^1, \ldots, r_1^n$ where $r_1^i$ is the random tape that it uses in subprotocol $\pi_i$ (for simulating $S_i$), and likewise $P_2$ chooses random tapes $r_2^1, \ldots, r_2^n$. Then, $P_2$ obtains from $P_1$ a random subset $W_1 \subset \{r_1^1, \ldots, r_1^n\}$ of $k$ out of the

[1] In actuality, a semi-honest protocol may only be secure when the random tape is uniformly distributed, and not chosen in an arbitrary way by the party. In such a case, a coin tossing protocol must be used. To simplify the explanation here, we assume that $\pi_i$ is secure for any choice of random tape. This additional property is often not difficult to achieve.

$n$ random tapes chosen by $P_1$; likewise, $P_1$ obtains a random subset $W_2$ of $k$ out of the $n$ random tapes chosen by $P_2$. We say that $P_2$ watches $P_1$ for server $S_i$ if $r_1^i \in W_1$, and we say that $P_1$ watches $P_2$ for server $S_i$ if $r_2^i \in W_2$. These subsets are obtained using an oblivious transfer type protocol, so that $P_1$ knows nothing about which strings are in $W_1$, and $P_2$ knows nothing about which strings are in $W_2$.

2. *Watchlist for protocol inputs:* As we have discussed, it is also necessary for a client to verify that each client uses the correct inputs in every invocation of the subprotocols. In particular, the check that we have described for protocol instructions can only be carried out when the client carrying out the check knows the input that the other client is supposed to use in this invocation. Furthermore, the check is only meaningful if the input is correct. Specifically, if the only checks carried out are via the protocol instruction watchlists, then it may be possible to cheat by simply changing the inputs in every invocation. Thus, we need to ensure that the input used is correct. (Note that the part of the input relating to the messages sent to $S_i$ in the previous phase is comprised of output generated in the previous invocations of *all* $\pi_1, \ldots, \pi_n$, since the message sent by $S_j$ to $S_i$ is computed in $\pi_j$.)

   In order to carry out this check, each client chooses a different secret key for each server. To be concrete, denote the keys chosen by $P_1$ by $k_1^1, \ldots, k_1^n$, and denote the keys chosen by $P_2$ by $k_2^1, \ldots, k_2^n$. Then, for every $j = 1, \ldots, n$, at the end of the execution of the inner protocol $\pi_j$ for a given round of the outer protocol, client $P_1$ (resp., $P_2$) encrypts the share of the message that $S_j$ should send to $S_i$ in this round under the key $k_1^i$ (resp., $k_2^i$); recall that the shares of the message that $S_j$ sends to $S_i$ are part of the output of $\pi_j$. Thus, given $k_1^i$ it is possible for $P_2$ to reconstruct all of the messages sent to $S_i$ from all servers $S_1, \ldots, S_n$ in any given round. Furthermore, for every $r_1^j \in W_1$, party $P_2$ knows the output that $P_1$ should have received in this invocation of $\pi_j$ and in particular the share of the message that $S_j$ sends to $S_i$. Thus, $P_2$ can verify that $P_1$ encrypts this exact share under $k_1^i$. This prevents $P_1$ from changing the shares of the messages sent. Of course, this is symmetric and so $P_1$ carries out the same checks. See Figure 3.

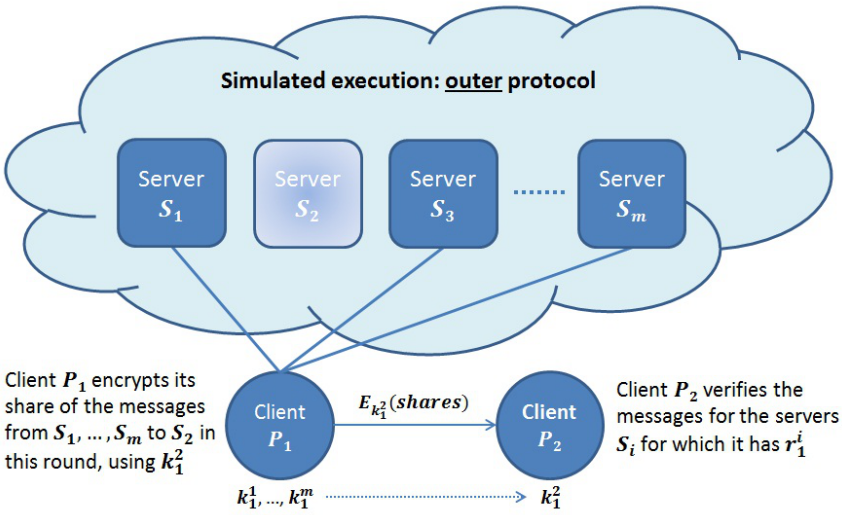


Figure 3: Protocol-input watchlist

As with the previous type of watchlist, the clients obtain the subset of keys obliviously so that the other party does not know where it is being checked. We stress that the clients must choose the random tapes and keys to watch consistently (i.e., $r_1^i$ together with $k_1^i$ and so on) so that the checks can be made.

It is important to note that when a malicious client $P_1$ has the pair of values $r_2^i, k_2^i$, this is equivalent to it *corrupting* server $S_i$ in the outer protocol. This is due to the fact that $P_1$ can see all of $S_i$'s internal state. Thus, when the watchlists are of size $k$, this is equivalent to a malicious party already corrupting $k$ servers. In addition, it is possible for a party to actively cheat in a small number of inner protocol executions, and the probability that it will be caught is not high. The combination of these two factors implies that the number of servers must be not too low. In the analysis carried out by [27], they cite that the number servers $n$ must be $O(m^2k)$. We reduce this to $O(mk)$ here. Intuitively, this number of servers is needed since $m-1$ corrupt clients view $(m-1)\cdot k$ servers via their watchlists and so have "corrupted" $(m-1)\cdot k$ servers already, as described above. In addition, if a malicious adversary tries to have corrupted clients cheat in $k$ or more of the inner protocols $\pi_1, \ldots, \pi_n$, then it will be caught with overwhelming probability. (This is because it only escapes being caught if the $k$ servers in the watchlist of the honest client do not intersect at all with the $k$ inner protocols in which it cheats. For a constant $m$, the probability of this happening is negligible in $k$.) We therefore have that with $n = 2mk$ servers, an honest majority is guaranteed except with negligible probability. We observe that some more efficient outer protocols tolerate only a quarter or even less corrupted parties; in these cases we can take $n = 4mk$ or more, depending on the required tolerance.

**A concrete instantiation of IPS for the VSS protocol of BGW [4].** In order to illustrate how the above method works, we informally describe how the computation would work with two parties running the BGW protocol for verifiable secret sharing (VSS) [4]. We assume familiarity with this protocol, but nevertheless begin by briefly describing it (we assume that the secret $s$ is from some finite field of size at least $n$, and we let $\alpha_1, \ldots, \alpha_n$ be some fixed field elements). In order to make this description consistent with IPS, we refer to "servers" rather than to "parties":

1. The dealer, with secret $s$, selects a uniformly distributed bivariate polynomial $S(x, y)$ of degree $t$ in both variables, under the constraint that $S(0, 0) = s$.

2. For every $i = 1, \ldots, n$, the dealer sends server $S_i$ the two univariate polynomials $f_i(x) = S(x, \alpha_i)$ and $g_i(y) = S(\alpha_i, y)$; these are $S_i$'s *shares*.

3. For every $i = 1, \ldots, n$, server $S_i$ sends $f_i(\alpha_j), g_i(\alpha_j)$ to server $S_j$, for every $j = 1, \ldots, n$.

4. For every $i = 1, \ldots, n$, server $S_i$ denotes the pair of elements received from $S_j$ by $(u_j, v_j)$. If $u_j \neq g_i(\alpha_j)$ or $v_j \neq f_i(\alpha_j)$ then $S_i$ broadcasts a message $\mathsf{complaint}(i, j, f_i(\alpha_j), g_i(\alpha_j))$ to all parties (in such a case, we say that these values are *inconsistent*).

5. The remainder of the protocol deals with the complaint resolution; as we will see below this is not needed in the IPS setting.

We now proceed to describe one possible IPS instantiation of the above VSS protocol for two parties. Let $P_1$ be the client playing the dealer, with the secret $s$.

1. *Watchlist setup for $P_1$:* Client $P_1$ chooses $n$ random strings and key pairs $(r_1^1, k_1^1), \ldots, (r_1^n, k_1^n)$. The parties then run an oblivious watchlist setup protocol in which $P_2$ receives a random subset of $k$ pairs $(r_1^i, k_1^i)$.

2. *Watchlist setup for $P_2$:* Client $P_2$ chooses $n$ random string and key pairs $(r_2^1, k_2^1), \ldots, (r_2^n, k_2^n)$. The parties then run an oblivious watchlist setup protocol in which $P_1$ receives a random subset of $k$ pairs $(r_2^i, k_2^i)$.

3. *The dealer client $P_1$ sends shares to all servers:* In order to do this, $P_1$ first sends the shares over the watchlist for protocol inputs. Technically, it encrypts the univariate polynomials $f_i(x), g_i(y)$ to be obtained by server $S_i$ with the key $k_1^i$, and sends all the encrypted values to the other client $P_2$.

4. *The servers exchange subshares $f_i(\alpha_j), g_i(\alpha_j)$:* This step seems to require a subprotocol $\pi_i$ for every server $S_i$ in order to compute $f_i(\alpha_j), g_i(\alpha_j)$ from an encryption of $f_i(x), g_i(y)$. However, since this step does not modify the state of any server and is deterministic, it can actually be computed singlehandedly by the client (this is called a type I computation in [27]). Thus, the dealer $P_1$ computes $f_i(\alpha_j), g_i(\alpha_j)$ for every $i$ and $j$, and encrypts $\langle f_1(\alpha_j), g_1(\alpha_j), \ldots, f_n(\alpha_j), g_n(\alpha_j)\rangle$ under key $k_1^j$ and sends all of the ciphertexts to $P_2$ (for all $j = 1, \ldots, n$).

5. *The servers check consistency:* Client $P_2$ decrypts the polynomials $f_i(x), g_i(y)$ and the subshares $\langle f_1(\alpha_i), g_1(\alpha_i), \ldots, f_n(\alpha_i), g_n(\alpha_i)\rangle$ for every $i$ on its watchlist and verifies consistency (i.e., that $g_i(\alpha_j) = f_i(\alpha_j)$ and $f_i(\alpha_j) = g_j(\alpha_i)$, for every $j$).

6. *Accept or abort:* In the VSS protocol above, in the case of inconsistency the servers broadcast complaints and then begin a phase of complaint resolution. The aim of this resolution phase is to determine whether the complaints are due to the fact that the dealer sent invalid shares, or due to the fact that some servers sent incorrect subshares. However, in the IPS translation of such a protocol, there is no need to resolve complaints since whenever cheating is detected the honest parties can just abort. Thus, if $P_2$ finds any inconsistent values it immediately aborts; otherwise $P_2$ accepts.

We informally describe why the above very simple protocol achieves the desired properties. In the proof of this VSS protocol for the standard multiparty setting, it is shown that as long as consistent values are dealt to at least $t + 1$ honest servers, the secret that is shared is well defined and fixed (and the polynomial is guaranteed to be of degree at most $t$). However, in the multiparty setting of the outer protocol, it is necessary that at the end of the VSS protocol *all* the shares of the at least $2t + 1$ honest parties are consistent. Thus, we need to guarantee that at most $t$ servers receive inconsistent values. Translating this to the IPS setting, we need $P_2$ to detect cheating if more than $t$ servers receive inconsistent values, or equivalently, if the subshares encrypted and sent by $P_1$ on the watchlist are inconsistent for more than $t$ servers. By setting $k$ and $n$ appropriately, it is possible to guarantee that this is not the case, except with negligible probability.

We remark that if computations must now be carried out by the servers based on these shared values, then the client $P_1$ must provide $P_2$ with an additive sharing of the VSS shares $f_i(x), g_i(y)$ for every $S_i$. This is achieved as described above. Specifically, $P_2$ has a subset of the random tapes that $P_1$ will use for computing $\pi_1, \ldots, \pi_n$ to simulate $S_1, \ldots, S_n$. Each random tape determines the additive share that $P_1$ generates. Thus, $P_1$ computes the additive sharing based on the random

tape, and sends the share for $P_2$ of $f_i(x), g_i(y)$ on the watchlist for protocol inputs by encrypting the shares with $k_1^i$. Client $P_2$ verifies that the sum of the share sent on the watchlist with the share defined by the random tape for $S_i$ equals the share $f_i(x), g_i(y)$ sent above over the watchlist. From this point on, the parties can carry out computations using inner protocols, as described.

# 3 Definitions

We refer the reader to [18, Chapter 7] for the definition of security for computation in the presence of semi-honest and malicious adversaries, and to [6] for the definition of universal composability. The notion of security in the presence of covert adversaries is defined in [1]. However, since it is less well known, we present the definition in Appendix A. We also need some very minor modifications to the definition to address the usage of a broadcast channel and of reactive functionalities; these modifications are described in Appendix A.3.

# 4 Optimizations of the IPS Compiler

## 4.1 Efficient Watchlist Setup

As we have mentioned, the first step in the IPS compiler involves the setup of watchlists. Recall that the number of real parties is $m$ and the number of virtual servers is $n$. Denote the real parties by $P_1, \ldots, P_m$. Technically, each party $P_i$ chooses $n$ random strings $r_i^1, \ldots, r_i^n$ (say, of length the security parameter $k$) and runs a two-party protocol with every other party $P_j$ in which $P_j$ receives $k$ of the strings without $P_i$ knowing which. (In Section 2, we explained that there are actually two types of watchlist, each with its own random string or key. Here we consider only a single random string, with the understanding that a longer string can be transferred and then different parts of it used for each type of watchlist.)

**The IPS setup.** The method proposed in [27] is for each pair of parties $P_i, P_j$ to run $n$ executions of Rabin oblivious transfer [33]; in the $\ell$th execution the sender $P_i$ inputs $r_i^\ell$ and the receiver $P_j$ obtains this string with probability $k/n$, and obtains nothing otherwise. The result of this procedure is that the expected number of strings obtained by the receiver is $n \cdot k/n = k$, as required. As described in [27] it is possible to construct a single Rabin oblivious transfer with receipt probability $k/n$ by running a single 1-out-of-$n$ oblivious transfer, which in turn can be constructed using $\log n$ regular 1-out-of-2 oblivious transfers [30]. Thus, the cost of this setup phase is $n \log n$ oblivious transfers for $P_i$ as sender and each $P_j$ as receiver, with an overall of $m(m-1)n \log n$ oblivious transfers between all pairs of parties. With $n = O(m^2 k)$ as stated in [27] we have that the cost is about $O(m^4 k \log m^2 k)$ oblivious transfers.

Concretely, this can be carried out in two ways; one is to use the efficient extending of oblivious transfers of [26]. However, in the case of malicious oblivious transfer the oblivious transfer extension is not so efficient since it is based on the cut-and-choose methodology; in addition, it requires the use of the less standard assumption of correlation-robust hash functions. Alternatively, one can use a highly efficient OT protocol with $O(1)$ exponentiations per transfer [32] (the cost here is 11 exponentiations per transfer, with UC and stand-alone variants at essentially the same cost).

**A new watchlist setup.** We propose a new watchlist setup with the following properties. First, the method guarantees that each malicious party views the *same* $k$ servers in all of its watchlists (i.e., for every honest party). As we will see, this means that it suffices to set $n = O(mk)$ instead of $O(m^2k)$. Second, it guarantees that each party views *exactly* $k$ servers in each of its watchlists; this enables a tighter and more straightforward analysis of the probability that an adversary can cheat. Finally, we present a concrete protocol that results in a considerable efficiency improvement over the IPS setup, for both aforementioned implementation options, and even when the *same* number of $m$ clients and $n$ servers are used (namely, when taking $n = O(mk)$ for both setups).

Recall that the aim of the watchlist setup procedure is for each party to obtain a watchlist of size $k$ that it can view, from all other parties. Rather than achieving this by having each pair of parties run a separate procedure, we have the parties run a protocol for what we call multi-sender $k$-out-of-$n$ oblivious transfer. This $m$-party functionality enables a receiver $P_m$ to obtain $k$-out-of-$n$ strings from $m-1$ different senders $P_1, \ldots, P_{m-1}$ (each with a set of $n$ strings). The functionality is formally defined in Figure 4.1.

**Figure 4.1 (Multi-Sender $k$-out-of-$n$ Oblivious Transfer $\mathcal{F}_n^k$)**

- **Inputs:**
  1. For every $j = 1, \ldots, m-1$, party $P_j$ inputs a vector of $n$ strings $(x_1^j, \ldots, x_n^j)$.
  2. The receiver $P_m$ inputs a set of indices $I \subset [n]$ of size exactly $k$.
- **Outputs:**
  1. If $|I| \neq k$ then all parties receive $\perp$ as output.
  2. Otherwise, for every $j = 1, \ldots, m-1$, party $P_m$ receives the set $\{(i, x_i^j)\}_{i \in I}$ of $k$ strings (and nothing else).
  3. Parties $P_1, \ldots, P_{m-1}$ receive no output.

We stress that that the receiver is forced to use the same index set $I$ for all sender parties. In the context of the IPS compiler, this means that each party can choose $k$ servers for which it watches *all* parties. Now, let $t$ be the number of corrupted real parties in the protocol. Then, the parties can together view $t \cdot k$ servers, which is equivalent to these servers being corrupted. In addition, the probability that the corrupted real parties can cheat in the semi-honest subprotocols for more than $L$ servers equals the probability that none of the honest parties have any of these servers in their watchlists. A single honest party chooses $k$ out of $n$ servers in its watchlist and so the probability that it does *not* detect such cheating equals $\binom{n-L}{k} / \binom{n}{k}$.

Let $m = 2$ and let $n = 4k$. We have that the corrupted real party views $k$ of the servers in its watchlist and needs to cheat in $L = k$ more in order to have corrupted at least half of the servers. The probability that it can do this without being detected is therefore

$$\frac{\binom{3k}{k}}{\binom{4k}{k}} = \frac{(3k)!}{(2k)! \cdot k!} \cdot \frac{(3k)! \cdot k!}{(4k)!} = \frac{(3k)!}{(2k)!} \cdot \frac{(3k)!}{(4k)!} = \frac{3k}{4k} \cdot \frac{3k-1}{4k-1} \cdots \frac{2k+1}{3k+1} < \left(\frac{3}{4}\right)^k .$$

Now consider the general case of $m$ real parties and $n = 4mk$. We first analyze the case that $m-1$ parties are corrupted. Then, the adversary can view $(m-1)k$ different servers, and needs

to corrupt $(m+1)k$ additional servers in order to have corrupted half of the servers. Thus, the probability that it goes undetected is

$$\frac{\binom{3mk}{k}}{\binom{4mk}{k}} = \frac{(3mk)!}{(3mk-k)!} \cdot \frac{(4mk-k)!}{(4mk)!} = \frac{3mk}{4mk} \cdot \frac{3mk-1}{4mk-1} \cdots \frac{3mk-k+1}{4mk-k+1} < \left(\frac{3}{4}\right)^k .$$

Thus, we conclude that it suffices to use $n = 4mk$ virtual servers rather than $O(m^2k)$. In addition, as is evident, the proof of this fact is straightforward. (We stress that as shown in Section 6, this is not necessarily the best way to choose the parameters for concrete efficiency. However, here we are dealing with asymptotic efficiency.)

We conclude with the following somewhat informal claim; it's proof follows from the analysis above and the proof of security of [27].

**Claim 4.2** *The IPS compiler with the alternative watchlist setup phase using the multi-sender $k$-out-of-$n$ oblivious transfer functionality is secure with $n = O(mk)$ servers.*

### 4.2 A Secure Protocol Realizing the Multi-Sender $k$-out-of-$n$ OT Functionality

**A general protocol from oblivious transfer.** It is possible to securely realize the multi-sender $k$-out-of-$n$ oblivious transfer functionality in a straightforward way using committed oblivious transfer, which in turn can be constructed in a black-box way from any 1-out-of-2 oblivious transfer [9]. This construction has the advantage of preserving the IPS general structure of working under general assumptions and using any oblivious transfer protocol that is secure in the presence of malicious adversaries. Thus, we obtain the efficiency improvement regarding the number of servers and the simpler analysis, while remaining within the same framework. We note, however, that this strategy will not yield a concretely efficient watchlist setup.

**A concrete protocol with greater efficiency.** One of the aims in this paper, which is dealt with in detail in Section 6, is to consider the concrete efficiency of the IPS compiler. As such, in this section we present a highly-efficient protocol for securely computing the multi-sender $k$-out-of-$n$ oblivious transfer functionality in the presence of malicious adversaries. The security of the protocol is based on the DDH assumption and requires just $O(n)$ exponentiations. Below, we compare the concrete efficiency of our watchlist setup method based on this protocol to the best-known concrete instantiations of the original IPS watchlist setup, and show that the efficiency improvement is dramatic. As we will see, the use of this protocol enables the use of significantly more servers than otherwise (in Section 6 it will become apparent why this is so important).

Our protocol uses ideas from the cut-and-choose oblivious transfer protocol of [29]. The idea is for the receiver $P_m$ to choose $n$ pairs of group elements $(a_i, b_i)$ so that relative to two fixed group elements $g, h$ it holds that at most $k$ out of the $n$ tuples $(g, h, a_i, b_i)$ are Diffie-Hellman tuples (and all the others are non-DH tuples). The receiver then broadcasts all of these pairs to the sending parties $P_1, \ldots, P_{m-1}$. Following this, all of the sending parties send values with the property that $P_m$ can obtain the $i$th string if and only if $(g, h, a_i, b_i)$ is a DH tuple. This ensures that $P_m$ receives $k$-out-of-$n$ of the strings and no more. Furthermore, since the pairs are broadcast to all $P_1, \ldots, P_{m-1}$, it is guaranteed that $P_m$ receives the $i$th string from every $P_1, \ldots, P_{m-1}$ if and

**Protocol 4.3 (Multi-Sender $k$-out-of-$n$ Oblivious Transfer)**

- **Inputs:** For every $j = 1, \ldots, n-1$, party $P_j$ inputs a vector of $n$ strings $(x_1^j, \ldots, x_n^j)$. The receiver $P_n$ inputs a set of indices $I \subset [n]$ of size exactly $k$.
- **Auxiliary input:** All parties hold $(\mathbb{G}, q, g)$, where $\mathbb{G}$ is an efficient representation of a group of order $q$ with a generator $g$.

1. **Step 1 – choose and broadcast pairs:**
   (a) $P_m$ chooses a random $y \leftarrow \mathbb{Z}_q$ and sets $h = g^y$.
   (b) For every $i \in I$, $P_m$ chooses a random $\alpha_i \leftarrow \mathbb{Z}_q$ and computes $a_i = g^{\alpha_i}$ and $b_i = h^{\alpha_i}$.
   (c) For every $i \notin I$, $P_m$ chooses a random $\alpha_i \leftarrow \mathbb{Z}_q$ and computes $a_i = g^{\alpha_i}, b_i = h^{\alpha_i+1}$.
   (d) $P_n$ *broadcasts* $(h, a_1, b_1, \ldots, a_n, b_n)$ to $P_1, \ldots, P_{m-1}$.
2. **Step 2 – prove valid choice of pairs:** For every $j = 1, \ldots, m-1$, $P_m$ proves to $P_j$ using a zero-knowledge proof of knowledge that at least $n-k$ of the tuples $(g, h, a_i, \frac{b_i}{h})$ are DH tuples. ($P_m$ must actually prove that at most $k$ of the tuples $(g, h, a_i, b_i)$ *are* DH tuples which is equivalent to proving that at least $n-k$ of them are *not*. This in turn is equivalent to proving that at least $n-k$ of the tuples $(g, h, a_i, \frac{b_i}{h})$ *are* DH tuples. This can be proven at the cost of just $7n$ exponentiations; see [29].)
   If $P_j$ rejects the proof then it broadcasts $\perp$ to all other parties and halts. If any party receives $\perp$, then it halts.
3. **Step 3 – each party $P_j$ sends values:** Define the function $RAND(w, x, y, z) = (u, v)$, where $u = (w)^s \cdot (y)^t$ and $v = (x)^s \cdot (z)^t$, and the values $s, t \leftarrow \mathbb{Z}_q$ are random. Then, for every $i = 1, \ldots, n$, each party $P_j$ computes $(u_i^j, v_i^j) = RAND(g, a_i, h, b_i)$, and sends $P_n$ the values $\{(u_i^j, w_i^j)\}_{i=1}^n$ where $w_i^j = v_i^j \cdot x_i^j$.
4. **Step 4 – compute output:** For every $j = 1, \ldots, m-1$ and every $i \in I$, party $P_m$ computes and outputs $x_i^j = w_i^j / (u_i^j)^{\alpha_i}$.

only if $(g, h, a_i, b_i)$ is a DH tuple. Thus, intuitively, the protocol securely computes the multi-sender $k$-out-of-$n$ oblivious transfer functionality. See Protocol 4.3 for a full description.

Before proving security, observe that if the parties behave honestly then

$$\frac{w_i^j}{(u_i^j)^{\alpha_i}} = \frac{v_i^j \cdot x_i^j}{(u_i^j)^{\alpha_i}} = \frac{(a_i)^s \cdot (b_i)^t}{(g^s \cdot h^t)^{\alpha_i}} \cdot x_i^j = \frac{(a_i)^s \cdot (b_i)^t}{(g^{\alpha_i})^s \cdot (h^{\alpha_i})^t} \cdot x_i^j = \frac{(a_i)^s \cdot (b_i)^t}{(a_i)^s \cdot (b_i)^t} \cdot x_i^j = x_i^j,$$

as required, and therefore $P_m$ obtains the correct output. We now prove security.

**Proposition 4.4** *Assume that the Decisional Diffie-Hellman assumption holds in the group $(\mathbb{G}, q, g)$. Then, Protocol 4.3 securely computes the multi-sender k-out-of-n oblivious transfer functionality in the presence of a malicious adversary controlling any number of parties.*

**Proof Sketch:** The intuition behind the security of the protocol appears above; we therefore proceed directly to the proof. First, consider the case that $P_m$ is corrupted and all $P_1, \ldots, P_{m-1}$ are honest. In this case, the simulator can learn the index set $I$ for which the tuples $(g, h, a_i, b_i)$ are of the Diffie-Hellman type by extracting the witness from the zero-knowledge proof of knowledge protocol. The simulator then sends $I$ to the trusted party and receives back the strings $\{x_i^j\}_{i \in I; 1 \leq j < m}$. For every $i \in I$, the simulator computes the pair $(u_i^j, w_i^j)$ exactly like an honest sender; in contrast

for every $\ell \notin I$ the simulator sets $u_\ell^j, w_\ell^j$ to be random group elements. The distribution of these messages is identical to a real protocol execution (when considering an ideal zero-knowledge proof of knowledge functionality) since for every $\ell \notin I$ the tuple $(g, h, a_\ell, b_\ell)$ is not a Diffie-Hellman tuple. By the property of the RAND function, see [32], this implies that the values $u_\ell^j, v_\ell^j$ are independent random elements of $\mathbb{G}$. Next observe that when $P_m$ is corrupted and some other parties $P_j$ $(j < m)$ are also corrupted, the same simulation strategy works. This is due to the fact that every sender behaves independently. Finally, consider the case that $P_m$ is honest and some of the sending parties $P_j$ are corrupted. In this case, the simulator chooses all of the $(a_i, b_i)$ pairs so that $(g, h, a_i, b_i)$ are Diffie-Hellman tuples and then "cheats" in the zero-knowledge proof by running the simulator. As a result, the simulator receives all of the strings $(x_1^j, \ldots, x_n^j)$ from every corrupted $P_j$ and can send them to the trusted party computing the functionality. The view of the corrupted parties $P_j$ is indistinguishable by the Decisional Diffie-Hellman assumption, and the joint output of the honest parties and the corrupted parties' views are indistinguishable since the distribution over the values sent by the corrupted parties is also computationally indistinguishable. ■

**Concrete efficiency.** The cost of the protocol is as follows: $P_m$ computes $4n+1$ exponentiations in the first step. Then, the cost of the zero-knowledge proof in the second step is $7(m-1)n$ exponentiations overall (recall that the proof is run $m-1$ times with every $P_j$ $(j < m)$ playing the verifier). In the third step, each $P_j$ computes $4n$ exponentiations (with an overall of $4(m-1)n$), and finally, $P_m$ computes $k(m-1)$ exponentiations in order to obtain the output. We have that the number of overall exponentiations is $4n + 7(m-1)n + 4(m-1)n + k(m-1)$. (For large $m$, this is approximately $11mn + km$ exponentiations. For the special case of $m = 2$ this comes to $15n + k$.)

**A comparison of concrete efficiency.** We now compare the concrete cost of running our watchlist setup protocol to the method of [27]. Recall that the IPS setup requires $m(m-1)n \log n$ oblivious transfers and this can be implemented using [32] at the cost of $11 \cdot m(m-1)n \log n$ exponentiations, or using the method of extending oblivious transfers of [26]. In contrast, our protocol requires $11mn + mk$ exponentiations. (All exponentiations here are in any group in which the DDH assumption holds.)

For the sake of comparison, assume that the same level of security is obtained using both setups and so the same values of $k$ and $n$ can be used.[2] We compare this for two sets of concrete parameters given in Section 6.3, optimizing the performance of the inner protocol for the case of $m = 2$ parties. For the AES-type circuit, we have $k = 207$ and $n = 1752$. Then, the cost of the original IPS setup is $m(m-1)n \log n \approx 37,754$ oblivious transfers. This can be implemented using [32] at a cost of 11 exponentiations per oblivious transfer, resulting in $415,294$ exponentiations. Alternatively, using the method of extending oblivious transfers of [26], the cost is about $5,632$ oblivious transfers (requiring about $62,000$ exponentiations) and approximately $4,153,000$ hash function computations.[3] In contrast, our new setup costs $15n + k = 15 \cdot 1752 + 207 = 26,487$ exponentiations, which is much less (even than the solution using [26]). An even more illustrative example relates to the two settings of parameters given for the case of a circuit of size $30,000$ in

[2] This is clearly not the case. First, the analysis of [27] requires $O(m^2 k)$ servers whereas for us it suffices to use $O(mk)$ servers. Second, the error of [27] must take into account the fact that with some probability the size of the watchlists may deviate from $k$ (and so the honest parties may have watchlists that are too small to catch the malicious, and the malicious may have watchlists that are too big thereby breaking the outer protocol). This can be bound without too much difficulty, but when taking concrete values, this will have a noticeable influence.

[3] We calculated this based on a seed length of 128 bits, and using $\sigma$ of size 44 in order to obtain an error of $2^{-40}$ in the oblivious transfer extension protocol. For these parameters, $44 \times 128 = 5,632$ actual oblivious transfers are run, and then an additional 110 hash computations are carried out per oblivious transfer; see [26, Figure 2].

Section 6.3; for this we just directly use the extending oblivious transfer alternative. With the first choice of parameters, $n = 19554$ and $k = 729$, we have that the IPS setup costs $62,000$ exponentiations and $61,324,000$ hash operations, versus $294,039$ exponentiations for our protocol. In addition, the bandwidth required to run an OT extension of this size would be huge. The other choice of parameters, of $n = 3362$ and $k = 292$, requires $62,000$ exponentiations and $8,664,000$ hash operations, versus $50,772$ exponentiations for our protocol. Thus, using our setup protocol, it is still feasible to choose either of the optimal choices of parameters and tradeoff the cost for the rest of the protocol. In contrast, using the IPS setup, only the latter setting of $k$ and $n$ can be reasonably used.

**Significance.** As will be demonstrated in Section 6, the ability to work with a large number of servers yields a much more efficient outer protocol and reduces the number of semi-honest subprotocols that need to be run when emulating the outer protocol. In the example parameters shown above, the cost of the watchlist setup phase is *prohibitive* for the original IPS solution and so this choice of parameters will not be optimal. In contrast, the cost of running our new setup protocol for the same number of servers is reasonable and so such parameters can be chosen. We stress that for larger values of $m$ (i.e., in multiparty computation settings), the efficiency gain will be even more dramatic.

### 4.3 Flexibility of the Outer Protocol

In this section, we describe an additional optimization for the IPS compiler. In the original analysis carried out by [27] and that discussed above, the outer protocol chosen is secure in the presence of a malicious adversary that can adaptively corrupt up to $t$ servers, for an appropriately chosen $t$. However, the corruptions of the servers are actually of two distinct types. The up to $(m-1)k$ corruptions that are due to the fact that the watchlists that the adversary has of the honest parties, are actually *semi-honest* corruptions, meaning that the adversary sees the internal state of these servers but does not cause them to deviate from the protocol specification. In contrast, the corruptions that are due to the corrupted real parties cheating in the semi-honest server simulation (without being caught) are *malicious* corruptions. Thus, it is possible to use an outer protocol that provides hybrid security in the presence of $t_1$ malicious corruptions and $t_2$ semi-honest corruptions. This model has been studied, and it has been demonstrated that better resilience can be achieved [13].

We conclude that it suffices to use an outer protocol with hybrid security of the flavor of [13], providing more flexibility in choosing parameters with the aim of using a more efficient outer protocol. A concrete example of where this can be utilized is in the simple and efficient multiplication protocol of BGW [4] for the case of $t < n/4$ malicious corruptions. This multiplication protocol can also be used in the case of $t_1 < n/6$ malicious corruptions together with $t_2 < n/6$ semi-honest corruptions. Thus, although the overall number of corruptions is $t < n/3$, the simpler and more efficient multiplication protocol can be used. This can be heavily utilized in the setting of IPS compilation.

# 5 IPS Variants Using Covert Adversaries

In this section we present variants of the IPS compiler in order to obtain security in the presence of malicious adversaries from security in the presence of covert adversaries, and security in the presence of covert adversaries from security in the presence of semi-honest adversaries. In addition, we show that this approach has a quantitative advantage regarding the black-box construction of malicious oblivious transfer from semi-honest oblivious transfer.

## 5.1 Secure Computation for Malicious from Covert Adversaries

In this section we show an extraordinary simple analog of the IPS compiler when the starting point is a protocol for secure computation in the presence of covert adversaries (instead of a protocol for semi-honest adversaries). As we have already discussed the idea behind our construction in the Introduction, we proceed directly to the construction.

### 5.1.1 The Protocol in the Hybrid Model

We construct a protocol for computing a function $f$ for $m$ parties, where any number can be corrupt. The protocol is secure against malicious adversaries. The security parameter is denoted by $k$.

**Tools:**

- Let $\pi$ be a multiparty (outer) protocol for $m$ clients and $n = 2k$ servers, which is secure for any number of corrupted clients and as long as less than $k$ servers are corrupted by an adaptive malicious adversary. For simplicity, the protocol $\pi$ is such that all messages are sent over a broadcast channel. In addition, every party broadcasts in every round (if $\pi$ does not instruct a party to broadcast in some round, then it sends an empty message $\lambda$). $\pi$ computes the function $f$ where parties $P_1, \ldots, P_m$ provide input and receive output; all other parties $P_{m+1}, \ldots, P_{m+n}$ have no input or output. The parties $P_1, \ldots, P_m$ are called *clients*, and the parties $P_{m+1}, \ldots, P_{m+n}$ are called *servers*.

- Let $\pi_1, \ldots, \pi_{m+n}$ be the instructions for the parties in $\pi$. That is, the clients $P_1, \ldots, P_m$ run $\pi_1, \ldots, \pi_m$ and the $i$th server runs $\pi_{m+i}$.

  For the servers, namely for $i = 1, \ldots, n$, let $\mathcal{F}_{m+i}$ be the *reactive* ideal functionality computing $\pi_{m+i}$. Loosely speaking, $\mathcal{F}_{m+i}$ is a functionality that receives $m+n-1$ inputs in each round and generates a single output; the $m+n-1$ inputs are the values broadcast by all parties $P_j$ for $j \neq i$ in the previous round and the output is the value that $P_i$ should broadcast in this round. The exact description of the functionality $\mathcal{F}_{m+i}$ is more involved. This functionality is actually run by the $m$ real parties. Thus, each party inputs a vector of length $m+n$ with the values broadcast in the previous round. The functionality then verifies that *all* vectors input by the $m$ clients are identical. If not, it outputs $\perp$. If yes, it computes the next message that $P_{m+i}$ would send and hands it to all the $m$ clients.

- We denote by $\mathcal{F}^{\epsilon}_{m+i}$ the functionality $\mathcal{F}_{m+i}$ in the covert model with deterrent $\epsilon$. This means that we consider an ideal functionality that computes $\mathcal{F}_{m+i}$ with the additional instructions of the trusted party of the ideal model of the definition of covert adversaries; see Section A.2.

**Protocol 5.1 (Secure Computation for Malicious from Covert in the hybrid model)**

- **Inputs:** Real parties $P_1, \ldots, P_m$ hold respective inputs $x_1, \ldots, x_m$
- **The protocol:** For every round of protocol $\pi$, the parties $P_1, \ldots, P_m$ work as follows:
  1. Each party $P_j$ $(1 \leq j \leq m)$ broadcasts the message that $\pi$ instructs the client $P_j$ to send in this round (using $\pi_j$), based on the messages from the last round.
  2. For every $i = 1, \ldots, n$, each party $P_j$ $(1 \leq j \leq m)$ sends $\mathcal{F}^{\epsilon}_{m+i}$ the vector of all messages broadcast in the previous round. (In the first round, the vector contains $m + n$ empty values $\lambda$.)
  3. For every $i = 1, \ldots, n$, each party $P_j$ $(1 \leq j \leq m)$ receives an output from $\mathcal{F}^{\epsilon}_{m+i}$. If the output is $\mathsf{corrupted}_\ell$ or $\mathsf{abort}_\ell$ (see [1]), then $P_j$ halts and outputs $\mathsf{abort}_\ell$. Otherwise, it records the output as the message "broadcast" by server $P_{m+i}$ in this round.
- **Output:** Each party $P_j$ $(1 \leq j \leq m)$ outputs the value that $\pi$ instructs client $P_j$ to output.

Protocol 5.1 uses these tools to obtain security in the presence of a malicious adversary controlling an arbitrary number of the $m$ parties. The protocol is defined in a hybrid model where the functionalities $\mathcal{F}^{\epsilon}_{m+1}, \ldots, \mathcal{F}^{\epsilon}_{m+n}$ are executed by a trusted party. Below, we derive security when these functionalities are instantiated by real protocols.

We now state the security of Protocol 5.1. It is very straightforward, and this highlights the conceptual advantage of this alternative IPS compiler.

**Theorem 5.2** *Let $\pi$ be a protocol for $m$ clients and $n = 2k$ servers that securely computes the $m$-party functionality $f$ with abort, in the presence of an adaptive malicious adversary corrupting any number of clients and less than $k$ of the servers, and let $\epsilon > 0$ be any constant. Then, Protocol 5.1 securely computes $f$ with abort in the $\mathcal{F}^{\epsilon}_{m+1}, \ldots, \mathcal{F}^{\epsilon}_{m+n}$ hybrid model, in the presence of an adaptive malicious adversary corrupting any number of corrupted parties.*

**Proof:** The intuition behind the proof has been described above. On the one hand, if an adversary attempts to cheat in at least $k$ of the $\mathcal{F}^{\epsilon}_{m+i}$ executions then it will be caught except with probability $(1 - \epsilon)^k$; for constant $\epsilon$ this is negligible. On the other hand, if an adversary attempts to cheat in less than $k$ of the $\mathcal{F}^{\epsilon}_{m+i}$ executions, then $\pi$ will maintain security because less than $k = n/2$ servers were corrupted. The formal proof works by showing how to simulate the real execution in this setting.

Let $\mathcal{A}$ be an adaptive adversary for Protocol 5.1. We begin by constructing an adaptive adversary $\mathcal{A}_\pi$ for $\pi$ that corrupts less than $k$ servers and any number of clients, such that

$$\left\{ \text{REAL}_{\pi, \mathcal{A}_\pi}(\overline{x}) \right\} \stackrel{\text{s}}{\equiv} \left\{ \text{HYBRID}^{\mathcal{F}^{\epsilon}_{m+i}}_{\Pi, \mathcal{A}}(\overline{x}) \right\} \tag{1}$$

where $\Pi$ denotes Protocol 5.1. The adversary $\mathcal{A}_\pi$ externally runs protocol $\pi$ while internally emulating an execution of Protocol 5.1 for $\mathcal{A}$, as follows:

1. Whenever $\mathcal{A}$ corrupts a party from the set $\{P_1, \ldots, P_m\}$ of clients, then $\mathcal{A}_\pi$ corrupts that party.
2. Whenever $\mathcal{A}$ broadcasts a message from some corrupted $P_j$ with $j \in \{1, \ldots, m\}$, adversary $\mathcal{A}_\pi$ broadcasts the same message to all parties $P_1, \ldots, P_{m+n}$.

3. Whenever $\mathcal{A}$ participates in an execution of $\mathcal{F}_{m+i}^{\epsilon}$, adversary $\mathcal{A}_\pi$ acts as follows:
   (a) If $\mathcal{A}$ participates in the execution, and does not send abort, corrupted or cheat as input to the trusted party (and so the honest parties receive output), then $\mathcal{A}_\pi$ sets the output of $\mathcal{F}_{m+i}^{\epsilon}$ for $\mathcal{A}$ to be the real message broadcast by $P_{m+i}$ in this round of $\pi$.
   (b) If $\mathcal{A}$ participates in the execution and sends corrupted or abort to the trusted party, then $\mathcal{A}_\pi$ broadcasts abort to all honest parties in $\pi$ and halts.
   (c) If $\mathcal{A}$ sends $\mathsf{cheat}_{m+i}$ for some corrupted $P_{m+i}$ for $i \in \{1, \ldots, n\}$ to the trusted party, then with probability $\epsilon$ adversary $\mathcal{A}_\pi$ broadcasts abort to all honest parties and halts. In contrast, with probability $1 - \epsilon$ (modeling the case that the cheat was undetected), adversary $\mathcal{A}_\pi$ corrupts party $P_{m+i}$ in the execution of $\pi$. Then, it internally sends undetected to $\mathcal{A}$ and receives back the output that $\mathcal{A}$ determines in this execution and the internal state of the functionality. $\mathcal{A}_\pi$ then sends this output to all honest parties in $\pi$ as the message from $P_{m+i}$ in this round, and sets the state of $P_i$ to be as sent by $\mathcal{A}$. From here on, $\mathcal{A}_\pi$ runs $P_{m+i}$ from this given state. (We note that if $\mathcal{A}$ sends $\mathsf{cheat}_{m+i}$ again for $\mathcal{F}_i^{\epsilon}$ then it works in exactly the same way, except that it does not need to corrupt $P_{m+i}$ again.)
   The above instructions are followed unless undetected occurs $k$ times. In this case, $\mathcal{A}_\pi$ does not corrupt any party, and just halts with output fail instead.
4. At the conclusion of the execution of $\pi$, adversary $\mathcal{A}_\pi$ outputs whatever $\mathcal{A}$ outputs.

Let fail be the event that in $k$ of the times that $\mathcal{A}$ sends $\mathsf{cheat}_{m+i}$ to the trusted party (in the real execution of $\pi$ with $\mathcal{A}_\pi$ or in the execution of Protocol 5.1 with $\mathcal{A}$), the result is undetected. We claim that

$$\left\{\text{REAL}_{\pi,\mathcal{A}_\pi}(\overline{x}) \mid \neg\mathsf{fail}\right\} \equiv \left\{\text{HYBRID}_{\Pi,\mathcal{A}}^{\mathcal{F}_{m+i}^{\epsilon}}(\overline{x}) \mid \neg\mathsf{fail}\right\} \ . \tag{2}$$

This follows almost immediately from the fact that each $\mathcal{F}_{m+i}^{\epsilon}$ runs the instructions of party $P_{m+i}$ in $\pi$ exactly. Thus, the only deviation is in the case that $\mathcal{A}$ sends cheat in an $\mathcal{F}_{m+i}^{\epsilon}$ execution and the result is undetected (in the case that the cheating is detected, then abort is received by all in both cases). In this case of undetected, in Protocol 5.1 adversary $\mathcal{A}$ can fully determine the message sent by $P_{m+i}$. However, this exact same behavior is achieved in the execution with $\mathcal{A}_\pi$ by having $\mathcal{A}_\pi$ corrupt $P_{m+i}$ and send the message determined by $\mathcal{A}$. Thus, the output distributions are identical.

The only difference that occurs is therefore in the case that $k$ or more cheat attempts are made and all are undetected (i.e., when the fail event occurs). The reason that this causes a discrepancy is that $\mathcal{A}_\pi$ is only able to corrupt less than $k$ parties when running $\pi$, and it needs to corrupt a new party every time undetected is obtained. Nevertheless, when conditioning on this event not happening, we have that the executions are identical.

Combining (2) with the fact that Pr[fail] equals *exactly* $\epsilon^k$ in both the real execution of $\pi$ with $\mathcal{A}_\pi$ and in the execution of Protocol 5.1 with $\mathcal{A}$, and the fact that $\epsilon^k = \epsilon^{n/2}$ is negligible in $n$, we obtain that (1) holds.

Now, by the security of $\pi$ and the fact that $\mathcal{A}_\pi$ corrupts at most $m$ clients and less than $k$ servers, we have that there exists a simulator $\mathcal{S}_\pi$ such that

$$\left\{\text{IDEAL}_{f,\mathcal{S}_\pi}(\overline{x})\right\} \stackrel{\text{c}}{\equiv} \left\{\text{REAL}_{\pi,\mathcal{A}_\pi}(\overline{x})\right\}. \tag{3}$$

We are now ready to construct our simulator $\mathcal{S}$ for Protocol 5.1. Simulator $\mathcal{S}$ works as follows:

1. $\mathcal{S}$ defines the adversary $\mathcal{A}_\pi$ based on $\mathcal{A}$, as described above. (This can be done even if $\mathcal{S}$ is given only black-box access to $\mathcal{A}$ because $\mathcal{A}_\pi$ is just a "wrapper" for $\mathcal{A}$.)

2. $\mathcal{S}$ runs the simulator $\mathcal{S}_\pi$ on adversary $\mathcal{A}_\pi$.

3. When $\mathcal{S}_\pi$ sends inputs to the trusted party for corrupted parties amongst the clients $P_1, \ldots, P_m$, simulator $\mathcal{S}$ sends these same inputs to its trusted party. (Recall that by the protocol definition only parties $P_1, \ldots, P_m$ have inputs or outputs.)

4. When $\mathcal{S}$ receives back outputs for corrupted parties amongst $P_1, \ldots, P_m$, it hands these outputs to $\mathcal{S}_\pi$.

5. $\mathcal{S}$ outputs whatever $\mathcal{S}_\pi$ outputs.

It is immediate that the output distribution of $\mathcal{S}$ and $\mathcal{S}_\pi$ is identical; that is:

$$\left\{\text{IDEAL}_{f,\mathcal{S}}(\overline{x})\right\} \equiv \left\{\text{IDEAL}_{f,\mathcal{S}_\pi}(\overline{x})\right\}. \tag{4}$$

Combining Equations (1), (3) and (4), we have that

$$\left\{\text{IDEAL}_{f,\mathcal{S}}(\overline{x})\right\} \stackrel{\text{c}}{\equiv} \left\{\text{HYBRID}^{\mathcal{F}^\epsilon_{m+i}}_{\Pi,\mathcal{A}}(\overline{x})\right\},$$

completing the proof that Protocol 5.1 securely computes $f$. ■

**The protocol in the real model.** The above description and analysis refer to a hybrid model where a trusted party is used to compute all of the $\mathcal{F}^\epsilon_{m+i}$ functionalities. In order to achieve security in the presence of static adversaries in the stand-alone model, it suffices to securely compute these functionalities in the presence of covert (static) adversaries. However, since these functionalities are called in parallel (note that because the functionalities maintain a state between invocations, calling them sequentially is still not "sequential composition" as in the sense of [5]), we require that the protocols securely computing them be secure under parallel composition. Thus, applying the sequential composition theorem of [5], as modified to the covert setting in [1], we have:

**Corollary 5.3** *Let $\pi$ be a protocol for $m$ clients and $n$ servers that securely computes the $m$-party functionality $f$ with abort, in the presence of an adaptive malicious adversary corrupting any number of clients and a minority of the servers. Furthermore, let $\pi_{m+1}, \ldots, \pi_{m+n}$ be $m$-party protocols that securely compute $\mathcal{F}^\epsilon_{m+1}, \ldots, \mathcal{F}^\epsilon_{m+n}$ for some constant $\epsilon > 0$, and maintain security in the presence of a static adversary when run in parallel. Then, the real protocol derived by running Protocol 5.1 with $\pi_{m+1}, \ldots, \pi_{m+n}$ securely computes $f$ in the presence of a static malicious adversary corrupting any number of corrupted parties.*

Constructing protocols $\pi_{m+1}, \ldots, \pi_{m+n}$ to be secure under parallel composition can yield additional complexity. We remark that if we redefine the functionalities $\mathcal{F}^\epsilon_{m+i}$ so that instead of maintaining state between executions, the parties share an authenticated state (so that it cannot be modified by a malicious party), then the requirement for parallelism is no longer needed. If $m = 2$ then the protocols $\pi_{m+1}, \ldots, \pi_{m+n}$ can be implemented as is described in [1]. If $m > 2$, then these protocols can be constructed using the efficient multiparty constructions, secure in the presence of covert adversaries, appearing in [20].

**Adaptive security and universal composability.** Corollary 5.3 refers to static adversaries and the stand-alone model. In order to obtain adaptive security, the protocols $\pi_{m+1}, \ldots, \pi_{m+n}$ need to be adaptively secure. Likewise, if $\pi$ and $\pi_{m+1}, \ldots, \pi_{m+n}$ are all universally composable, then the result is also universally composable.

## 5.2 Secure Computation for Covert from Semi-Honest Adversaries

We describe a black-box transformation from semi-honest protocols to covert protocols, using a covert oblivious transfer protocol. This result answers an open question left by the work of [10], which showed a similar transformation in the information-theoretic setting with an honest majority, but did not cover the case of a majority of corrupt parties. The construction is very similar to the original construction of IPS [27] with two exceptions. First, only small watchlists are used. Second, it suffices for us to use an oblivious transfer protocol with security for covert adversaries, rather than security for malicious adversaries, in order to set up the watchlists. (Oblivious transfer protocols with security for covert adversaries were described in [1], based on homomorphic encryption. Alternatively, one can use the black-box construction of covert OT from $O(1)$ invocations of semi-honest OT described in Lemma 5.7 below.)

We construct a protocol for computing a function $f$ for $m$ parties, where any number of them can be corrupt. The security parameter is denoted by $k$. As in Protocol 5.1, we assume that all messages are broadcast in $\pi$ and there is no point-to-point communication (private point-to-point communication can be implemented using public-key encryption over the broadcast channel). This enables us to use only the first type of watchlist described in Section 2 since the inner protocols output the actual broadcast message to all clients, and not just shares of the messages sent. Thus, the parties all know the messages sent in the previous round and it suffices to use watchlists to just check that the semi-honest protocol instructions are faithfully followed.

**Tools:**

- Let $\pi$ be a multiparty protocol for $m$ clients and $n = 4m$ servers, which is secure for any number of corrupted clients and less than $n/2$ corrupted servers. As in Protocol 5.1, all messages of $\pi$ are sent over a broadcast channel and every party broadcasts in every round. Furthermore, the clients $P_1, \ldots, P_m$ are the only ones who provide input and receive output.
- Let $\pi_1, \ldots, \pi_{m+n}$ be the instructions for the parties in $\pi$, and let $\mathcal{F}_{m+i}$ be the *reactive* ideal functionality computing $\pi_{m+i}$, for $i = 1, \ldots, n$. The functionality is as defined for Protocol 5.1.
- Let $\rho_{m+1}, \ldots, \rho_{m+n}$ be protocols such that $\rho_{m+i}$ securely computes $\pi_{m+i}$ between $m$ parties in the presence of semi-honest adversaries. Without loss of generality we assume that the random-tape of each party in each $\rho_i$ is of length exactly $k$ (a pseudorandom generator can be used if it is longer).

The compiler is described in Protocol 5.4.

**Security.** The proof of security of this compiler is almost exactly the same as the original compiler of [27]. In order to see this, observe that Protocol 5.4 is exactly the same as that of [27] with two exceptions. First, we use a 1-out-of-$4m$ oblivious transfer and not a Rabin oblivious transfer (this difference is already discussed in Section 4.1). Second, we use oblivious transfer that is secure in

**Protocol 5.4 (Secure Multiparty Computation for Covert from Semi-Honest)**

- **Inputs:** Real parties $P_1, \ldots, P_m$ hold respective inputs $x_1, \ldots, x_m$
- **The protocol:**
  1. *Phase 1 – set up watchlists:*
     (a) For every $j = 1, \ldots, m$, party $P_j$ chooses a vector of $n = 4m$ random seeds $s_1^j, \ldots, s_{4m}^j \in \{0,1\}^n$. The parties all then run $m$ multi-sender 1-out-of-$4m$ oblivious transfers that are secure in the presence of covert adversaries, so that each party receives $P_j$ receives $\{s_{r_j}^i\}_{i=1}^m$ for some $r_j \in_R \{1, \ldots, n\}$.
     (b) At the conclusion of this phase, each client $P_j$ holds the following:
        i. A vector $\overline{s}_j = (s_1^j, \ldots, s_n^j)$ of random seeds
        ii. A set of strings $\{s_{r_j}^i\}_{i=1}^m$
  2. *Phase 2 – emulate $\pi$:* For every round of the $n = 4m$-party protocol $\pi$, the parties $P_1, \ldots, P_m$ work as follows:
     (a) Each party $P_j$ $(1 \leq j \leq m)$ broadcasts the message that $\pi$ instructs the client $P_j$ to send in this round, based on the messages from the last round.
     (b) For every $i = 1, \ldots, n$, each party $P_j$ $(1 \leq j \leq m)$ runs $\rho_{m+i}$ using the input as the vector of all messages broadcast in the previous round and random-tape $s_i^j$.
     (c) Each party $P_j$ checks its watchlists for the executions run in the previous step. Specifically, for every $\ell = 1, \ldots, m$ $(\ell \neq j)$, party $P_j$ verifies that party $P_\ell$ used the random tape $s_{r_j}^\ell$ for the computation of protocol $\rho_{m+r_j}$. If no, then $P_j$ outputs abort.
     (d) For every $i = 1, \ldots, n$, each party $P_j$ $(1 \leq j \leq m)$ receives from $\rho_{m+i}$ an output and records it as the message "broadcast" by $P_i$ in this round.
- **Output:** Each party $P_j$ $(1 \leq j \leq m)$ outputs the value that $\pi$ instructs client $P_j$ to output.

the presence of covert and not malicious adversaries. This is also of little consequence, because there exist modular composition theorems for the covert setting that are analogous to those of the malicious setting. Thus, all that remains is to observe that setting $n = 4m$ where $m$ is the number of clients, and using a protocol $\pi$ that is secure for any $t < n/2 = 2m$ corruptions, it suffices to use only a watchlist containing a *single* server and security is obtained in the presence of covert adversaries. Based on the concrete parameters we used above, we have that in order for the corrupted parties to cheat, they must cheat in more than $m$ of the emulated executions. In order to see this, observe that there are at most $m - 1$ corrupted clients. Furthermore, together they view at most $m - 1$ servers via their watchlists, because each party's watchlist has just one server. Thus, overall they have the effect of having "corrupted" $m - 1$ of the $4m$ servers in $\pi$. Thus, in order to cheat in $\pi$ they must cheat in over $m$ of the server emulations. In order to cheat without being detected, these incorrect server emulations must all go undetected, meaning that none of them took place on the honest party's watchlist. Now, the probability that a single incorrect emulation will be detected is at least $1/4m$ (at least one corrupted party must cheat, and then it will be caught if it cheats where the watchlist with the honest party is). Thus, the probability that none of the

incorrect emulations will be detected is strictly less than

$$\left(1-\frac{1}{4m}\right)^{m} = \left(\left(1-\frac{1}{4m}\right)^{4m}\right)^{\frac{1}{4}} < \left(\frac{1}{e}\right)^{\frac{1}{4}} \tag{5}$$

and so we obtain security with $\epsilon$-deterrent for $\epsilon > 1 - e^{-0.25} \approx 0.22$. We therefore conclude:

**Theorem 5.5** *Let $\pi$ be a protocol for $m$ clients and $n = 4m$ servers that securely computes the $m$-party functionality $f$ with abort, in the presence of an adaptive malicious adversary corrupting any number of clients and a minority of servers. Then, Protocol 5.4 securely computes $f$ in the $\mathcal{F}_{m+1}, \ldots, \mathcal{F}_{m+n}$ (semi-honest) hybrid model, where $n = 4m$, in the presence of an adaptive covert adversary corrupting any number of corrupted parties, with $\epsilon$-deterrence for $\epsilon > 1 - e^{-0.25}$.*

We have just taken one set of parameters, in order to demonstrate that one can set $k = O(m)$ here. It is possible to choose different values, in order to achieve whatever value of $\epsilon$ is desired. Recall also that Protocol 5.1 works well even for values of $\epsilon$ that are small. Thus, it is not necessary to increase the number of servers too much.

### 5.3 The Semi-Honest Cost of Malicious Oblivious Transfer

In this section, we present an application of our modified IPS compiler to the question of the *cost* of black-box constructions of oblivious transfer that is secure in the presence of malicious adversaries, from oblivious transfer that is secure only in the presence of semi-honest adversaries.

Currently, the only known black-box construction of malicious from semi-honest oblivious transfer is due to [25, 21]; see [22] for the combined result. The cost of this construction is $O(k^2)$ invocations of a semi-honest oblivious transfer for every malicious oblivious transfer for security parameter $k$. This count is obtained as follows. First, defensible oblivious transfer is constructed, at the cost of just a single semi-honest oblivious transfer.[4] Next, the defensible oblivious transfer is boosted so that defensible security is obtained if one of the parties is corrupted, and malicious security is obtained if the other party is corrupted. The cost of this transformation is $O(k)$ defensible oblivious transfers. Finally, fully secure oblivious transfer is obtained by transforming the intermediate construction, where the cost of this transformation is $O(k)$ of these intermediate oblivious transfers. Thus, the overall cost is $O(k^2)$ semi-honest oblivious transfers, where $k$ is the security parameter.

We stress that we do not count the cost of commitments, coin tossing and any other primitives that can be obtained from one-way functions. This makes sense because oblivious transfer is more "expensive", both in terms of the fact that it is a strictly stronger hardness assumption in the black-box world, and because it is typically also computationally more expensive. We stress also that the original IPS compiler cannot be used for this task since it requires malicious oblivious transfer to start with.

We now use our IPS-type compilers via covert adversaries in order to obtain a black-box construction of an oblivious transfer protocol that is secure in the presence of malicious adversaries, at *linear* rather than *quadratic* cost. That is, the construction uses only $O(k)$ semi-honest oblivious

[4] A defensible oblivious transfer provides privacy against adversaries which provide a good "defense". A defense is an input and random-tape that is provided by the adversary after the execution of the protocol. A defense is "good" if an honest party, given that input and random-tape, would have sent the same messages as those sent in the protocol.

transfers. This demonstrates that our optimized compiler has quantitative benefits from a theoretical perspective, as well as deepening our understanding of security in the presence of covert adversaries. We have the following theorem:

**Theorem 5.6** *There exists a black-box reduction from bit oblivious transfer that is secure in the presence of malicious adversaries to one-way functions and $O(k)$ invocations of bit oblivious transfer that is secure in the presence of semi-honest adversaries.*

**Proof:** The oblivious transfer functionality is computed between $m = 2$ parties using an arithmetic circuit of constant size (just compute $(1 - \sigma) \cdot x_0 + \sigma \cdot x_1$). We use Protocol 5.1 for computing this circuit, by running a protocol $\pi$ for 2 clients and $n = 2k$ servers computing the circuit implementing the oblivious transfer functionality. By [1], and since $\epsilon$ is constant, each $F^{\epsilon}_{m+i}$ used in Protocol 5.1 to implement a basic step of protocol $\pi$, can be securely computed in the presence of covert adversaries with $O(1)$ covert oblivious transfers (and one-way functions). In Lemma 5.7, we use the results of [22] to show that there exists a black-box reduction from oblivious transfer that is secure in the presence of covert adversaries to one-way functions and $O(1)$ invocations of oblivious transfer that is secure in the presence of semi-honest adversaries. Therefore, each $F^{\epsilon}_{m+i}$ can be computed using $O(1)$ semi-honest oblivious transfers. There are a constant number of multiplication gates, and each of these requires $O(k)$ invocations of $F^{\epsilon}_{m+i}$ protocols. We therefore conclude that the overall number of semi-honest oblivious transfers is $O(k)$, as required. ■

It remains to prove that there exists the aforementioned reduction from covert oblivious transfer to one-way functions and $O(1)$ semi-honest oblivious transfers.

**Lemma 5.7** *There exists a black-box reduction from oblivious transfer that is secure in the presence of covert adversaries to one-way functions and $O(1)$ invocations of oblivious transfer that is secure in the presence of semi-honest adversaries.*

**Proof:** It is possible to provide a direct proof of this fact. However, this would essentially be no more than reproving claims that appear in [22]. Specifically, it is shown in [21, 22] that there exists a black-box reduction from oblivious transfer that is secure in the presence of defensible adversaries to one-way functions and a single invocation of oblivious transfer that is secure in the presence of semi-honest adversaries. (Recall that a protocol is secure in the presence of defensible adversary if it is secure as long as no malicious adversary can simultaneously break privacy and provide a "defense" which is a retroactive proof that it behaved honestly.) Next, it is shown in [25, 22] that there exists a black-box reduction from oblivious transfer that is secure in the presence of malicious adversaries to one-way functions and $O(k^2)$ invocations of oblivious transfer that is secure in the presence of defensible adversaries (this is actually proven in two steps as we will see below). The transformations of [25, 22] are such that $O(k^2)$ invocations are used to obtain an error of $2^{-k}$, where "error" means cheating that is undetected. This construction works by having the parties run multiple defensible oblivious transfers on random inputs, and then uses cut-and-choose to have the parties prove that they behaved honestly. This cut-and-choose is a request by one party to the other to provide *defenses* for a subset of the oblivious transfers that were run. The proof of security demonstrates that the resulting protocol is simulatable, and thus secure in the presence of malicious adversaries, if the adversary can provide defenses for the unopened oblivious transfers. Stated differently, the protocol is simulatable unless the adversary is caught cheating, and thus is secure in the presence of covert adversaries even when fewer oblivious transfers are executed.

It can be verified in a straightforward way that two defensible transfers are needed to boost the security of the defensible oblivious transfer so that it provides security in the presence of a *covert receiver* with $\epsilon = 1/2$. (See [22, Section 4], and specifically the proof of Claim 4.3.) This can then be further boosted to provide security in the presence of a covert sender and receiver using two oblivious transfers of the previous type. Thus, overall four semi-honest oblivious transfers are required.[5] This completes the proof. ■

**Remark 1:** It is tempting to apply the notion of extending oblivious transfers [3, 26] in order to obtain a linear reduction, by first extending $O(k)$ semi-honest oblivious transfers to $O(k^2)$ semi-honest oblivious transfers and then applying [22] to obtain a single malicious oblivious transfer. However, this does not work since the construction of [3] is not black-box in the one-way function. Furthermore, the construction of [26] relies on the assumption of correlation-robust hash functions.

**Remark 2:** In the case of many oblivious transfers, the results of [27, 22] imply constant overhead. That is, they demonstrate that it is possible to obtain $N$ oblivious transfers that are secure in the presence of malicious adversaries given $N+\text{poly}(k)$ oblivious transfers that are secure in the presence of semi-honest adversaries. For an asymptotically large $N$, this implies constant overhead. Our above result relates to the different case of a single or few oblivious transfers. In such a case, we obtain a linear reduction, in contrast to the quadratic reduction of [22].

**Remark 3:** We stress that in writing $O(k)$ or $O(k^2)$, we consider protocols that achieve security except with probability $2^{-k}$. (Equivalently, we could write that the construction of [25, 21] requires $\omega(\log^2 k)$ semi-honest oblivious transfers, while our protocol requires $\omega(\log k)$ semi-honest oblivious transfers, and security would hold except with negligible probability, i.e., except with probability $k^{-\omega(1)}$.)

# 6 The Concrete Efficiency of IPS

In this section, we describe our analysis of the *concrete* efficiency of the best IPS-type protocols. Due to the high level of abstraction in the IPS construction, its concrete complexity was completely unknown.

The protocol that we examined is based on sharing values using block secret sharing as in [14], in which $\ell$ values are encoded in a single polynomial. Thus, given blocks $a = (a_1, ..., a_\ell), b = (b_1, ..., b_\ell)$ which are shared using two polynomials of degree $\delta$, addition results in a sharing of a polynomial of degree $\delta$ that hides the block $a + b = (a_1 + b_1, ..., a_\ell + b_\ell)$, while multiplication results in sharing a polynomial of degree $2\delta$ which hides the block $ab = (a_1 b_1, ..., a_\ell b_\ell)$; as usual, a protocol is used to reduce the degree of the polynomial to $\delta$.

In the two-party protocol, after sharing the inputs, one side (Bob) will provide shares for blinding the output of the gates, and the other side (Alice) will receive these blinded outputs. Alice will then recover the blinded results and form the inputs for the next layer (proving to Bob that

[5] It is possible to prove that two defensible – and thus semi-honest – oblivious transfers actually suffice. However, this is not of significance for this theoretical result, and we therefore did not see this as justifying reproving much of [22].

they were formed correctly). Bob will provide the un-blinding shares (proving to Alice that they were formed correctly). This will be done for each layer, until the final outputs are computed.

In the IPS setting the sharing of inputs among $n$ servers is simulated by additively sharing those shares between the two clients, and the operations on shares are simulated as described in [27]. Type I computations will be encountered during the proofs described below, these are computations for which one side can perform by itself and send the results on the watchlists for verification by the other side. Type II computations are ones that must involve both sides. Addition is easily simulated by local computations by each client, while multiplication involves a semi-honest inner protocol for its simulation, as described below.

Our in-depth analysis of the protocol, described in Appendix B, reveals that overall complexity of the protocol is dominated by the number of multiplications and the number of OTs (both the communication complexity and other computational operations are negligible in comparison to these). Our efficiency analysis will therefore present those two factors. The OTs are only needed for the inner semi-honest multiplication protocols,[6] and so the other building blocks will be analyzed only in terms of the number of multiplications. We emphasize that these OTs must only be secure against a semi-honest adversary, and not a malicious one.

### 6.1 An Analysis of the Building Blocks

**Secret sharing for blocks:** This secret sharing scheme is a variant of Shamir's secret sharing [35], presented in [14]. Each polynomial encodes a block of $\ell$ values. The cost of sharing $w$ elements among $n$ servers, using blocks of size $\ell$ and polynomials of degree $\delta$, is $(w/\ell)(\delta^2+n\delta)$ multiplications. Details are given in Appendix B.1.

**Proving that shares lie on $\delta$-degree polynomials:** After sharing the secrets it must be proved that the shares are indeed encoded by $z$ polynomials, each of degree $\delta$ (each polynomial is used to hide $\ell$ field elements; depending on the number of inputs, multiple polynomials must be used for the sharing). A protocol for such a proof is presented in [28]. It requires $\delta(z+n+k)$ mutliplications (details are in Appendix B.2).

**Proving some replication pattern of shared blocks:** The protocol requires parties to prove that certain shared blocks follow some replication pattern (namely that a certain output value is used as an input value for the next layer). The protocol we used is mentioned in the computation complexity analysis of [28]. See details in Appendix B.3. The cost is $(4\delta)^2+4\delta n+2((2\delta n+(2\delta)^2)u+(\delta n+\delta^2)v)+2(n+k)(u+v)$ mutliplications, where $u$ (resp. $v$) is the number of output (resp. input) blocks represented by $2\delta$-degree (resp. $\delta$-degree) polynomials.

**Semi-honest inner multiplication:** For multiplication gates the parties run a semi-honest protocol for the functionality $(x_1, x_2) \mapsto (x_1x_2 - r, r)$ for a random $r \in_R \mathcal{F}$. Six different protocols are presented for this functionality in [28]. We present the analysis for the most efficient protocols only (based on our concrete analysis for all options). The first protocol is based on packed Reed-Solomon

[6] We remark that the OTs needed for setting up the watchlists (which must be secure against malicious adversaries) are also a factor. However, they depend only on the number of servers $n$ and so can be considered at the end.

encoding and is black-box in the field [28], and the second protocol is due to Glboa [17] and makes nonblack-box usage of the field and assumes standard bit representation of elements.[7]

As is detailed in Appendix B.4, for a security parameter $s = 40$ giving error $2^{-40}$, the packed Reed-Solomon encoding protocol costs 2734 multiplications and 16 1-out-of-2 OTs per inner multiplication, while the protocol of [17] uses 40 multiplications and 40 1-out-of-2 OTs. Namely, one protocol is more efficient in terms of OTs and the other is more efficient in terms of multiplications. For concrete numbers this phenomenon might present implementers with a real dilemma.

## 6.2 Instantiating the parameters

In order to count concrete efficiency, the values of the different parameters must be set. We do not claim to have found the absolute optimal parameters, as the analysis of their effect on the overhead is very complex. We do present for each parameter the different considerations affecting the choice of its value, and eventually show that the protocol is comparable in its efficiency to other protocols from the literature and may be competitive in some settings.

The four main parameters that must be set are the degree of the polynomials $\delta$, the block size $\ell$, the number of corrupted parties tolerated $t$, and the number of servers $n$. Three out of the four different parameters, the degree, the block size and the corruption threshold, are tightly interconnected in that setting any two of them determines the third one. In addition, these three parameters are all chosen as a function of the number of servers $n$, and given their descriptions the actual concrete value of $n$ is chosen (independently of the circuit). As we will see below, the determination of the degree $\delta$ is a straightforward choice. We then determine the block size based on the actual circuit being computed, thereby essentially setting the threshold.

**The degree $\delta$:** Due to the replication proving protocol it must hold that $\delta < \frac{n}{4}$. This is due to the fact that we need to be able to reconstruct a polynomial which has degree $4\delta$ (this is because two multiplications are applied to the original polynomial; one from a multiplication dictated by the circuit itself, and another one coming from the protocol to prove replication patterns). Other than that, it seems that we should take the $\delta$ to be as big as we can, allowing us later to use a bigger block size. For simplicity we will take $\delta = \frac{n}{4}$, although in reality we need $\delta = \frac{n}{4} - 1$.

**The block size $\ell$:** The block size has to be strictly smaller than the degree $\delta$, but otherwise the larger the block size, the more efficient the outer protocol gets. This is because more multiplications are carried out together (note that there is no use in having a block size larger than the width of a layer in the circuit since this already upper bounds the number of multiplications that can be carried out together). However, the number of corrupted servers that the outer protocol can tolerate is $\delta - \ell$, thus the closer $\ell$ is to $\delta$, the smaller the fraction of corrupt servers that can be tolerated. As a result, more servers are required in order to ensure that the probability of catching the adversary cheating in the server simulation does not go down.

It is clear that the block size should not be bigger than the number of multiplication gates in a layer (this is because we work with blocks in the same layer and not across layers). Note that if the block size equals the number of multiplication gates in a layer, then we can multiply the whole layer at the cost of multiplying just the shares of one block. Below we present specific efficiency

[7] We do not present protocols based on homomorphic encryption since, even though they might be efficient, comparing them to the other protocols is more complex as they use completely different building blocks.

values for two concrete circuit parameters. We do not present a general optimum for every possible circuit structure.

**The corruption threshold $t$:** The number of the corrupt servers that we can tolerate is dictated by the chosen degree $\delta$ and block size $\ell$. As shown in [14], up to $t < \delta - \ell$ corrupt servers receive no information about the secret block when using block secret sharing with degree $\delta$ and block size $\ell$. Thus, $t$ is set to $\delta - \ell - 1$.

**The number of servers $n$:** We have already established that $n = O(mk)$ is enough in order for the adversary to be caught with an overwhelming probability if it tries to cheat in $t$ servers. Let us denote $\tau = \frac{n}{t}$, and so $\frac{1}{\tau}$ is the ratio of servers that the adversary needs to corrupt. In general we can say that $n = O(mk) = amk$ for some parameter $a$, and what we wish to analyze now is what is the best $a$ to choose. Because $a$ does not affect any other parameter than the number of servers, we conclude that we should choose the value that will minimize that number $n$. We have already seen in Section 1.2.3 that the naive approach of choosing $a$ as low as possible is not optimal, since a small $a$ allows for a higher cheating probability of the adversary which in turn must be reduced by raising $k$. Since $n = amk$, the aim is to find the *optimal tradeoff* between $a$ and $k$ that gives the smallest $n$ for a given error probability. The analysis below reveals that for the two-party case the best choice is to take $a = \tau$, and use $n = 2\tau k$ servers.

The adversary observes $(m-1)k$ servers in its watchlists and so in order to cheat in the outer protocol it needs to corrupt more than $\frac{amk}{\tau} - (m-1)k$ more servers (this is because $t = \frac{amk}{\tau}$ and it needs to corrupt more than $t$ servers in order to cheat). The adversary succeeds in corrupting this many servers (i.e., cheat in this many inner protocols) without being caught when none of the corrupted servers are in the honest client's set of $k$ watchlists. Since there are $amk$ servers, this happens with probability

$$\frac{\binom{amk-k}{\frac{amk}{\tau}-(m-1)k}}{\binom{amk}{\frac{amk}{\tau}-(m-1)k}}.$$

We now wish to minimize this cheating probability. Observe that this function depends on $a$, $m$, $k$ and $\tau$, where $m$ and $\tau$ are fixed ($m$ is the number of clients, and $\tau$ is fixed by the outer protocol that is being used). Thus, for any *given* error probability, fixing one of $a$ and $k$ determines the other. Specifically, if we fix the allowed error probability to be $2^{-40}$, for example, then for any given $a$ there exists a smallest possible $k$ that gives an error probability of at most $2^{-40}$. We therefore fix the error probability and find the value of $a$ that minimizes $k$ and thus the number of servers $n$, while staying within the given error probability.

Denote $\zeta = \frac{amk}{\tau} - (m-1)k$. Then, we have:

$$\begin{aligned}
\Pr[\mathsf{cheat}] &= \frac{\binom{amk-k}{\frac{amk}{\tau}-(m-1)k}}{\binom{amk}{\frac{amk}{\tau}-(m-1)k}} = \frac{\binom{amk-k}{\zeta}}{\binom{amk}{\zeta}} \\
&= \frac{(amk-k)!}{\zeta!\cdot(amk-\zeta-k)!}\cdot\frac{\zeta!\cdot(amk-\zeta)!}{(amk)!} \\
&= \frac{(amk-k)!}{(amk)!}\cdot\frac{(amk-\zeta)!}{(amk-\zeta-k)!} \\
&= \frac{1}{amk\cdot(amk-1)\cdot(amk-k+1)}\cdot(amk-\zeta)\cdot(amk-\zeta-1)\cdots(amk-\zeta-k+1) \\
&= \frac{amk-\zeta}{amk}\cdot\frac{amk-\zeta-1}{amk-1}\cdots\frac{amk-\zeta-k+1}{amk-k+1} \\
&\leq \left(\frac{amk-\zeta}{amk}\right)^k
\end{aligned}$$

Plugging in the value of $\zeta = \frac{amk}{\tau} - mk + k$, we obtain

$$\Pr[\mathsf{cheat}] \leq \left(\frac{amk - \frac{amk}{\tau} + mk - k}{amk}\right)^k = \left(1 - \frac{1}{\tau} + \frac{1}{a} - \frac{1}{am}\right)^k.$$

We wish to have $\Pr[\mathsf{cheat}] = 2^{-40}$ and thus

$$k \cdot \log\left(1 - \frac{1}{\tau} + \frac{1}{a} - \frac{1}{am}\right) = -40,$$

or equivalently

$$k = \frac{-40}{\log\left(1 - \frac{1}{\tau} + \frac{1}{a} - \frac{1}{am}\right)} \approx \frac{40}{\frac{1}{\tau} - \frac{1}{a} + \frac{1}{am}},$$

using the fact that $\log(1-x) \approx -x$. Since $n = amk$, we derive that

$$n \approx am \cdot \left(\frac{40}{\frac{1}{\tau} - \frac{1}{a} + \frac{1}{am}}\right) = \frac{40am}{\frac{1}{\tau} - \frac{1}{a} + \frac{1}{am}}.$$

Deriving by $a$ and solving, we have that a minima is reached when

$$a \approx 2\tau \cdot \left(1 - \frac{1}{m}\right).$$

We conclude that for an error probability of $2^{-40}$, the constant $a$ should be set at about $2\tau\cdot(1-1/m)$. In the special case of $m = 2$ clients, this means that $a$ should be set to $\tau$. In Section 1.2.3, we analyzed the concrete parameters for the case of $m = 2$ clients, and an outer protocol tolerating $t < n/2$ corruptions. In that case, we showed that it is better to set $n = 4k$ than $n = 3k$. Observe that this is exactly the result of the above general analysis: For $m = 2$ and $\tau = n/t = 2$, we have that $n = amk = \tau mk = 4k$ is optimal.

**Numerical experiments.** We have run a Python script to find the best choice of $a$ for for many different block sizes (and thus various corruption thresholds), ranging from $\ell = n/5$ to $\ell = n/105$, for an error probability of less than $2^{-40}$). We ran this script for different values of $m$ (the number of clients). The results of these experiments are that for the two party case ($m = 2$) the best choice is to take $a = \tau$, and have $n = 2\tau k$ servers. For larger values of $m$ the results are different: for example, for $m = 5$ the best choice is $a \approx 1.5\tau$, and for $m = 10$ the best choice is $a \approx 2\tau$. These results support our analysis above stating that $a$ should be approximately $2\tau \cdot (1 - \frac{1}{m})$; observe that for $m = 2$, $m = 5$ and $m = 10$ we obtain respective values of $a = \tau$, $a = 1.6\tau$ and $1.8\tau$, which are close to the numerical results.

### 6.3 Setting Concrete Values

We now show the concrete cost of IPS based on the results of our analysis regarding parameter instantiation. The choices of parameters are demonstrated for two different circuits, which clarify the dilemmas that arise in practice. As stated above the only parameter which we did not set independently of the circuit, but rather want to optimize for the concrete circuit, is the block size. We therefore calculated the number of operations required for a large range of block sizes. Our calculations are based on a combination of an analytic and numerical analysis of the parameters that yield a cheating probability of at most $2^{-40}$.

The first example uses circuit parameters similar to the AES circuit of [12], assuming 2400 multiplication gates split over 100 layers. Based on our Python script, we found that in this case minimal values for the number of OTs and the number of multiplications occur for the same block size of $\ell = n/73$ (as we will see in the next example, both of these costs are not necessarily minimized with the same choice of block size). Remembering that $\delta = n/4$, the threshold ratio is $\tau = \frac{1}{(\delta-\ell)/n} \approx 4.231$. Setting $a = \tau$ as suggested above results in $n = 4.231 \cdot 2k$. In order to obtain a cheating probability of $2^{-40}$, each client must have a watchlist of size $k = 207$, and the number of servers is $n \approx 1752$. Note that as expected, the block size, of $n/73 = 24$, is equal to its maximum reasonable value, namely to the width of a layer of the circuit, which is also 24.

The number of OTs and multiplications which are required in this setting depends on the inner multiplication protocol that is used (based on Reed-Solomon codes, or on the protocol of [17]). The first choice results in approximately $5.5 \cdot 10^6$ OTs and $5.5 \cdot 10^9$ multiplications, while the latter choice requires approximately $13.8 \cdot 10^6$ OTs and $4.5 \cdot 10^9$ multiplications. The choice of the inner protocol is therefore not trivial, and depends on the properties of concrete implementations of the OT and multiplication primitives. Recall that multiplications are in a finite field of size $2^{40}$ and therefore elements fit in a single word of a modern 64-bit architecture, and can be done very efficiently. The OTs need only be secure against semi-honest adversaries, and so can be efficiently implemented using methods of extending OT as in [26]. Given these two observations, the run time of the protocol seems reasonable in comparison to that of other protocols providing security against malicious adversaries.

The second example is of a circuit of 30000 multiplication gates split over 10 layers, and results in another optimization dilemma. Setting the block size $\ell = n/5.7$ results in the minimal number of OTs, but minimizing the number of multiplications requires setting $\ell = n/13.1$. The actual numbers of operations are described below.

| | $n$ | $k$ | RS OT | Gilboa OT | RS mult | Gilboa mult |
|---|---|---|---|---|---|---|
| $\boldsymbol{\ell = n/5.7}$ | 19554 | 729 | $5.6 \cdot 10^6$ | $11.1 \cdot 10^6$ | $54 \cdot 10^9$ | $53 \cdot 10^9$ |
| $\boldsymbol{\ell = n/13.1}$ | 3362 | 292 | $12 \cdot 10^6$ | $31 \cdot 10^6$ | $13 \cdot 10^9$ | $11 \cdot 10^9$ |

The reason for this tradeoff is that the number of OTs is minimized when the block size can accommodate an entire layer in a single block, but this setting requires more servers (compared to a smaller block size), and so multi-point evaluation and interpolation become more expensive (as they depend on the number of servers), which results in an increased amount of multiplications. Observe also that setting $\ell = n/5.7$ results in a much larger number of servers which in turn affects the cost of the watchlist setup protocol. Plugging in the cost of our watchlist setup ($15n + k$ exponentiations), we have that when $\ell = n/5.7$ the setup cost is $294,039$ exponentiations, versus just $50,722$ when $\ell = n/13.1$. This cost may also weigh in as a factor. (The actual optimum might be taking some block size between these two given values, but this depends on the cost of a single OT compared to the cost of a single multiplication.)

We conclude that the IPS protocol may be competitive in some settings. We are currently implementing the protocol in order to empirically verify our analysis and conclusions.

## References


[1] Y. Aumann and Y. Lindell. Security Against Covert Adversaries: Efficient Protocols for Realistic Adversaries. In the *Journal of Cryptology*, 23(2):281–343, 2010.

[2] D. Beaver. Multiparty Protocols Tolerating Half Faulty Processors. In *CRYPTO'89*, Springer-Verlag (LNCS 435), pages 560–572, 1990.

[3] D. Beaver. Correlated Pseudorandomness and the Complexity of Private Computations. In the 28*th STOC*, pages 479–488, 1996.

[4] M. Ben-Or, S. Goldwasser and A. Wigderson. Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation. In 20*th STOC,* pages 1–10, 1988.

[5] R. Canetti. Security and Composition of Multiparty Cryptographic Protocols. *Journal of Cryptology*, 13(1):143–202, 2000.

[6] R. Canetti. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In 42*nd FOCS*, pages 136–145, 2001. Full version available at `http://eprint.iacr.org/2000/067`.

[7] D. Chaum, C. Crépeau and I. Damgård. Multi-party Unconditionally Secure Protocols. In 20*th STOC*, pages 11–19, 1988.

[8] H. Chen and R. Cramer. Algebraic Geometric Secret Sharing Schemes and Secure Multi-Party Computations over Small Fields. In *CRYPTO 2006*, Springer (LNCS 4117), pages 521–536, 2006.

[9] C. Crépeau, J. van de Graaf and A. Tapp. Committed Oblivious Transfer and Private Multi-Party Computation. In *CRYPTO 1995*, Springer (LNCS 963), pages 110–123, 1995.

[10] I. Damgård, M. Geisler, and J.B. Nielsen. From Passive to Covert Security at Low Cost. In the 7*th TCC*, Springer (LNCS 5978), pages 128–145, 2010.

[11] I. Damgård and Y. Ishai. Scalable Secure Multiparty Computation. In *CRYPTO 2006*, Springer (LNCS 4117), pages 501–520, 2006.

[12] I. Damgård and M. Keller. Secure Multiparty AES. In the 14*th Financial Cryptography*, Springer (LNCS 6052), pages 367–374, 2010.

[13] M. Fitzi, M. Hirt and U.M. Maurer. Trading Correctness for Privacy in Unconditional Multi-Party Computation. In *CRYPTO 1998*, Springer (LNCS 1462), pages 121–136, 1998.

[14] M.K. Franklin and M. Yung. Communication Complexity of Secure Computation (Extended Abstract). In the 24*th STOC*, pages 699–710, 1992.

[15] J. Gathen and J. Gerhard. *Modern Computer Algebra*. Cambridge University Press, 2003

[16] Y. Gertner, S. Kannan, T. Malkin, O. Reingold, and M. Viswanathan. The Relationship Between Public Key Encryption and Oblivious Transfer. In the 41*st FOCS*, page 325–335, 2000.

[17] N. Gilboa. Two Party RSA Key Generation. In *CRYPTO'99*, Springer (LNCS 1666), pages 116–129, 1999.

[18] O. Goldreich. *Foundations of Cryptography: Volume 2 – Basic Applications*. Cambridge University Press, 2004.

[19] O. Goldreich, S. Micali and A. Wigderson. How to Play any Mental Game – A Completeness Theorem for Protocols with Honest Majority. In 19*th STOC,* pages 218–229, 1987. For details, see [18].

[20] V. Goyal, P. Mohassel and A. Smith. Efficient Two Party and Multi Party Computation Against Covert Adversaries. In *EUROCRYPT 2008*, Springer (LNCS 4965), pages 289–306, 2008.

[21] I. Haitner. Semi-honest to Malicious Oblivious Transfer – the Black-Box Way. In the *5th TCC*, Springer-Verlag (LNCS 4948) pages 412–426, 2008.

[22] I. Haitner, Y. Ishai, E. Kushilevitz, Y. Lindell and E. Petrank. Black-Box Constructions of Protocols for Secure Computation (full version combining [21, 25]). In the *SIAM Journal on Computing*, 40(2):225–266, 2011.

[23] R. Impagliazzo and S. Rudich. Limits on the Provable Consequences of One-way Permutations. In 21*st STOC*, pages 44–61, 1989.

[24] Y. Ishai. Personal communication, 2011.

[25] Y. Ishai, E. Kushilevitz, Y. Lindell and E. Petrank. Black-Box Constructions for Secure Multiparty Computation. In the 38*th STOC*, pages 99-108, 2006.

[26] Y. Ishai, J. Kilian, K. Nissim and E. Petrank. Extending Oblivious Transfer Efficiently. In *CRYPTO 2003*, Springer (LNCS 2729), pages 145–161, 2003.

[27] Y. Ishai, M. Prabhakaran and A. Sahai. Founding Cryptography on Oblivious Transfer – Efficiently. In *CRYPTO 2008*, Springer (LNCS 5157), pages 572–591, 2008.

[28] Y. Ishai, M. Prabhakaran and A. Sahai. Secure Arithmetic Computation with No Honest Majority. In the 6*th TCC*, Springer (LNCS 5444), pages 294–314, 2009.

[29] Y. Lindell and B. Pinkas. Secure Two-Party Computation via Cut-and-Choose Oblivious Transfer. In the *8th TCC*, Springer (LNCS 6597), pages 329-346, 2011.

[30] M. Naor and B. Pinkas. Oblivious Transfer with Adaptive Queries. In *CRYPTO 1999*, Springer (LNCS 1666), pages 573–590, 1999.

[31] B. Pinkas, T. Schneider, N.P. Smart and S.C. Williams. Secure Two-Party Computation Is Practical. *ASIACRYPT 2009*, Springer (LNCS 5912), 250–267, 2009.

[32] C. Peikert, V. Vaikuntanathan and B. Waters. A Framework for Efficient and Composable Oblivious Transfer. In *CRYPTO 2008*, Springer-Verlag (LNCS 5157), pages 554–571, 2008.

[33] M. Rabin. How to Exchange Secrets by Oblivious Transfer. Tech. Memo TR-81, Aiken Computation Laboratory, Harvard U., 1981.

[34] T. Rabin and M. Ben-Or. Verifiable Secret Sharing and Multi-party Protocols with Honest Majority. In 21*st STOC*, pages 73–85, 1989.

[35] A. Shamir. How to Share a Secret. In *Communications of the ACM*, 22(11):612–613, 1979.

[36] A. Yao. How to Generate and Exchange Secrets. In 27*th FOCS*, pages 162–167, 1986.

# A Security in the Presence of Covert Adversaries

## A.1 Motivation

The definition of security in this model is based on the ideal/real simulation paradigm (as in the standard definition of security today; see [5, 18]), and provides the guarantee that if the adversary cheats, then it will be caught by the honest parties (with some probability). In order to understand what we mean by this, we have to explain what we mean by "cheating". Loosely speaking, we say that an adversary successfully cheats if it manages to do something that is impossible in the ideal model. Stated differently, successful cheating is behavior that cannot be simulated in the ideal model. Thus, for example, an adversary who learns more about the honest parties' inputs than what is revealed by the output has cheated. In contrast, an adversary who uses pseudorandom coins instead of random coins (where random coins are what are specified in the protocol) has not cheated.

We are now ready to informally describe the guarantee provided by this notion. Let $0 < \epsilon \leq 1$ be a value (called the *deterrence factor*). Then, any attempt to cheat by a real adversary $\mathcal{A}$ is detected by the honest parties with probability at least $\epsilon$. Thus, provided that $\epsilon$ is sufficiently large, an adversary that wishes not to be caught cheating will refrain from *attempting* to cheat, lest it be caught doing so. Clearly, the higher the value of $\epsilon$, the greater the probability adversarial behavior is caught and thus the greater the *deterrent* to cheat. This notion is therefore called security in the presence of covert adversaries with $\epsilon$-deterrent. Note that the security guarantee does not preclude successful cheating. Indeed, if the adversary decides to cheat it may gain access to the other parties' private information or bias the result of the computation. The only guarantee is that if it attempts to cheat, then there is a fair chance that it will be caught doing so. This is in contrast to standard definitions, where absolute privacy and security are guaranteed for the

given type of adversary. We remark that by setting $\epsilon = 1$, the definition can be used to capture a requirement that cheating parties are always caught.

The above intuitive notion can be interpreted in a number of ways. We present the main formulation here. The definition works by modifying the ideal model so that the ideal-model adversary (i.e., simulator) is explicitly given the ability to cheat. Specifically, the ideal model is modified so that a special $\mathsf{cheat}$ instruction can be sent by the adversary to the trusted party. Upon receiving such an instruction, the trusted party tosses coins and with probability $\epsilon$ announces to the honest parties that cheating has taken place (by sending the message $\mathsf{corrupted}_i$ where party $P_i$ is the corrupted party that sent the $\mathsf{cheat}$ instruction). In contrast, with probability $1 - \epsilon$, the trusted party sends the honest party's input to the adversary, and in addition lets the adversary fix the output of the honest party. We stress that in this case the trusted party does not announce that cheating has taken place, and so the adversary gets off scot-free. Observe that if the trusted party announces that cheating has taken place, then the adversary learns absolutely nothing. This is a strong guarantee because when the adversary attempts to cheat, it must take the risk of being caught and gaining nothing.

## A.2 The Actual Definition

We begin by presenting the modified ideal model. In this model, we add new instructions that the adversary can send to the trusted party. Recall that in the standard ideal model, the adversary can send a special $\mathsf{abort}_i$ message to the trusted party, in which case the honest party receives $\mathsf{abort}_i$ as output. In the ideal model for covert adversaries, the adversary can send the following additional special instructions:

- *Special input* $\mathsf{corrupted}_i$*:* If the ideal-model adversary sends $\mathsf{corrupted}_i$ instead of an input, the trusted party sends $\mathsf{corrupted}_i$ to the honest party and halts. This enables the simulation of behavior by a real adversary that always results in detected cheating. (It is not essential to have this special input, but it sometimes makes proving security easier.)

- *Special input* $\mathsf{cheat}_i$*:* If the ideal-model adversary sends $\mathsf{cheat}_i$ instead of an input, the trusted party tosses coins and with probability $\epsilon$ determines that this "cheat strategy" by $P_i$ was detected, and with probability $1 - \epsilon$ determines that it was not detected. If it was detected, the trusted party sends $\mathsf{corrupted}_i$ to the honest party. If it was not detected, the trusted party hands the adversary the honest party's input and gives the ideal-model adversary the ability to set the output of the honest party to whatever value it wishes. Thus, a $\mathsf{cheat}_i$ input is used to model a protocol execution in which the real-model adversary decides to cheat. However, as required, this cheating is guaranteed to be detected with probability at least $\epsilon$. Note that if the cheat attempt is not detected then the adversary is given "full cheat capability", including the ability to determine the honest party's output.

The idea behind the new ideal model is that given the above instructions, the adversary in the ideal model can choose to cheat, with the caveat that its cheating is guaranteed to be detected with probability at least $\epsilon$. We stress that since the capability to cheat is given through an "input" that is provided to the trusted party, the adversary's decision to cheat must be made before the adversary learns anything (and thus independently of the honest party's input and the output).

We are now ready to present the modified ideal model. Let $\epsilon : \mathbb{N} \to [0, 1]$ be a function. Then, the ideal execution for a function $f : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}^* \times \{0,1\}^*$ with parameter $\epsilon$ proceeds as follows:

**Inputs:** Each party obtains an input; the $i^{\text{th}}$ party's input is denoted by $x_i$; we assume that all inputs are of the same length, denoted $n$. The adversary receives an auxiliary-input $z$.

**Send inputs to trusted party:** Any honest party $P_j$ sends its received input $x_j$ to the trusted party. The corrupted parties, controlled by $\mathcal{A}$, may either send their received input, or send some other input of the same length to the trusted party. This decision is made by $\mathcal{A}$ and may depend on the values $x_i$ for $i \in I$ and the auxiliary input $z$. Denote the vector of inputs sent to the trusted party by $\overline{w}$.

**Abort options:** If a corrupted party sends $w_i = \mathsf{abort}_i$ to the trusted party as its input, then the trusted party sends $\mathsf{abort}_i$ to all of the honest parties and halts. If a corrupted party sends $w_i = \mathsf{corrupted}_i$ to the trusted party as its input, then the trusted party sends $\mathsf{corrupted}_i$ to all of the honest parties and halts. If multiple parties send $\mathsf{abort}_i$ (resp., $\mathsf{corrupted}_i$), then the trusted party relates only to one of them (say, the one with the smallest $i$). If both $\mathsf{corrupted}_i$ and $\mathsf{abort}_j$ messages are sent, then the trusted party ignores the $\mathsf{corrupted}_i$ message.

**Attempted cheat option:** If a corrupted party sends $w_i = \mathsf{cheat}_i$ to the trusted party as its input, then the trusted party works as follows:

1. With probability $\epsilon$, the trusted party sends $\mathsf{corrupted}_i$ to the adversary and all of the honest parties.
2. With probability $1 - \epsilon$, the trusted party sends $\mathsf{undetected}$ to the adversary along with the honest parties' inputs $\{x_j\}_{j \notin I}$. Following this, the adversary sends the trusted party output values $\{y_j\}_{j \notin I}$ of its choice for the honest parties. Then, for every $j \notin I$, the trusted party sends $y_j$ to $P_j$.

If the adversary sent $\mathsf{cheat}_i$, then the ideal execution ends at this point. Otherwise, the ideal execution continues below.

**Trusted party answers adversary:** The trusted party computes $(f_1(\overline{w}), \ldots, f_m(\overline{w}))$ and sends $f_i(\overline{w})$ to $\mathcal{A}$, for all $i \in I$.

**Trusted party answers honest parties:** After receiving its outputs, the adversary sends either $\mathsf{abort}_i$ for some $i \in I$, or $\mathsf{continue}$ to the trusted party. If the trusted party receives $\mathsf{continue}$ then it sends $f_j(\overline{w})$ to all honest parties $P_j$ ($j \notin I$). Otherwise, if it receives $\mathsf{abort}_i$ for some $i \in I$, it sends $\mathsf{abort}_i$ to all honest parties.

**Outputs:** An honest party always outputs the message it obtained from the trusted party. The corrupted parties output nothing. The adversary $\mathcal{A}$ outputs any arbitrary (probabilistic polynomial-time computable) function of the initial inputs $\{x_i\}_{i \in I}$, the auxiliary input $z$, and the messages obtained from the trusted party.

The output of the honest parties and the adversary in an execution of the above ideal model is denoted by $\text{IDEALSC}^{\epsilon}_{f,\mathcal{S}(z),I}(\overline{x}, n)$.

Notice that there are two types of "cheating" here. The first is the classic $\mathsf{abort}$ and is used to model "early aborting" due to the impossibility of achieving fairness in general when there is no honest majority. The other type of cheating in this ideal model is more serious for two reasons: first, the ramifications of the cheating are greater (the adversary may learn the honest party's input and

may be able to determine its output), and second, the cheating is only guaranteed to be detected with probability $\epsilon$. Nevertheless, if $\epsilon$ is high enough, this may serve as a deterrent. We stress that in the ideal model the adversary must decide whether to cheat obliviously of the honest party's input and before it receives any output (and so it cannot use the output to help it decide whether or not it is "worthwhile" cheating). We have the following definition.

**Definition A.1** (security – strong explicit cheat formulation [1]): *Let $f$ be an $m$-party functionality and $\pi$ a protocol, and let $\epsilon : \mathbb{N} \to [0,1]$ be a function. Protocol $\pi$ is said to* securely compute $f$ in the presence of covert adversaries with $\epsilon$-deterrent *if for every non-uniform probabilistic polynomial-time adversary $\mathcal{A}$ for the real model, there exists a non-uniform probabilistic polynomial-time adversary $\mathcal{S}$ for the ideal model such that for every $I \subseteq [m]$:*

$$\left\{\text{IDEALSC}^{\epsilon}_{f,\mathcal{S}(z),I}(\overline{x},n)\right\}_{\overline{x},z,n} \stackrel{\text{c}}{\equiv} \left\{\text{REAL}_{\pi,\mathcal{A}(z),I}(\overline{x},n)\right\}_{\overline{x},z,n}$$

*where $n \in \mathbb{N}$ and $\overline{x}, z \in \{0,1\}^*$, and all elements of $\overline{x}$ are of the same length.*

**An ideal functionality $\mathcal{F}^{\epsilon}$.** An equivalent way of defining security in the presence of covert adversaries is to consider a *reactive* functionality that includes all of the instructions for the trusted party that are unique to the covert model. This reactive functionality is then run by the trusted party in the standard model for malicious adversaries. This yields an equivalent definition, and it is sometimes useful to view it in this way.

### A.3 Some Small Modifications

**Output agreement.** Recall that when the adversary is *not* caught cheating, it can provide whatever outputs it wishes to the honest parties. For our purposes here, we need to ensure that when the functionality being computed is such that all parties are supposed to receive the same output, then even when the adversary is not caught cheating (i.e., it is undetected) and so can determine the honest parties' output, all honest parties must receive the *same output*. When using a broadcast channel (or Byzantine agreement), this property is easily achieved.

**Reactive functionalities.** Another additional modification that we need relates to the computation of reactive functionalities. Such functionalities have multiple stages of computation, and *internal state* is kept between stages. In the case of a cheating adversary who is undetected, the adversary can not only singlehandedly determine the outputs of the honest parties, but can also singlehandedly determine the internal state of the functionality. In this way, the constructions of [1, 20] can be modified in a straightforward way to also work for reactive functionalities (briefly, each party's output at the end of a stage includes a signed share of the internal state).

# B Detailed Description of the Protocols and Their Exact Analysis

In this section, we present our "raw experimental results" of counting the number of operations in the IPS instantiation that we studied. As such, this section is less polished and is presented only to enable a verification of our results, and in order to be very concrete about the exact instantiation and methodology that we used.

## B.1 Secret Sharing for Blocks

This secret sharing scheme is a variant of Shamir's secret sharing [35] which is presented in [14]. In order to share $w$ elements, using blocks of size $\ell$ and polynomials of degree $\delta$ among $n$ servers we do the following. First we set up the elements in $w/l$ blocks. Next for each such block we interpolate through $\delta$ points to get the polynomial, and then evaluate the polynomial at $n$ other points to get its shares. Overall this costs $(w/l)(\delta^2 + n\delta)$ multiplications. [8]

**Protocol B.1 (Block Secret Sharing)**

- **Input:** $w$ elements
- **Output:**
  1. Having the input split into blocks, the appropriate share of each block sent on its matching watchlist.
  2. An additive share of each share sent to the other party.
- **Auxiliary input:**
  1. $\ell$ - the size of a block.
  2. $\delta$ - the polynomial degree.
  3. $\zeta_1, ..., \zeta_\delta$ - the evaluation points that hide the secrets.
  4. $n$ - the number of shares per block.
  5. $\xi_1, ..., \xi_n$ - the evaluation points that are used to create shares.

1. Split the input into $\frac{w}{\ell}$ blocks
2. For each block $b = (x_1, ..., x_\ell)$:
   (a) For evaluation points $\zeta_1, ..., \zeta_\delta$ set $f_b(\zeta_i) = x_i$ for $1 \leq i \leq \ell$, and $f_b(\zeta_i) \in_R \mathcal{F}$ for $\ell < i \leq \delta$
   (b) Interpolate to obtain $f_b$
   (c) Evaluate $f_b(\xi_i)$ for $1 \leq i \leq n$ (where $\xi_i \neq \zeta_j$ for $1 \leq i \leq n, 1 \leq j \leq \ell$
   (d) For $1 \leq i \leq n$:
      i. On watchlist $i$ send the share $f_b(\xi_i)$
      ii. Choose $a_{f_b(\xi_i)} \in_R \mathcal{F}$, and sent it to other client, while keeping $f_b(\xi_i) - a_{f_b(\xi_i)}$ (this is additive sharing of the shares)

[8] We are using Horner's rule for multi-point evaluation and interpolation in our calculations which costs $O(n^2)$. Though asymptotically it is worse than the methods which use FFT (that require $O(n \log^2 n \log \log n)$ operations and can be found in [15]), for our magnitude of $n$ using Horner's rule is more efficient, or approximately as efficient.

**Cost analysis:**

- **Random elements:** $\frac{w}{\ell}(\delta - \ell) + \frac{w}{\ell}n$
- **Multiplications:** $\frac{w}{\ell}(\delta^2 + \delta n)$
- **Elements sent on watchlists:** $\frac{w}{\ell}n$
- **Elements sent in plain:** $\frac{w}{\ell}n$

### B.2 Proving that Shares Lie on $\delta$-degree Polynomials

After sharing the secrets as described in the previous section, we will want a party to prove to the other party that the shares indeed lie on (some number) $z$ polynomials, each of degree $\delta$. A protocol for such a proof (in a generalized form) is presented in [28]. The multiplication cost for the prover is $\delta n$ (the cost of creating shares for a blinding polynomial) and $z\delta$ (the cost of forming the linear combinations from the polynomials). The cost for the verifier is $k\delta$ (the cost of checking results on the $k$ watchlists he sees). So the total cost is $\delta(z + n + k)$.

**Prover cost analysis:**

- **Random elements:** $\delta$
- **Multiplications:** $n\delta + z\delta$
- **Elements sent on watchlists:** $n$
- **Elements sent in plain:** $\delta$

**Verifier cost analysis:**

- **Random elements:** $z$
- **Multiplications:** $kz$
- **Elements sent on watchlists:** 0
- **Elements sent in plain:** $z$

### B.3 Proving some replication Pattern in Shared blocks

We will need at some stages for a party to prove to the other party that certain shared blocks follow some replication pattern (which represents the fact that a certain output value is indeed the one used as the input value for the next layer). We also note that some of the blocks will be shared by $\delta$-degree polynomials while others by $2\delta$-degree polynomials (as some of them are results of a previous layer, and others are inputs for the next layer).

Here we chose not the implementation which is presented in the main part of the original protocol described in [28], but the one that is mentioned in its computation complexity analysis, in the end. The idea is that given $v$ blocks $b_i$, we can build $v$ random blocks $r_i$, and from them form a set of blocks $r'_i$ by shift cycling the values of the random blocks in the positions which are to be verified to be equal. Then what is left to prove is that $\sum(v_i r_i) = \sum(v_i r'_i)$. For example, given 2 blocks $(a, b, c, d), (e, f, g, h)$ and a replication pattern $(x, x, y, y), (x, y, x, x)$, we build the random blocks $r_i$ to be $(r_1, r_2, r_3, r_4), (r_5, r_6, r_7, r_8)$. Then the appropriate shift cycled blocks $r'_i$ will be $(r_2, r_5, r_4, r_6), (r_7, r_3, r_8, r_1)$.

In general we can label the different random values $r_i$ with different labels $x, y, z$,etc., according to the replication pattern, similarly to the description in the above example. Then the shift cycled blocks are created by shifting the $r_i$'s which have the same label. The idea is that shifting along indexes which are supposed to be equal will not change the sum.

**Protocol B.2 (Proving sets of shares lie on polynomials of degree $\delta$)**

- **Input:**
  1. Prover: $z$ polynomials $f_i$
  2. Verifier: $k$ shares for each polynomial $f_i$ that it saw on the watchlists
- **Output:**
  1. Verifier accepts the proof (or aborts if some check fails)
- **Auxiliary input:**
  1. $\delta$ - the polynomial degree.
  2. $n$ - the number of shares per block.
  3. $\xi_1, ..., \xi_n$ - the evaluation points that are used to create shares.
  4. $z$ - the number of polynomials that the prover has as input.

1. Prover chooses a $\delta$-degree random polynomial $f_b$ for blinding
2. Prover evaluates $f_b(\xi_i)$ for $1 \leq i \leq n$
3. Prover sends on watchlist $i$ the share $f_b(\xi_i)$
4. Verifier chooses $z$ elements $r_1, ..., r_z \in_R \mathcal{F}$
5. Verifier sends the $z$ random elements to the prover
6. Prover computes $f(x) = \sum_{i=1}^{z}(r_i f_i(x)) + f_b(x)$, where $f_i$ is the $i$th polynomial, and $f_b$ is the blinding polynomial from step 1
7. Prover sends $f$ to the verifier
8. Verifier checks for the $k$ watchlists it sees that: $f(\kappa) = \sum_{i=1}^{z}(r_i f_i(\kappa)) + f_b(\kappa)$
   (we note that $f_i(\kappa), f_b(\kappa)$ are shares that have been sent on the watchlist)

The cost for the prover encompasses the sharing of a $4\delta$-degree blinding polynomial ($(4\delta)^2 + 4\delta n$ multiplications), creating shares of the $2(u + v)$ (where $u$ is the number of output blocks represented by $2\delta$-degree polynomials, and $v$ is the number of input blocks represented by $\delta$-degree polynomials) random polynomials sent by the verifier ($2(2\delta nu + \delta nv)$ multiplications) and lastly calculating shares of the resulting polynomial (which costs $2n(u + v)$ multiplications). The cost for the verifier is creating the $2(u + v)$ random polynomials ($2((2\delta)^2 u + \delta^2 v)$ multiplications) and checking the results on the watchlists he sees ($2k(v + u)$ multiplications). In total we get the cost of $(4\delta)^2 + 4\delta n + 2((2\delta n + (2\delta)^2)u + (\delta n + \delta^2)v) + 2(n + k)(u + v)$.

**Protocol B.3 (Proving some replication patterns in shared blocks)**
**Input:**

1. Prover: $v$ blocks shared by $\delta$-degree polynomials and $u$ blocks shared by $2\delta$-degree polynomials.
2. Verifier: $k$ shares for each polynomial that it saw on the watchlists

**Output:**

1. Verifier accepts the proof (or aborts if some check fails)

**Auxiliary input:** $\boldsymbol{\ell}$ - the size of a block; $\boldsymbol{\delta}$ - the polynomial degree; $\boldsymbol{\zeta_1, ..., \zeta_2\delta}$ - the evaluation points that hide the secrets; $\boldsymbol{n}$ - the number of shares per block; $\boldsymbol{\xi_1, ..., \xi_n}$ - the evaluation points that are used to create shares; $\boldsymbol{z}$ - the number of polynomials that the prover has as input; A description of the circuit to evaluate $C$.

**The protocol:**

1. Prover $P$ sets for evaluation points $\zeta_1, ..., \zeta_{4\delta}$: $f_0(\zeta_j) = 0$ for $1 \leq j \leq \ell$, and $f_0(\zeta_j) \in_R \mathcal{F}$ for $\ell < j \leq 4\delta$
2. $P$ interpolates to obtain $f_0$
3. $P$ evaluates $f_0(\xi_i)$ for $1 \leq i \leq n$
4. $P$ sends on watchlist $i$ the share $f_0(\xi_i)$
5. Verifier $V$ chooses $(v + u)\ell$ elements $r_1, ..., r_{(v+u)\ell} \in_R \mathcal{F}$ and sets blocks $q_h = (r_{(h-1)+1}, ..., r_{(h-1)+\ell})$ for $1 \leq h \leq v + u$
6. For each block $q_h$ where $1 \leq h \leq v$:
   (a) $V$ sets for evaluation points $\zeta_1, ..., \zeta_\delta$: $f_{q_h}(\zeta_j) = r_{(h-1)+j}$ for $1 \leq j \leq \ell$, and $f_{q_h}(\zeta_j) \in_R \mathcal{F}$ for $\ell < j \leq \delta$
   (b) $V$ interpolates to obtain $f_{q_h}$
   (c) $V$ sends $f_{q_h}$ to $P$
7. For each block $q_h$ where $v < h \leq v + u$:
   (a) $V$ sets for evaluation points $\zeta_1, ..., \zeta_{2\delta}$: $f_{q_h}(\zeta_j) = r_{(h-1)+j}$ for $1 \leq j \leq \ell$, and $f_{q_h}(\zeta_j) \in_R \mathcal{F}$ for $\ell < j \leq 2\delta$
   (b) $V$ interpolates to obtain $f_{q_h}$
   (c) $V$ sends $f_{q_h}$ to $P$
8. $V$ sets the shift cycled blocks $p_h$ according to the desired replication pattern and repeats steps 6 and 7 for $p_h$
9. $P$ evaluates $f_{q_h}(\xi_i), f_{p_h}(\xi_i)$, for $1 \leq i \leq n$, for each of the polynomials $f_{q_h}, f_{p_h}$
10. For $1 \leq i \leq n$: $P$ computes $f_{res}(\xi_i) = \sum_{h=1}^{v+u} (f_h(\xi_i) f_{q_h}(\xi_i) - f_h(\xi_i) f_{p_h}(\xi_i)) + f_0(\xi_i)$, where $f_h$ is the polynomial used to share the original $h$th block
11. $P$ sends to $V$ all resulting shares
12. $V$ interpolates over the shares to get $f_{res}$
13. $V$ evaluates $f_{res}(\zeta_j)$ for $1 \leq j \leq \ell$
14. $V$ validates that $\sum_{j=1}^{\ell} f_{res}(\zeta_j) = 0$
15. $V$ validates for every server on its watchlist that it sees that: $f_{res}(\kappa) = \sum_{h=1}^{v+u} (f_h(\kappa) f_{q_h}(\kappa) - f_h(\kappa) f_{p_h}(\kappa)) + f_0(\kappa)$

**Prover cost analysis:**

- **Random elements:** $4\delta - \ell$
- **Multiplications:** $(4\delta)^2 + 4\delta n + 2(\delta nv + 2\delta nu) + n(2(v+u))$
- **Elements sent on watchlists:** $n$
- **Elements sent in plain:** $n$

**Verifier cost analysis:**

- **Random elements:** $(v+u)\ell + 2(v(\delta - \ell) + u(2\delta - \ell))$
- **Multiplications:** $2(\delta^2 v + (2\delta)^2 u) + (4\delta)^2 + 4\delta\ell + k(2(v+u))$
- **Elements sent on watchlists:** 0
- **Elements sent in plain:** $2(\delta v + 2\delta u)$

### B.4 Semi-honest Inner Multiplication Protocol

As we have mentioned multiplication gates involve having the parties engage in a semi-honest protocol for the following functionality:

$$(x_1, x_2) \mapsto (x_1 x_2 - r, r)$$

for some $r \in_R \mathcal{F}$.

Six different protocols are presented for this functionality in [28], that fall in three categories: statistically secure, based on noisy linear codes and based on homomorphic encryption. We analyze here only the most efficient protocol of the six which is the packed Reed-Solomon encoding based protocol, and an alternative *non black-box* protocol from [17] which makes non black-box usage of the underlying field and assumes standard bit representation of elements, but achieves comparable efficiency. [9]

In the packed Reed-Solomon encoding based protocol we are multiplying $w = s/2$ elements in a single run of the protocol (where $s$ is a security parameter). We will also set $n$ which is the length of the codeword to be $8s$ as suggested in [28].

A detailed inspection of the protocol results in a cost of $s^2 + (2s-1)s + (2s)^2 + 2sw = 8s^2 - s$ multiplications for side B, and $s^2 + 8s^2 + 16s^2 + 8s + 2sw = 26s^2 + 8s$ for side A, and $8s$ instances of 1-out-of-2 OT. Which means that per inner multiplications the cost is $(34s^2 + 7s)/(s/2) = 68s + 14$, and $8s/(s/2) = 16$ 1-out-of-2 OTs.

Setting $s = 40$ we get the cost of 2734 multiplications and 16 1-out-of-2 OTs per inner multiplication.

The protocol from [17] is much simpler and it is easy to see that it requires $\log |\mathcal{F}|$ multiplications and $\log |\mathcal{F}|$ 1-out-of-2 OTs. Remembering that we assumed $\mathcal{F}$ that is big enough to allow a single invocation of the different proofs above (to achieve a small enough cheating probability), we can set $|\mathcal{F}| = 2^{40}$ (as we did in the concrete computations presented later on) and get the cost of 40 multiplications and 1-out-of-2 OTs per inner multiplication.

As we can see one protocol is more efficient in terms of OTs and the other in terms of multiplications. We will see that for concrete numbers this presents a real dilemma which is not simple to resolve.

We present the analysis for the statistically secure protocol, the Reed-Solomon encoding based protocol and the more efficient homomorphic encryption based protocol, all from [28]. In addition we give a detailed presentation of the protocol from [17].

[9] We do not present the homomorphic encryption based protocols because even though they might be efficient, their comparison to the other protocols is more complex as they use completely different building blocks.

**Basic statistically secure protocol:** Following the same notations as in [28] ($n > \log|F| + k$) we get:

**Alice cost analysis:**

- **Random elements:** $n$
- **Multiplications:** $2n$
- **Elements sent in plain:** 0

**Bob cost analysis:**

- **Random elements:** $2n - 1$ (and $n$ more random bits)
- **Multiplications:** 0
- **Elements sent in plain:** $2n$

**Mutual cost:**

- **1-out-of-2 OTs:** $n$

**Packed Reed-Solomon encoding based protocol:** Here we are multiplying $w = s/2$ elements in a single run of the protocol (where $s$ is a security parameter), and so in order to get a comparable (amortized) analysis we divide all the results by $s/2$. We will also, as in [28] take the length of the codeword $n = 8s$.

**Alice cost analysis:**

- **Random elements:** $2.5s/(s/2) = 5$
- **Multiplications:** $(26s^2 + 8s)/(s/2) = 52s + 16$
- **Elements sent in plain:** 0

**Bob cost analysis:**

- **Random elements:** $(15.5s + 1)/(s/2) = 31 + \frac{2}{s}$
- **Multiplications:** $(8s^2 - s)/(s/2) = 16s - 2$
- **Elements sent in plain:** $17s/(s/2) = 34$

**Mutual cost:**

- **1-out-of-2 OTs:** $8s/(s/2) = 16$ (implementing a $s$-out-of-$n$ OT using $n$ instances of 1-out-of-2 OTs)

**Homomorphic encryption based protocol:** This protocol should be measured in terms of number of encryptions, decryptions and homomorphic operations on ciphertexts. The key generation can be done once and used as many times as necessary.

**Alice cost analysis:**

- **Encryptions:** 1
- **Decryptions:** 1
- **Homomorphic operations:** 0
- **Ciphertexts sent:** 1

**Bob cost analysis:**

- **Encryptions:** 1
- **Decryptions:** 0
- **Homomorphic operations:** 2
- **Ciphertexts sent:** 1

**Non black-box statistically secure protocol:** This protocol works as follows:

1. Bob selects $\log|F|$ random elements $s_0, ..., s_{\log|F|-1}$, and sets for $0 \leq i \leq \log|F| - 1$ $t_i^0 = s_i, t_i^1 = 2^i b + s_i$, where $b$ is Bob's input.
2. Alice and Bob execute $\log|F|$ 1-out-of-2 OTs where in the $i$th invocation Alice chooses $t_i^{a_i}$ from the pair $(t_i^0, t_i^1)$, where $a_i$ is the $i$th bit in the bit representation of Alice's input $a$.
3. Alice outputs $x = \sum_{i=0}^{\log|F|-1} t_i^{a_i}$ and Bob outputs $y = -\sum_{i=0}^{\log|F|-1} s_i$

Assuming Bob calculates the powers of 2 once and stores them, we get the following analysis:

**Alice cost analysis:**

- **Random elements:** $\log|F|$
- **Multiplications:** $\log|F|$
- **Additions and substractions:** $2\log|F|$
- **Elements sent in plain:** 0

**Bob cost analysis:**

- **Random elements:** 0
- **Multiplications:** 0
- **Additions and substractions:** $\log|F|$
- **Elements sent in plain:** 0

**Mutual cost:**

- **1-out-of-2 OTs:** $\log|F|$

## B.5 The Entire Protocol in the IPS Setting

Protocol B.4 is a detailed presentation of the entire protocol from Appendix C of [28] in the IPS setting, using the building blocks presented before. The watchlist setup phase and its analysis is omitted here.

In the cost analysis we leave the call for a semi-honest inner multiplication protocol (which is used to emulate a multiplication done by a server, and called in [27] a Type II operation) as a single operation.

The idea of the protocol is to utilize our block secret sharing in the computation by evaluating blocks of gates. We will arrange the input wires for each layer in a way that allows us to perform the same operation (addition or multiplication) on all the elements of the blocks.

First we show how given two input values and a blinding value which are additively shared among Alice and Bob they can compute a blinded sum of the values, and a blinded multiplication of them.

That is given that Alice holds $\big((a-\alpha),(b-\beta),(c-\gamma)\big)$ and Bob holds $(\alpha,\beta,\gamma)$ for some $\alpha,\beta,\gamma \in_R \mathcal{F}$, we show to how perform addition (have Alice hold $d-\lambda$, such that $d=((a+b)+c)$ and Bob hold $\lambda$ for some $\lambda \in_R \mathcal{F}$), and how to perform multiplication (have Alice hold $d-\lambda$, such that $d=(ab+c)$ and Bob hold $\lambda$ for some $\lambda \in_R \mathcal{F}$). $c$ here is the blinding value created by Bob in the outer protocol, and so each operation involves the addition of this value.

**Addition:**

Alice sets $(d-\lambda) = (a-\alpha)+(b-\beta)+(c-\gamma)$

Bob sets $\lambda = \alpha + \beta + \gamma$
We can see that $d - \lambda + \lambda = d = a + b + c$ as required, and $\lambda \in_R \mathcal{F}$.

**Multiplication:**
Alice and Bob run the semi-honest inner multiplication functionality twice:
$\big((a - \alpha), \beta)\big) \mapsto \big((a\beta - \alpha\beta - \mu), \mu\big)$, for $\mu \in_R \mathcal{F}$
$\big((b - \beta), \alpha)\big) \mapsto \big((b\alpha - \alpha\beta - \sigma), \sigma\big)$ for $\sigma \in_R \mathcal{F}$
Alice sets $(d - \lambda) = (a\beta - \alpha\beta - \mu) + (b\alpha - \alpha\beta - \sigma) + (a - \alpha)(b - \beta) + (c - \gamma)$
Bob sets $\lambda = \mu + \sigma + \alpha\beta + \gamma$
We can see that $d - \lambda + \lambda = ab + c$

**Protocol B.4 (Appendix C Protocol in the IPS Setting)**

- **Input:** Each client has $e_1$ inputs.
- **Output:** Each client receives output as dictated by the circuit $C$ (or aborts if cheating is detected)
- **Auxiliary input:**
  1. $\ell$ - the size of a block.
  2. $\delta$ - the polynomial degree.
  3. $\zeta_1, ..., \zeta_2\delta$ - the evaluation points that hide the secrets.
  4. $n$ - the number of shares per block.
  5. $\xi_1, ..., \xi_n$ - the evaluation points that are used to create shares.
  6. $z$ - the number of polynomials that the prover has as input.
  7. A description of the circuit to evaluate $C$.

1. Each client arranges its $e_1$ inputs into $\frac{e_1}{\ell}$ blocks as required by the structure of $C$ and shares these using protocol B.1
2. Each client proves using protocol B.2 that its shares lie on $\delta$-degree polynomials
3. For layer $i$ (having $2e_i$ input wires, and $e_i$ outputs wires, and so $e_i$ gates):
   (a) Bob chooses $e_i$ random blinding values $b_{e_i}$
   (b) Bob shares these values using $2\delta$-degree polynomials and protocol B.1
   (c) Bob proves using protocol B.2 that these shares lie on $2\delta$-degree polynomials
   (d) For each block of gates $g$:
      i. For $1 \leq j \leq n$:
         A. Compute the operation of $g$ on the two input shares and the blinding share as described above
   (e) Bob sends to Alice all his additive shares
   (f) Alice sums all the additive shares to get real shares of polynomials
   (g) For each output block:
      i. Alice interpolates through $2\delta$ shares to get the polynomial
      ii. Alice validates that the rest $n - 2\delta$ shares lie on the polynomial

**Protocol B.5 (Appendix C Protocol in the IPS Setting (cont.))**

3. (cont.)
   (g) (cont.)
      iii. Alice evaluates the polynomial at $\ell$ points to restore the secrets
   (h) Alice builds from the $e_i$ outputs the $2e_{i+1}$ inputs for the next layer
   (i) Alice shares these $2e_{i+1}$ elements using protocol B.1
   (j) Alice proves that these shares lie on $\delta$-degree polynomials
   (k) Alice proves that these $\frac{2e_{i+1}}{\ell}$ input blocks and the $\frac{e_i}{\ell}$ outputs blocks follow a replication pattern dictated by the structure of $C$ using protocol B.3
   (l) Bob arranges the blinding values form step 3.(a) for the new layer according to the structure of $C$
   (m) Bob shares these values using protocol B.1
   (n) Bob proves that these shares lie on $\delta$-degree polynomials
   (o) Bob proves that these $\frac{2e_{i+1}}{\ell}$ newly arranged blinding blocks and the original $\frac{e_i}{\ell}$ blinding blocks follow a replication pattern dictated by the structure of $C$ using protocol B.3
   (p) For each of the $\frac{2e_{i+1}}{\ell}$ blocks
      i. For $1 \leq j \leq n$:
         A. Alice and Bob each substract the new blinding share from the share of the block
4. Output delivery - For each output block:
   (a) If Alice should receive this output block:
      i. Bob sends Alice the additive shares for the shares of the block
      ii. Alice sums up the additive shares to restore the polynomial shares
      iii. Alice interpolates through $\delta$ shares to get the polynomial
      iv. Alice validates that the rest $n - \delta$ shares lie on the polynomial
      v. Alice evaluates the polynomial at $\ell$ points to restore the secrets
   (b) If Bob should receive this output block - Alice sends Bob the additive shares and Bob restores the secrets as in 4.(a).

## B.6 Efficiency Analysis

We denote by $e_{i,\times}$ the number of multiplication gates at layer $i$, and by $e_{i,+}$ the number of addition gates at layer $i$ (we have it that $e_i = e_{i,\times} + e_{i,+}$).

We present the analysis separately for each client, and for each operation (the amount of random elements, multiplications, elements sent on watchlists and elements send in plain). For Alice we first present the detailed step-by-step analysis followed by the sum of all steps and then an asymptotic analysis (for Bob the step-by-step and the asymptotic analysis are omitted as they are rather similar to these of Alice).

**Alice cost analysis:**

- **Random elements:**

1. $\frac{e_1}{\ell}(\delta-\ell)+\frac{e_1}{\ell}n$ - Step 1.
2. $\delta+\frac{e_1}{\ell}$ - Step 2 (once as prover and once as verifier).
3. $\frac{e_i}{\ell}$ - Step 3.(c) (as verifier).
4. $\frac{2e_{i+1}}{\ell}(\delta-\ell)+\frac{2e_{i+1}}{\ell}n$ - Step 3.(i).
5. $\delta$ - Step 3.(j) (as prover).
6. $4\delta-\ell$ - Step 3.(k) (as prover).
7. $\frac{2e_{i+1}}{\ell}$ - Step 3.(n) (as verifier).
8. $(\frac{2e_{i+1}}{\ell}+\frac{e_i}{\ell})\ell+2(\frac{2e_{i+1}}{\ell}(\delta-\ell)+\frac{e_i}{\ell}(2\delta-\ell))$ - 3.(o) (as verifier).

In total we get:

$$\frac{e_1}{\ell}(\delta-\ell+n+1)+\delta+\sum_{i=1}^{d-1}\big(\frac{e_i}{\ell}(4\delta-\ell+1)+\frac{2e_{i+1}}{\ell}(3\delta-2\ell+n+1)+5\delta-\ell\big)$$

Remembering that we have $\ell,\delta=O(n)$ we get that the above equation is $O(|C|+nd)$

- **Multiplications:**

1. $\frac{e_1}{\ell}(\delta^2+\delta n)$ - Step 1.
2. $n\delta+\frac{e_1}{\ell}\delta+k\frac{e_1}{\ell}$ - Step 2 (once as prover and once as verifier).
3. $k\frac{e_i}{\ell}$ - Step 3.(c) (as verifier).
4. $\frac{e_{i,\times}}{\ell}n$ - Step 3.(d).i.A (the local multiplications from computing the multiplications as presented before).
5. $\frac{e_i}{\ell}(2\delta)^2$ - Step 3.(g).i.
6. $\frac{e_i}{\ell}(2\delta)(n-2\delta)$ - Step 3.(g).ii.
7. $\frac{e_i}{\ell}(2\delta)\ell$ - Step 3.(g).iii.
8. $\frac{2e_{i+1}}{\ell}(\delta^2+\delta n)$ - Step 3.(i).
9. $n\delta+\frac{2e_{i+1}}{\ell}\delta$ - Step 3.(j) (as prover).
10. $(4\delta)^2+4\delta n+2(\delta n\frac{2e_{i+1}}{\ell}+2\delta n\frac{e_i}{\ell})+n(2(\frac{2e_{i+1}}{\ell}+\frac{e_i}{\ell}))$ - Step 3.(k) (as prover).
11. $k\frac{2e_{i+1}}{\ell}$ - Step 3.(n) (as verifier).
12. $2(\delta^2\frac{2e_{i+1}}{\ell}+(2\delta)^2\frac{e_i}{\ell})+(4\delta)^2+4\delta\ell+k(2(\frac{2e_{i+1}}{\ell}+\frac{e_i}{\ell}))$ - Step 3.(o) (as verifier).
13. $\frac{e_d}{\ell}\delta^2$ - Step 4.(a).ii.
14. $\frac{e_d}{\ell}\delta(n-\delta)$ - Step 4.(a).iii.
15. $\frac{e_d}{\ell}\delta\ell$ - Step 4.(a).iv.

In total we get:

$$\begin{aligned}&\frac{e_1}{\ell}(\delta^2+\delta n+\delta+k)+\delta n+\\&+\sum_{i=1}^{d-1}\big(\frac{e_{i,\times}}{\ell}n+\frac{e_i}{\ell}(6\delta n+2\delta\ell+4\delta^2+3k+2n)+\\&+\frac{2e_{i+1}}{\ell}(3\delta n+3\delta^2+3k+2n+\delta)+5\delta n+32\delta^2+4\delta\ell\big)+\frac{e_d}{\ell}(\delta n+\delta\ell)\end{aligned}$$

Asymptotically this is $O(|C|n+n^2d)$

- **Elements sent on watchlists:**

  1. $\frac{e_1}{\ell}n$ - Step 1.
  2. $n$ - Step 2 (as prover).
  3. $\frac{2e_{i+1}}{\ell}n$ - Step 3.(i).
  4. $n$ - Step 3.(j) (as prover).
  5. $n$ - Step 3.(k) (as prover).

  In total we get:

  $$\frac{e_1}{\ell}n + n + \sum_{i=1}^{d-1}\big(\frac{2e_{i+1}}{\ell}n + 2n\big)$$

  Which is $O(|C| + nd)$

- **Elements sent in plain:**

  1. $\frac{e_1}{\ell}n$ - Step 1.
  2. $\delta + \frac{e_1}{\ell}$ - Step 2 (once as prover and once as verifier).
  3. $\frac{e_i}{\ell}$ - Step 3.(c) (as verifier).
  4. $\frac{2e_{i+1}}{\ell}n$ - Step 3.(i)
  5. $\delta$ - Step 3.(j) (as prover).
  6. $n$ - Step 3.(k) (as prover).
  7. $\frac{2e_{i+1}}{\ell}$ - Step 3.(n) (as verifier).
  8. $2(\delta\frac{2e_{i+1}}{\ell} + 2\delta\frac{e_i}{\ell})$ - Step 3.(o) (as verifier).
  9. $\frac{e_d}{\ell}n$ - Step 4.(b).i.

  In total we get:

  $$\frac{e_1}{\ell}(n+1) + \delta + \sum_{i=1}^{d-1}\big(\frac{e_i}{\ell}(4\delta+1) + \frac{2e_{i+1}}{\ell}(2\delta+n+1) + \delta + n\big) + \frac{e_d}{\ell}n$$

  And asymptotically this is $O(|C| + nd)$

**Bob cost analysis:**

- **Random elements:**

  $$\frac{e_1}{\ell}(\delta - \ell + n + 1) + \delta + \sum_{i=1}^{d-1}\big(\frac{e_i}{\ell}(6\delta - 2\ell + n) + \frac{2e_{i+1}}{\ell}(3\delta - 2\ell + n + 1) + e_i + 7\delta - \ell\big)$$

- **Multiplications:**

$$\begin{aligned}&\frac{e_1}{\ell}(\delta^2+\delta n+\delta+k)+\delta n+\\&+\sum_{i=1}^{d-1}\big(\frac{e_{i,\times}}{\ell}n+\frac{e_i}{\ell}(6\delta n+12\delta^2+2k+2n+\delta)+\\&+\frac{2e_{i+1}}{\ell}(3\delta n+3\delta^2+3k+2n+\delta)+7\delta n+32\delta^2+4\delta\ell\big)+\frac{e_d}{\ell}(\delta n+\delta\ell)\end{aligned}$$

- **Elements sent on watchlists:**

$$\frac{e_1}{\ell}n+n+\sum_{i=1}^{d-1}\big(\frac{e_i}{\ell}n+\frac{2e_{i+1}}{\ell}n+3n\big)$$

- **Elements sent in plain:**

$$\frac{e_1}{\ell}(n+1)+\delta+\sum_{i=1}^{d-1}\big(\frac{e_i}{\ell}(4\delta+2n)+\frac{2e_{i+1}}{\ell}(2\delta+n+1)+3\delta+n\big)+\frac{e_d}{\ell}n$$

In addition Alice and Bob multiply some elements using the semi-honest inner multiplication protocol. The number of such multiplications is:

$$\sum_{i=1}^{d-1}\big(2\frac{e_{i,\times}}{\ell}n\big)$$