The Interdisciplinary Center, Herzlia
Efi Arazi School of Computer Science
M.Sc. program - Research Track

Practical Covert Multiparty Computation with Interactive Public Verifiability

by
Eran Goldstein

M.Sc. dissertation, submitted in partial fulfillment of the requirements
for the M.Sc. degree, research track, School of Computer Science
The Interdisciplinary Center, Herzliya

August 2020

This work was carried out under the supervision of Dr. Tal Moran from the Efi Arazi School of Computer Science, The Interdisciplinary Center, Herzliya.

Acknowledgements

I would like to express my gratitude to my advisor, Dr. Tal Moran from the Interdisciplinary Center (IDC). I would also like to thank Prof. Yehuda Lindell from Bar-Ilan University (BIU) for many insightful discussions, and the anonymous reviewers of Asiacrypt 2020 for their comments.

Abstract

Secure multiparty computation (MPC) allows mutually distrusting parties to evaluate a function of their inputs without requiring any party to reveal their input to another.

In the *covert security* setting of MPC, the security guarantee is parameterized by a deterrent factor ϵ — an adversary is willing to cheat only if its probability of being detected is less than ϵ. This models adversarial behavior, in cases where being caught is costly for the adversary.

When our basis for accepting covert security is the cost of getting caught, the honest party should be able to *prove* that cheating was detected. This requirement is captured by the notion of *publicly-verifiable* covert security: honest users can construct a "certificate" proving the adversary's corruption.

Today, in the presence of blockchain protocols, the model of publicly-verifiable covert security becomes even more relevant, because a blockchain can be used to automatically enforce monetary punishments. This gives a strong motivation for optimizing the *verification cost* of the cheating proof. Existing publicly-verifiable covert protocols are not optimized for this metric, and require the verifier to perform complex public-key operations or to simulate the protocol's execution.

In this paper, (1) we formally define *interactive* publicly verifiable covert security, which can allow a broader range of protocols, while still being compatible with blockchain verifiers and (2) construct an efficient protocol for two-party secure computation in this setting in which the verifier is only required to perform signature verification, hashing and bit operations.

To allow our protocol to be used for computing reactive functionalities, we also construct a generic compiler from publicly-verifiable 2PC in the covert model to one that satisfies a stronger security definition, where output-correctness is always preserved, regardless of whether cheating was detected. Our compiler preserves the verification complexity of the base 2PC protocol (up to a constant factor).

Contents

1. Introduction

Secure multiparty computation (MPC) allows mutually distrusting parties to evaluate a function of their inputs without requiring either party to reveal their input to another [18, 10].

In the case of an honest majority, any function can be securely realized with *full security*: even when an adversary can deviate arbitrarily from the honest protocol, honest parties are guaranteed to terminate with correct outputs.

In the dishonest majority case, on the other hand (and in particular in the two-party case), full security is provably impossible [6]. This led to the definition of a weaker security notion, that of *security with abort*, in which an adversary is "allowed" to force honest parties to output $\perp$ instead of the correct output.

Covert Security

In an attempt to improve the efficiency of maliciously-secure (with abort) protocols in the standard model, Aumann and Lindell proposed the definition of *covert security* [3]. Under this definition, the security guarantee is parameterized by a deterrent factor ϵ — an adversary is willing to cheat only if its probability of being detected is less than ϵ. This models weaker, but still realistic, adversarial behavior, in cases where being caught is costly for the adversary.

While covert security does not solve the problem of guaranteeing output delivery, protocols in this setting do have significant efficiency benefits over maliciously-secure protocols.

Covert Security With Public Verifiability

The covert security model allows honest participants to detect cheating attempts with some fixed probability. However, if our basis for accepting covert security is that the adversary is deterred by loss of reputation or monetary cost, *detecting* a cheating attempt may not be enough—the honest party must be able to *prove* that a cheating attempt was detected.

The recently proposed notion of *publicly-verifiable* covert security [1], captures this requirement. Such protocols guarantee that when misbehavior is detected, the honest users can construct a "certificate" proving the adversary's corruption that can be checked by any third-party arbitrator.

Arbitration Via Blockchain

Today, in the presence of blockchain protocols, the model of covert security becomes even more relevant, because a blockchain can be used to automatically enforce monetary punishments when cheating is detected.

Use of the blockchain as an enforcement mechanism is a strong motivation for optimizing the *verification cost* of the cheating proof. This is because use of blockchain computation is expensive (and sometimes limited in the possible operations). Existing publicly-verifiable covert protocols are not optimized for this metric, and require complex public-key operations from the verifier [1, 17] or require the verifier to simulate the protocol's execution [12].

On the other hand, blockchain-based arbitration need not be limited to verifying a certificate. Even Bitcoin, which is one of the most computationally-limited blockchains, supports a notion of *interactive* verification via time-locked scripts.

1.1. Our Contributions

In this paper we attempt to address these issues on both the theoretic and practical fronts:

- **Covert Security with Interactive Public Verification:** We formally define a notion of covert security with public verification to include *interaction*. Our definition, which encompasses non-interactive public verification as a special case, is simpler than previous definitions, but at the same time can be used to show composability properties in a natural way.

- **Covert 2PC with efficient verification:** We construct a 2PC protocol with covert security (based on garbled circuits) that is (interactively) publicly-verifiable with a very efficient verifier.

  We consider two settings. In the *preprocessing* setting, the parties execute an initial preprocessing phase (before they know their inputs), in which aborts are allowed, and then an online phase in which an aborting adversary can be penalized. In the *online-only* setting, parties only execute online.

  In both settings, the proof sizes are independent of the input length and circuit size (we consider separately arbitration for "non-responsiveness"—i.e. when the adversary simply refuses to send any message—the adversary *can* force the non-responsiveness arbitrator to read the entire transcript of the protocol, albeit without performing any processing on it).

  In the preprocessing setting, the arbitrator is only required to perform signature verification, hashing and bit operations (all "cheap" operations in blockchain terms). In the online-only setting, the arbitrator may also be required to perform three group operations in total (in the worst case).

- **One-shot to reactive two-party computation protocol compiler:**

While semi-honest (and maliciously) secure protocols are secure under sequential composition in the standard model, the same does not hold for publicly-verifiable covert security. This is because covert security allows the adversary to change the output of the protocol undetectably with non-negligible probability, hence the output of such protocols are not assured to be valid and cannot be used for future computations. This means, in particular, that the standard compiler from a one-shot computation to an interactive functionality [10] does not work.

To allow our protocols to be used for reactive computation, we construct a generic compiler from publicly-verifiable 2PC in the covert model to one that satisfies a stronger security definition, where output-correctness is always preserved, regardless of whether cheating was detected. This is enough to allow sequential composition, and hence reactive 2PC. Our compiler preserves the verification complexity of the base 2PC protocol (up to a constant factor).

1.2. High-Level Overview of Techniques

Covert Security with Interactive Public Verification

We generalize the definition of publicly verifiable covert security that was presented in [1], s.t. the public verification can be interactive, rather than non-interactive.

We construct a protocol that realizes any two-party functionality under this new security definition. Our protocol is based on garbled circuits. When constructing the circuit, the garbler commits to each component of the garbled circuit separately (i.e., gates, wire labels and wire masks). Each gate encodes not just the wire labels, but also the opening of their corresponding commitments.

In case of a bad circuit (i.e., one which would cause the evaluator to abort), at least one of the gates or input labels must be bad. If the input labels are good, the evaluator will be able to generate a public proof by pointing to a single gate with "good" input labels and a bad output label (that doesn't open the corresponding commitment). If the input labels are bad, we rely on an underlying OT protocol that has provable output values (i.e., the honest receiver can prove that its output is a result of the OT execution).

The OT protocol is where the two settings differ. In the preprocessing model, we can make black-box use of a random-OT protocol (which can be computed very efficiently using OT extension [2]), and verification of the OT output requires only hash operations. (See chapter 12 for details)

For the online-only case, on the other hand, we require the OT protocol itself to support output proofs. We construct an OT protocol with the desired properties based on the protocol of Doerner, Kondi, Lee, and abhi shelat [9], which is secure against malicious adversaries (but allows selective aborts). Their protocol uses ZK proofs in order to ensure the sender doesn't cheat, which are computationally expensive to verify. We replace the use of ZK proofs with a simpler mechanism whose verification requires only hashing, bitwise operations and at most three group operations. (See chapter 9 for details)

Compiling Covert to Covert With Output-Correctness

Building on our construction above (in a black-box fashion), we show how to construct a protocol that realizes any two-party functionality with covert security guarantees for privacy, but unconditional output correctness (while retaining the verifiability properties and efficiency).

The main technique is the use of the dual-execution paradigm [16], in which parties run two symmetric executions of the protocol. This is similar to the technique of Canetti, Riva, and Rothblum [4], but simplified and in a circuit setting. In a garbled circuit protocol the evaluator can't cheat, which means that in a dual-execution protocol the adversary won't be able to cheat in one of the executions (i.e., the one where she acts as evaluator). Therefore, any attempt to cheat and change the output in that execution will be detected because it will be different from the output in the other execution, where she *can't* cheat.

Running two independent instances of the one-shot protocol opens another avenue for attack however—the adversary can submit different inputs to each instance, and thus potentially get more information about the honest user's input, while not risking detection with sufficient probability to satisfy covert security.

To prevent this attack, we slightly modify the underlying one-shot functionality. Our new functionality computes, in addition to the desired output, pairwise-independent hashes of the parties' inputs, in such a way that any modification to the input will change the output with sufficient probability. While this does add some overhead to the secure computation, we can use information-theoretic hashes that have extremely efficient circuit implementations, so the overhead is quite small. (See chapter 3 for details of our construction)

Handling Non-Responsive Adversaries

In order to satisfy our security definition, we must prevent adversaries from aborting in the middle of a protocol's execution (e.g., in order to avoid having to send a message that implicates them in malfeasance).

However, honest parties cannot construct a proof that a message is *not* sent (otherwise the adversary could simulate not receiving an honest message, and thus frame the honest user).

We'll call an adversary that refuses to send messages (during an execution of a protocol) a *non-responsive* adversary.

To handle such adversaries, we consider a separate "non-responsiveness" arbitrator (NRA), that may differ from the "correctness" arbitrator (CA) used to adjudicate other types of complaints.

In practice, these could be implemented by the same smart contract, but due to the different verification requirements it may be useful to implement them separately (e.g., the NRA could be implemented on a completely separate blockchain). In all cases, we assume that any message sent to either of the arbitrators is also broadcast to all parties.

The non-responsiveness arbitrator is only responsible for ensuring parties communicate

some message in each round. Thus, if the adversary refuses to send a message in round i, the honest party can complain to the NRA and specify the identifier of the missing message. When one party complains, the NRA will accept any (signed) message from the other party with index i, as long as it was delivered before a predefined timeout has elapsed (after which the non-responsive party is penalized). Thus, in the worst case the adversary can cause the honest party to send all protocol messages through the NRA, but the the NRA is generic (does not depend on the functionality being computed) and only verifies the parties' signatures.

Even though it seems like a trivial task, in practice the adversary might send false complaints to the NRA about messages that he already received, messages that he wasn't supposed to receive at the given state of the execution, or even a message that doesn't exists (e.g., a message that is "expected to be sent" after the protocol's execution is finished). (For more details, see chapter 15)

In contrast, the CA is responsible for ensuring the correctness of the messages transmitted in the protocol. This arbitrator is protocol-specific. The goal in designing this type of protocol is that if an adversary is responsive, an honest party can always convince the CA of malfeasance with a small amount of communication and low computational complexity.

1.3. Related Work

Covert security was introduced by Aumann and Lindell [3]. The efficiency improvements made possible by this new security notion led to a plethora of follow-up work such as [5, 8, 1, 15, 14], extending their definition to a multi-party setting, building compilers for constructing covert-secure protocols, extending covert security to include public verification.

Publicly-Verifiable Covert Computation

The definition of publicly verifiable covert (PVC) security was introduced by Asharov and Orlandi [1]. In follow-up work, Kolesnikov and Malozemoff [14] optimized the efficiency of the verification process by constructing a verifiable OT extension.

A recent paper by Hong, Katz, Kolesnikov, Lu, and Wang [12] constructs a PVC-secure protocol with a constant-size non-interactive proof — in terms of public verifiability their implementation mainly focus on optimizing the proof's size, which is constant, but it is inefficient w.r.t. verification time. Their protocol makes use of parallel execution of multiple semi-honest protocols, combined with cut-and-choose where the verification process involves checking correctness of all-but-one *executions*. In order to verify each execution, the garbler sends the randomness he used in each execution (which he commits to, prior to sending it) and then the evaluator is able to check whether the transcript correspond to the messages generated using the provided randomness. Hence, in any case of a *detected* corrupt behavior one would be able to prove it by simply comparing an simulated execution of the protocol (using the parties' randomness) and the actual transcript and see which party deviated from the protocol.

Dual-Execution and Output Correctness

The paradigm of dual execution was introduced by Mohassel and Franklin [16], which addressed the problem with the all-or-nothing guarantee provided by cut-and-choose based protocols. They constructed a two-party protocol that guarantees correctness and leaks one bit of information to a malicious adversary — the parties execute two symmetrical executions of a semi-honest garbled-circuit protocol, and compare their output using an equality-checking functionality (implementation details of this functionality were not provided).

Mohassel and Riva [17] introduced a variant of covert security that guarantees output correctness and at most a single bit of leakage in case of undetected cheating, based on the standard definition presented in Aumann and Lindell [3]. Their protocol involves dual execution of the semi-honest Yao's garbled circuit protocol and evaluating, in addition, several circuits, used to enforce input consistency in both executions. Finally, they make use of an equality checking functionality (which they defined in the paper) in order to compare the outputs obtained in the former dual-execution protocol. Their protocol uses zero-knowledge-proofs-of-knowledge (ZKPoK), therefore making such protocol publicly verifiable would result in an inefficient verification time.

In our protocol, in contrast, the amount of leakage is *not* bounded when cheating was not detected, but we leverage this to get better efficiency.

Huang, Katz, and Evans [13] also constructed an optimized implementation of the protocol introduced by Mohassel and Franklin [16], including a construction of an equality testing protocol, which is used in their protocol as well.

2. The Model

2.1. Preliminaries

A function $\mu(\cdot)$ is negligible in n, or just *negligible*, if for every positive polynomial $p(\cdot)$ and all sufficiently large ns it holds that $\mu(n) < 1/p(n)$. A probability ensemble $X = \{X(a,n)\}_{a\in\{0,1\}^*;n\in\mathbb{N}}$ is an infinite sequence of random variables indexed by a and $n \in \mathbb{N}$. (The value a will represent the parties inputs and n the security parameter.) Two distribution ensembles $X = \{X(a,n)\}_{a\in\{0,1\}^*;n\in\mathbb{N}}$ and $Y = \{Y(a,n)\}_{a\in\{0,1\}^*;n\in\mathbb{N}}$ are said to be computationally indistinguishable, denoted $X \stackrel{c}{\equiv} Y$, if for every non-uniform polynomial-time algorithm D there exists a negligible function $\mu(\cdot)$ such that for every $a \in \{0,1\}^*$ and every $n \in \mathbb{N}$:

$$\left| \Pr\left[D(X(a,n)) = 1\right] - \Pr\left[D(Y(a,n)) = 1\right] \right| \leq \mu(n)$$

2.2. Covert Security with Interactive Arbitration

This definition generalizes *covert security with public verification* (Asharov and Orlandi [1]) by making it a special case of *covert security with ϵ-deterrence* (Aumann and Lindell [3], for completeness we include the definition in appendix A.2).

The definition of *covert security with public verification* involves an additional party who judges which party, out of the n participating parties, should be accused in a case of an abort (this party is defined implicitly, as a routine for resolving complaints in case of an abort). Under this definition, a complaint is a *single* message (which is referred as a *certificate* in the paper).

Instead of implicitly defining the arbitrator, our definition *explicitly* adds the arbitrator as an extra party; this party has no input and receives no output in an honest execution. However, the standard definition of covert security ensures that the arbitrator *will* output the id of the cheating party when cheating is detected.

Moreover, this definition abstracts the amount of interaction with the arbitrator — protocols may specify arbitration routines that require arbitrary message exchanges with the arbitrator.

Formally, let $\mathcal{F}$ an n-party functionality. We denote $\mathcal{F}_{extended}$ as the $(n+1)$-party functionality that adds an extra arbitrator as party $n+1$, using $\mathcal{F}$ internally as follows:

- $\mathcal{F}_{extended}$ ignores messages sent from party P_{n+1}.
- Each input message sent from party P_i (for all $i \in [1,n]$) is sent to $\mathcal{F}$.
- Each output message destined to party P_i (for all $i \in [1,n]$) is sent to P_i.

Definition 2.2.1 (Covert Security with Interactive Arbitration)**.** A protocol Π realizes $\mathcal{F}$ with *ϵ-covert security with interactive arbitration* (ϵ-CIA security) iff Π realizes $\mathcal{F}_{arb}$ with *covert security with ϵ-deterrence.*

Note that even though $\mathcal{F}_{extended}$ ignores P_{n+1} — i.e. neither sends him any messages, nor process any of his input messages — the definition of *covert security with ϵ-deterrence* specifies that *all* parties receive the output in case of a detected cheating attempt, including party P_{n+1}.

2.3. Covert Security with Public Interactive Arbitration

In the context of covert security, we emphasized the need for *public verifiability*: a protocol realizing covert security doesn't have much of a deterrence where the only entities able to prove the adversary's corruption are the honest participants. While the definition of *covert security with public verification* (Asharov and Orlandi [1]) realizes this property, it actually requires publicly verifiable *non-interactive* arbitration, which consists of a single message (i.e. certificate) presented by the complaining party. The arbitration in our protocol is interactive, rather than non-interactive, in order to simplify the arbitration routine.

The definition of *Covert Security with Interactive Arbitration* (c.f. definition 2.2.1) doesn't necessarily realize *public* verifiability, because protocols realizing this definition may include private interaction between the arbitrator and a subset of the participants, as part of the arbitration. However, if the interaction with the arbitrator in the protocol is strictly public (e.g. a broadcast channel), such a protocol does realize public verifiability. Therefore, public verifiability can be described as a syntactic property of protocols realizing *Covert Security with Interactive Arbitration* (definition 2.2.1).

Definition 2.3.1 (Covert Security with Public Interactive Arbitration)**.** A protocol Π realizes functionality $\mathcal{F}$ with *ϵ-covert security with public interactive arbitration* iff Π realizes functionality $\mathcal{F}$ with ϵ-CIA security and the following holds:

1. All messages addressed to the arbitrator are sent using *a broadcast channel* (i.e. the arbitrator only receives messages), and finally the arbitrator outputs the id of the corrupt party
2. The arbitrator doesn't send messages
3. The arbitrator doesn't have a secret state.

Note that *covert security with public verification* is a special case of definition 2.3.1 where the arbitration consists of a single message.

2.4. Verifiable Covert Security with Abortable Preprocessing

In definition 2.2.1, the arbitrator will catch an adversary that aborts the computation, even if the abort happens at the very beginning of the protocol (e.g., before the first

protocol message is sent). This level of security might be overkill for some applications, and is infeasible to realize in many settings (for example, an honest user can't hope to generate a verifiable "certificate" proving that the adversary did not send any messages).

We define a weaker version of security that allows *unverifiable* aborts in a preprocessing phase (before inputs are submitted).

Formally, we define an "abortable preprocessing" version of an arbitrary functionality $\mathcal{F}$

Definition 2.4.1 (Abortable Preprocessing)**.** For a subset I of the participating parties, we define $\text{abortpre}_I(\mathcal{F})$ to be the following functionality:

1. Wait to receive (**done-preprocessing**) from all parties in subset I. If any party in I aborts during this step, send $\perp$ to all parties and abort.
2. Send **start-online** to all parties in I.
3. Execute functionality $\mathcal{F}$.

The abortable version of Covert Security with Interactive Arbitration can now be defined:

Definition 2.4.2 (Covert Security with Interactive Arbitration and Abortable Preprocessing)**.** A protocol Π realizes $\mathcal{F}$ with *ε-covert security with interactive arbitration and abortable preprocessing* iff Π realizes $\text{abortpre}(\mathcal{F})$ with ε-CIA.

Note that since the adversary is explicitly "allowed" to abort in the preprocessing stage of $\text{abortpre}(\mathcal{F})$, the covert security definition doesn't require every honest party to output **cheat-detected** in this case (thus, the arbitrator will not output anything, corresponding to an "unverifiable" abort).

Covert security with *public* interactive arbitration and abortable preprocessing is defined analogously.

Our protocols are in the online setting (without preprocessing). However, by replacing our online OT primitive with an abortable preprocessing version (which can be verified much more efficiently), we get a corresponding protocol that satisfies the abortable preprocessing security definition.

We make use of the following lemma to show that this can be done generically (i.e., given any protocol that is secure in the online setting, replacing the underlying primitives with their preprocessing versions will give a protocol secure in the preprocessing model).

Let Π be a protocol that realizes $\mathcal{F}$ in the $\mathcal{G}$-hybrid model. We define Π' to be the following protocol in the $\text{abortpre}_I(\mathcal{G})$-hybrid model, where I is a subset of the parties (also, denote $\overline{I} = \{1, ..., n\} \setminus I$).

The protocol Π' is defined as follows:

1. If the party participates in the preprocessing phase: (i.e., if $i \in I$)
   a) Send **done-preprocessing** to all instances of $\text{abortpre}_I(\mathcal{G})$
   b) Wait to receive message **start-online** from all instances of $\text{abortpre}_I(\mathcal{G})$. If any instance aborted, then abort.

2. Execute Π — replacing every call to an instance of $\mathcal{G}$ with the corresponding instance of $\text{abortpre}_I(\mathcal{G})$.

Lemma 2.4.3. *If Π realizes $\mathcal{F}$ in the $\mathcal{G}$-hybrid model with malicious security, and the parties in $\overline{I}$ don't receive output if the parties in I don't send any messages, then Π' realizes $\text{abortpre}_I(\mathcal{F})$ in the $\text{abortpre}_I(\mathcal{G})$-hybrid model with malicious security.*

Proof Sketch. Π realizes $\mathcal{F}$, therefore exists a PPT $\mathcal{S}$ that generates a view which is indistinguishable from a real world execution.

Let n_g be the amount of instances of $\mathcal{G}$ used in protocol Π, and denote $\mathcal{S}'$ the following ideal world simulator of Π':

1. If the corrupt party participates in the preprocessing phase:
   a) Until receiving from the corrupt party **done-preprocessing** messages for all instances of $\text{abortpre}(\mathcal{G})$, do the following upon receiving a new message sent to $\text{abortpre}_I(\mathcal{G})_j$ (for any $j \in [1, n_g]$):
      - If the corrupt party sent message **abort-preprocessing** to $\text{abortpre}_I(\mathcal{G})_j$, send **abort-preprocessing** to $\text{abortpre}(\mathcal{F})$ and simulate the honest parties' abort.
      - If the corrupt party sent message **done-preprocessing** to $\text{abortpre}_I(\mathcal{G})_j$, simulate a message **start-online** sent from $\text{abortpre}_I(\mathcal{G})_j$ to the corrupt party.
   b) Send **done-preprocessing** to $\text{abortpre}(\mathcal{F})$, and simulate a message **start-online** to the corrupt party from all instances of $\text{abortpre}_I(\mathcal{G})$
2. Execute $\mathcal{S}$.

First, note that the execution of protocol Π in the real world is independent from the messages sent in the preprocessing phase, this allows us to execute the simulator $\mathcal{S}$ in the ideal world simulator $\mathcal{S}'$. Therefore, the view generated by executing Π' is identical to the view of Π in addition to a constant-sized prefix of messages sent in the preprocessing phase.

We will consider two cases: (1) the adversary cheated in the preprocessing phase, and (2) the adversary didn't cheat in the preprocessing phase.

Considering the case where the adversary cheats in the preprocessing phase (w.l.o.g. the adversary sent **abort-preprocessing** to $\text{abortpre}(\mathcal{G})_j$, for some $j \in [1, n_g]$).

In the real world, the output of the parties in I is **abort-preprocessing**, therefore the parties in $\overline{I}$ receive output $\perp$ (we assumed that the parties in $\overline{I}$ don't receive output if the parties in I don't send any messages). In the ideal world, $\mathcal{S}'$ sends **abort-preprocessing** to $\text{abortpre}(\mathcal{F})$ and then simulates the honest parties' abort — the output for the parties in I is **abort-preprocessing**, and the output for the parties in $\overline{I}$ is $\perp$.

Therefore, in this case, the real and ideal worlds' views are identical.

Considering the case where the adversary doesn't cheat in the preprocessing phase — i.e., the adversary sent **done-preprocessing** to all instances where the corrupt party

participates in the preprocessing phase. In the real world, the corrupt party receives the **start-online** from all instances of abortpre($\mathcal{G}$) where he participates in the preprocessing phase. The rest of the corrupt party's view, and the honest party's output, are obtained by executing protocol Π. In the ideal world, $\mathcal{S}'$ simulates a messages **start-online** sent from all instances of abortpre($\mathcal{G}$) where the corrupt party participates in the preprocessing phase. The rest of the corrupt party's view, and the honest party's output, are obtained by executing simulator $\mathcal{S}$.

The corrupt party's view in the processing phase is identical in real and ideal worlds. Hence, the views generated in the real and ideal worlds are indistinguishable iff the views generated by Π and $\mathcal{S}$ are indistinguishable.

$\mathcal{S}$ generates views that are indistinguishable from a real world execution of Π, by definition, and therefore the views generated by Π' and $\mathcal{S}'$ are indistinguishable in that case as well. □

Lemma 2.4.4. *For any n-party functionality $\mathcal{F}$, where $I = \{1, ..., n\}$, the following holds:*

$$CIA_{\varepsilon}(abortpre_I(\mathcal{F})) = abortpre_I(CIA_{\varepsilon}(\mathcal{F}))$$

Proof Sketch. $\text{abortpre}_I(\mathcal{F})$ is an n-party functionality where the parties $P_1, ..., P_n$ participate in the preprocessing phase. $\text{CIA}_{\varepsilon}(\text{abortpre}_I(\mathcal{F}))$ is an $(n+1)$-party functionality where $P_1, ..., P_n$ participate in the preprocessing phase, P_{n+1} has no inputs and the adversary receives the honest parties' inputs w.p. ε.

$\text{CIA}_{\varepsilon}(\mathcal{F})$ is an $(n+1)$-party functionality where P_{n+1} has no inputs and the adversary receives the honest parties' inputs w.p. ε. $\text{abortpre}_I(\text{CIA}_{\varepsilon}(\mathcal{F}))$ is an $(n+1)$-party functionality where P_{n+1} has no inputs and the adversary receives the honest parties' inputs w.p. ε, and parties $P_1, ..., P_n$ participate in the preprocessing phase.

As it can be observed, the functionalities $\text{CIA}_{\varepsilon}(\text{abortpre}_I(\mathcal{F}))$ and $\text{abortpre}_I(\text{CIA}_{\varepsilon}(\mathcal{F}))$ are identical. □

3. One-Shot Protocol: Overview

In this chapter we describe our covert publicly-verifiable two-party computation protocol at a high level (the full formal description appears in chapter 4).

Our protocol is based on Aumann and Lindell [3], where two parties are involved: a garbler, who generates garbled circuits that will be used for verification and evaluation; and an evaluator, using cut-and-choose for verifying all circuits except one, and evaluating the remaining circuit using input labels provided by the garbler. This protocol doesn't provide public verifiability, and requires several changes in order to make it such.

First, we add signatures to each message transferred between the parties — a third party can be easily convinced that a message was originated from another party P when attaching P's signature on the corresponding message. Even by doing this single change, we get a publicly-verifiable protocol, but it has one big flaw — certificates proving a corrupt behavior might be very big. Considering a garbler who sent a garbled circuit with an invalid gate, or a corrupt evaluator who evaluated a corrupt output, a verifier needs the *entire* circuit and the input labels (and the garbler's signatures of the corresponding components) in order to check if a corrupt gate exists or the evaluated output is wrong. Such a certificate is inefficient in terms of space (linear w.r.t. the circuit's size) and validation time (at worst case, requires evaluating the entire circuit).

In order to make the certificate compact, a garbler will commit to each component in the garbled circuit *separately*. Thus, each party gets assurance about the evaluation process:

- From the evaluator's point of view, proving a circuit's invalidity requires only a corrupt gate and two valid input labels, and the verifier only needs to check if the gate actually outputs an invalid output label, using the commitments of the corresponding wire's labels.
- From the garbler's point of view, proving that the output labels are invalid requires only a single invalid label and the commitments of the corresponding wire's labels. The verifier only needs to check if the given label isn't a valid opening of both commitments.

However, the modified protocol is still vulnerable against corrupt parties who cheat during OT, or a corrupt garbler sending invalid inputs to OT — trying to guess the evaluator's input (i.e. selective-failure attack).

To deal with a cheating sender, the honest party must be able to publicly prove that the output was actually received from the OT functionality. Therefore, we require an OT protocol with *publicly verifiable* output (PVOT).

In order to deal with the selective-failure attack, the evaluator's input is masked is order to prevent such attack (hence guessing the value of an input bit is as good as guessing the mask's value).

We construct a PVOT protocol in chapter 9 that is publicly-verifiable with similar certificate guarantees (small size and efficiently validated).

In addition, we define an alternative protocol in the preprocessing model (c.f. chapter 12), which guarantees the same security, a more efficient public verification process, and allows black-box use of any maliciously-secure random-OT protocol in the preprocessing phase, allowing efficient use of OT extensions.

3.1. Ideal functionalities $\mathcal{F}_{\varepsilon-\text{one-shot}}$ and $\mathcal{F}_{PVOT}$

The functionality we want to realize is $\mathcal{F}_{\varepsilon-\text{one-shot}}$, which realizes $\mathcal{F}_f$ with ε-CIA security, with the additional guarantees that only P_0 can attempt cheating, and only before sending his inputs. Also, P_1 can ask $\mathcal{F}_{\varepsilon-\text{one-shot}}$ to send his input to all participants. The formal description of the functionality appears in fig. 3.1.1.

Figure 3.1.1.: Functionality $\mathcal{F}_{\varepsilon-\text{one-shot}}$

Inputs: Party P_i sends a message $(\textbf{input}, x_i)$ (for his input x_i) to $\mathcal{F}_{\varepsilon-\text{one-shot}}$, while $\mathcal{A}$ sends input on behalf of the corrupted party.
Input commitment: Upon receiving message $(\textbf{input}, x_0)$ from P_0, send message (**committed**) to both parties.
Attempted Cheat: A cheating attempt is not possible after sending **committed** message. If $\mathcal{F}_{\varepsilon-\text{one-shot}}$ receives (**cheat**) from a corrupt party P_i, the honest parties' inputs are sent to $\mathcal{A}$. Then, $\mathcal{F}_{\varepsilon-\text{one-shot}}$ decides randomly if the cheat was detected or not:

- **Undetected:** With probability $1-\varepsilon$, $\mathcal{F}_{\varepsilon-\text{one-shot}}$ sends (**cheat-undetected**) to $\mathcal{A}$. Then, $\mathcal{A}$ specifies the output that'll be provided by $\mathcal{F}_{\varepsilon-\text{one-shot}}$ to the honest party (after sending "reveal").
- **Detected:** With probability ε, $\mathcal{F}_{\varepsilon-\text{one-shot}}$ sends $(\textbf{cheat-detected}, P_i)$ to all parties.

Output generation: Upon receiving (**reveal**) from P_0 and P_1 (following **input** messages from both parties), in case of no cheating, $\mathcal{F}_{\varepsilon-\text{one-shot}}$ computes the *authenticated* output $y = f(x_0, x_1)$ and sends (**output**, y) to P_0, P_1.
P_1**'s input correctness proof:** Party P_1 sends (**prove-input**) command to $\mathcal{F}_{\varepsilon-\text{one-shot}}$, and then $\mathcal{F}_{\varepsilon-\text{one-shot}}$ sends $(\textbf{prove-input}, x_1)$ to all parties.
Output correctness proof: Party P_1 sends $(\textbf{prove-output}, w)$ command to $\mathcal{F}_{\varepsilon-\text{one-shot}}$, and then $\mathcal{F}_{\varepsilon-\text{one-shot}}$ sends $(\textbf{prove-output}, v_w)$ to all parties, where v_w is the evaluated value of output-wire w

We will make use of the functionality $\mathcal{F}_{PVOT}$, defined in fig. 3.1.2. The functionality $\mathcal{F}_{PVOT}$ extends $\mathcal{F}_{OT}$ with provable output. The probability of detecting a cheating attempt depends on the adversary's prior knowledge about the receiver's input (so this functionality does not satisfy standard covert security).

Figure 3.1.2.: Functionality $\mathcal{F}_{PVOT}$

Choose: On receiving $(\textbf{choose}, \omega)$ from the receiver, store $(\textbf{choice}, \omega)$ if no such message exists in memory and send $(\textbf{chosen})$ to the sender.
Guess: On receiving $(\textbf{guess}, \hat{\omega})$ from the sender, if $\hat{\omega} \in \{0, 1, \perp\}$ and if $(\textbf{choice}, \omega)$ exists in memory, and if $(\textbf{guess}, \cdot)$ does not exist in memory, then store $(\textbf{guess}, \hat{\omega})$ in memory and do the following:

- If $\hat{\omega} = \perp$, send $(\textbf{no-cheat})$ to the receiver.
- If $\hat{\omega} = \omega$, send $(\textbf{cheat-undetected})$ to the sender and $(\textbf{no-cheat})$ to the receiver.
- Otherwise, send $(\textbf{cheat-detected})$ to both the sender and receiver.

Transfer: On receiving $(\textbf{transfer}, \alpha_0, \alpha_1)$ from the sender, if $\alpha_0 \in \mathbb{Z}_q$ and $\alpha_1 \in \mathbb{Z}_q$ and if $(\textbf{complete})$ does not exist in memory, and if there exist in memory messages $(\textbf{choice}, \omega)$ and $(\textbf{guess}, \hat{\omega})$ such that $\hat{\omega} = \perp$ or $\hat{\omega} = \omega$, then send $(\textbf{message}, \alpha_\omega)$ to the receiver and store $(\textbf{complete})$ in memory.
Prove output: On receiving $(\textbf{prove-output}, p)$ from the receiver, if $(\textbf{complete})$ is stored in memory then send α_ω to party p.

3.2. Garbler's routine

The garbler generates two garbled-circuits (denoted GC_0, GC_1) based on the circuit C_f, and then commits to the circuits.

Upon receiving the index of the to-be-verified garbled-circuit (denoted b) the garbler reveals GC_b to the evaluator, this garbled-circuit will be used only for correctness verification. As for the evaluation process, the garbler reveals the gates of GC_{1-b}, as well as the required input-values.

Then, when he receives the output-labels' decommitments from the evaluator, he verifies they are valid decommitments of output-wires' labels — if all decommitments are valid, the garbler computes the plain output, and reveals the masks corresponding to the output-wires.

Otherwise, i.e. one of the decommitments is invalid, he complains to the arbitrator by presenting him the evaluator's message containing the invalid decommitment.

The full specification of the Garbler's protocol appears in protocol $\Pi^{OS}_{garbler}$ [2.1].

3.3. Evaluator's routine

The evaluator waits to receive the garbler's commitments on the garbled-circuits, and then chooses (randomly) one of the circuits to be verified (denoted by index b) and the other one to be evaluated.

The garbler reveals garbled-circuit GC_b (that will be verified), in addition to the gates of the garbled-circuit GC_{1-b} that will be used for the evaluation process.

Using the garbled-circuit GC_b, the evaluator checks its correctness and consistency w.r.t. the received commitments (by *correctness* we refer to the claim that each gate should implement a valid garbled NAND-gate), and in any case of failure notifies the arbitrator with an appropriate complaint proving the garbler's corruption.

In case the verification process has been completed successfully, the evaluator receives the input values for the evaluation circuit via OT, and uses them in order to evaluate the circuit's output labels' decommitments, and send the evaluated values back to the garbler.

Finally, the garbler reveals the wire bits corresponding to each output wire, implying the plain value corresponding to each label that was computed previously in the evaluation process. The full specification of the Evaluator's protocol appears in protocol $\Pi^{OS}_{evaluator}$ [2.3].

3.4. Security Analysis

Our main theorem is as follows:

Theorem 3.4.1. *Π^f_{OS} realizes functionality $\mathcal{F}_{\frac{1}{2}-one\text{-}shot}$ with malicious security.*

By observation, one can confirm that $\mathcal{F}_{\varepsilon-\text{one-shot}}$ is an extension of $\mathcal{F}_{extended}$ from the CIA-security definition (it is more restrictive, since $\mathcal{F}_{\varepsilon-\text{one-shot}}$ allows only one of the parties to attempt cheating, and only before sending his input). Therefore theorem 3.4.1 implies the following:

Corollary 3.4.2. *For any two-party function f, Π^f_{OS} realizes f with $\frac{1}{2}$-covert security and public interactive arbitration.*

Proof Sketch of theorem 3.4.1

The full proof of theorem 3.4.1 appears in chapter 5. Below we give a high-level overview of our security proof.

We describe a simulator $\mathcal{S}$ that runs $\mathcal{A}$ internally and interacts with the trusted party that computes f. We consider the two possible corruption scenarios separately (garbler and evaluator).

Corrupt garbler: $\mathcal{S}$ extracts both garbled-circuits when the garbler sends their commitments to the evaluator. Then, he checks whether these circuits were built correctly or not:

Case 1: **(both circuits are invalid)** there's at least one invalid component in each circuit, and therefore an honest evaluator will trigger a complaint in the verification phase (no matter which circuit will be opened). Hence, $\mathcal{S}$ simulates an abort for this case.

Case 2: **(both circuits are valid)** this means the verification process of any of these circuits won't trigger a complaint in a real execution and therefore any choice of b won't trigger a complaint of the honest party (and hence b can be chosen uniformly), and continues the simulation.

Case 3: **(one circuit is valid and the other is invalid)** in the real world, the invalid circuit will be revealed w.p. $\frac{1}{2}$, which will trigger a complaint from the honest party. Hence, in the ideal world, $\mathcal{S}$ performs a coin-toss for deciding whether to detect the cheating attempt or not – for detecting the cheating attempt, $\mathcal{S}$ asks the garbler to open the invalid circuit (triggering an abort in the verification phase in a real execution) and finally simulates an abort. For not detecting the cheating attempt, $\mathcal{S}$ asks to open the valid circuit, and continues as follows.

During evaluation, each bit of the evaluator's input is masked using mask $m \xleftarrow{\$} \{0,1\}$ which is picked by the evaluator. Hence, the evaluator's input to the OT functionality is uniform and therefore a garbler's attempt to send bad inputs (this is a selective-failure attack) fails w.p. $\frac{1}{2}$.

Due to the fact that m isn't revealed, unless the evaluator asks the OT functionality to prove the value he received for m, the $\mathcal{S}$ has the flexibility to postpone setting the value of m to the case where a selective failure attack attempt is made (or set $m \xleftarrow{\$} \{0,1\}$ in case of not cheating at all during OT). Setting the value of m is indistinguishable from a real view, where the value of m is picked at random, because $\mathcal{S}$ sets its value according to a coin-toss made by $\mathcal{F}_{\frac{1}{2}-\text{one-shot}}$ and the value of m is identically-distributed in both cases.

Next, $\mathcal{S}$ extracts the garbler's input by using the input-labels sent to the evaluator, and the corresponding wires' labels commitments that were sent earlier — if there's an invalid label (i.e. a decommitment which is invalid w.r.t. its corresponding commitment) an honest evaluator will abort while checking the input labels' correctness, hence $\mathcal{S}$ simulates an abort. Otherwise, $\mathcal{S}$ extracts the garbler's input $x_{garbler}$ using the sent labels and the corresponding wires' masks and labels (as extracted in the beginning of the simulation).

Then $\mathcal{S}$ obtains the output y by querying $\mathcal{F}_{\frac{1}{2}-\text{one-shot}}$ using the garbler's input $x_{garbler}$, and simulates the evaluator's response by using the output-wires' labels corresponding to output y. This is indistinguishable from the real world, because that the evaluated output (in the real world), in case of no cheating attempt, is the evaluation of f using the parties' inputs, which is *by definition* the output obtained in the ideal world from the ideal functionality. Therefore the simulated response in both worlds are identical, as they contain the same output labels.

Corrupt evaluator: $\mathcal{S}$ is able to perfectly-simulate the garbler's protocol until the start of the evaluation phase — he doesn't hold the garbler's inputs, and therefore can't

simulate sending a valid garbled-circuit and the "real" garbler's input labels. Instead, $\mathcal{S}$ will send a circuit that outputs a constant output which will be set to the functionality's output, given the corrupt receiver's (extract) input.

$\mathcal{S}$ extracts the evaluator's input $x_{evaluator}$ by observing the values sent to $\mathcal{F}_{PVOT}^{w}$ (for each wire w in the evaluator's input-wires set) and then queries $\mathcal{F}_{\frac{1}{2}-\text{one-shot}}$ for the output y.

Instead of opening the evaluation circuit, $\mathcal{S}$ uses the commitment scheme's equivocability property in order to open a circuit that outputs the hardcoded output y. This circuit is indistinguishable from the original circuit because of the garbled circuit's semantic security.

4. Full Description of the One-Shot Protocol

In this chapter we give a full, formal description of our protocol to realize, for any function f, the functionality $\mathcal{F}_f$ with $\frac{1}{2}$-Covert security and public interactive arbitration.

4.1. Notations

A **gate** g is defined by a 4-tuple: $tuple_g = (w_0^g, w_1^g, w_{out}^g, tt^g)$, denoting the gate's input garbled-wires, output garbled-wire and truth-table.

A **garbled-wire** w is defined by a tuple $w = (l_0^w, l_1^w)$, denoting the wire's labels. A label's position in the tuple doesn't imply it's plain value.

A garbled-wire's **mask** m_w indicates the plain-value of each label of the garbled-wire w. Specifically, the labels $l_{m_w}^w, l_{\overline{m_w}}^w$ correspond to the values 0, 1 respectively.

For a given circuit C_f that evaluates some function f, denote a **garbled-circuit** as $GC = \big(Gates, Labels, Masks\big)$, s.t.:

- $Gates$ is the set of gates (as defined above) consisting the garbled-circuit, defined by: $Gates = \{g \mid \text{g is a gate in } C_f\}$
- $Labels$ is the set of wire-labels (as defined above) corresponding to each wire in the garbled-circuit, defined by: $Labels = \{l_i^w \mid i \in \{0,1\}, \text{w is a wire in } C_f\}$
- $Masks$ is the set of wire-masks (as defined above) corresponding to each garbled-wire in the garbled-circuit, defined by: $Masks = \{m_w \mid \text{w is a wire in } C_f\}$

$W_{garbler}, W_{evaluator}$ — the sets of garbler's input-wires and evaluator's input-wires (respectively)

$I_p = \{v_w \mid w \in W_p\}$ — set of input values of party p, where $p \in \{garbler, evaluator\}$

4.2. One-shot protocol main subroutines

The garbler's main routine is described in protocol $\Pi_{garbler}^{OS}$ [2.1], where protocol $\Pi_{\text{garbler-rebuttal}}^{OS}$ [2.2] is invoked if the other party broadcasts a complaint.

The evaluator's main routine is described in protocol $\Pi_{evaluator}^{OS}$ [2.3], where protocol $\Pi_{\text{evaluator-rebuttal}}^{OS}$ [2.4] is invoked if the other party broadcasts a complaint.

The arbitrator's main routine is described in protocol Π_{arb}^{OS} [2.5]. It uses protocol HandleOSComplaint [2.6] for handling the parties' complaints (if such are published).

Protocol $\Pi^{OS}_{garbler}$ **[2.1]** Garbler's main routine

function $\Pi^{OS}_{garbler}$
  // Initialize garbled-circuits
  for $i \in \{0,1\}$ **do**
    $GC_i = (Gates_i, Labels_i, Masks_i) \leftarrow$ GenerateGarbledCircuit(C_f)
  end for
  $X \leftarrow (GC_0, GC_1)$
  $Comms \leftarrow Commit(X; r)$ for randomness r
  Send $Comms$ to the evaluator

  Wait to receive bit b from the evaluator // Indicating the verification-circuit's index

  // Create decommitments of garbled-circuits
  $(GC_b, \pi_{GC_b}) \leftarrow$ OpenItem$(X, Comms, (\textbf{circuit}, b))$
  $(Gates_{1-b}, \pi_{Gates_{1-b}}) \leftarrow$ OpenItem$(X, Comms, (\textbf{gates}, 1-b))$

  // Send verification-circuit and the corresponding set of possible wire-masks
  Send (GC_b, π_{GC_b}) to the evaluator

  // Send evaluation-circuit and the required input-wires values
  Send $(Gates_{1-b}, \pi_{Gates_{1-b}})$ to the evaluator
  Execute SendInputLabelsDecommitments$(I_{garbler}, W_{garbler}, W_{evaluator}, Comms, Labels_{1-b}, M$
// c.f. protocol SendInputLabelsDecommitments [2.2]

  // Verify output-labels' correctness
  Wait to receive output-labels decommitments *outputLabelsDecomms*
  $O_{values} \leftarrow$ VerifyOutputLabelsCorrectness$(Comms, outputLabelsDecomms)$ // c.f. protocol VerifyOutputLabelsCorrectness [2.1]
  if O_{values} is a complaint **then**
    Broadcast complaint O_{values}
    return $\perp$
  end if
  return O_{values}
end function

Protocol $\Pi^{OS}_{\textbf{garbler-rebuttal}}$ **[2.2]** The garbler's rebuttal routine in the one-shot protocol

function $\Pi^{OS}_{\text{GARBLER-REBUTTAL}}$
end function

Protocol $\Pi^{OS}_{evaluator}$ **[2.3]** Evaluator's main routine

function $\Pi^{OS}_{evaluator}$
  // Receive garbled-circuits from the garbler
  Wait to receive $Comms$
  $b \xleftarrow{\$} \{0,1\}$ // select a random bit
  Send b to the garbler
  Wait to receive $\pi_{GC_b}, \pi_{Gates_{1-b}}$ from the garbler

  // Verify the correctness of circuit GC_b
  $res \leftarrow$ VerifyCircuit$(\pi_{GC_b}, b, Comms)$ // c.f. protocol VerifyCircuit [2.3]
  if res is a complaint **then** // a complaint is returned in case of an invalid verification-circuit
    Broadcast complaint res
    return $\perp$
  end if

  // Request evaluation circuit's inputs
  $m \xleftarrow{\$} \{0,1\}$ // generate the input's mask value
  $Labels_{evaluator} \leftarrow$ GetEvaluatorInputLabels$(x_{evaluator}, m, W_{evaluator}, Comms, b)$ // c.f. protocol GetEvaluatorInputLabels [2.6]
  if $Labels_{evaluator}$ is a complaint **then**
    Broadcast complaint $Labels_{evaluator}$
    return $\perp$
  end if
  $Labels_{garbler} \leftarrow$ GetGarblerInputLabels$(W_{garbler}, Comms, b)$ // c.f. protocol GetGarblerInputLabels [2.5]
  if $Labels_{garbler}$ is a complaint **then** // in case of an abort during the OT protocol
    Broadcast complaint $Labels_{garbler}$
    return $\perp$
  end if
  $I \leftarrow Labels_{garbler} \cup Labels_{evaluator}$

  // Evaluate output of the remaining garbled-circuit
  $outputLabels \leftarrow$ EvaluateGarbledCircuit$(\pi_{Gates_{1-b}}, I, 1-b, Comms)$ // c.f. protocol EvaluateGarbledCircuit [2.4]

  // Verify that the evaluation process completed successfully
  if $outputLabels$ is a complaint **then**
    Broadcast complaint $outputLabels$
    return $\perp$
  end if
  Send $outputLabels$ to the garbler
  $outputBits \leftarrow$ GetOutputBits$(outputMasks, outputLabels)$
  return $outputBits$
end function

Protocol $\Pi^{OS}_{\textbf{evaluator-rebuttal}}$ [2.4] The evaluator's rebuttal routine in the one-shot protocol

function $\Pi^{OS}_{\text{EVALUATOR-REBUTTAL}}$
end function

Protocol Π^{OS}_{arb} [2.5] The arbitration routine of the one-shot protocol

function Π^{OS}_{arb}
    Wait for a complaint $\Big(reason, (message)\Big)$. If received (**cheat-detected**, id) from $\mathcal{F}^{w}_{PVOT}$ (for any input wire w), **return** (**cheat-detected**, id).
    if received a complaint **then**
        Denote P_i the sender of the complaint
        $res \leftarrow$ HANDLEOSCOMPLAINT$(P_i, reason, message)$
        if $res \neq \perp$ **then**
            return res
        end if
    end if
end function

4.3. One-shot protocol auxiliary routines

4.3.1. Garbler subroutines

The garbler executes several subroutines during the execution of the main routine (c.f. protocol $\Pi^{OS}_{garbler}$ [2.1]): after sending the verification circuit, the garbler calls protocol SendInputLabelsDecommitments [2.2] in order to send the evaluation circuit's input labels, corresponding to the parties' inputs, to the evaluator. After receiving the evaluated labels from the evaluator, the garbler calls protocol VerifyOutputLabelsCorrectness [2.1] in order to check whether the evaluator's evaluated labels are valid labels or not. If so, the plaintext output is returned.

4.3.2. Evaluator subroutines

The evaluator executes several subroutines during the execution of the main routine (c.f. protocol $\Pi^{OS}_{evaluator}$ [2.3]): after receiving the verification circuit, the evaluator calls protocol VerifyCircuit [2.3] in order to check if the verification circuit is valid or not.

- The evaluator check if all the gates in the circuit implement a NAND gate (by observing the gates' input and output wires' labels and masks) by calling protocol OpenAllWireLabelsDecommitments [3.1]. Each specific check is done by calling protocol OpenWireLabelDecommitment [3.2]

If the verification circuit is valid, the evaluator receives the evaluation circuit and calls protocols GetGarblerInputLabels [2.5] and GetEvaluatorInputLabels [2.6] in order to receive the input labels in the evaluation circuit corresponding to the parties' inputs.

Protocol HandleOSComplaint [2.6] The arbitration routine of the one-shot protocol

function $HandleOSComplaint(id, reason, message)$
  if $reason =$ **InvalidOpening** and $id = id_{evaluator}$ **then**
    parse $message$ as a commitment c, and a decommitment d and a plain value x.
    $res \leftarrow$ HandleInvalidOpening$(index_x, (x, \pi_x), comm)$ // c.f. protocol HandleInvalidOpening [3.1]
  else if $reason =$ **InvalidVerificationGate** and $id = id_{evaluator}$ **then**
    parse $message$ as a garbled-circuit's id $id(GC)$, a gate decommitment (g, π_g), input labels' decommitments $(l_0, \pi_{l_0}), (l_1, \pi_{l_1})$, output labels' decommitments $(l_0^{out}, \pi_{l_0^{out}}), (l_1^{out}, \pi_{l_1^{out}})$, wire-masks' decommitments $(m_0, \pi_{m_0}), (m_1, \pi_{m_1}), (m_{out}, \pi_{m_{out}})$, indices b_0, b_1, a commitments' set $Comms^*$ and a decommitments' set $Decomms^*$.
    $res \leftarrow$ HandleInvalidVerificationGate$(id(GC), (g^*, \pi_{g^*}), (l_0, \pi_{l_0}), (l_1, \pi_{l_1}), (l_0^{out}, \pi_{l_0^{out}}), (l_1^{out}, \pi_{l_1^{out}}), (m^*_{w_0}, \pi_{m^*_{w_0}}), (m^*_{w_1}, \pi_{m^*_{w_1}}), (m^*_{w_{out}}, \pi_{m^*_{w_{out}}}), b_0, b_1, Comms^*)$ // c.f. protocol HandleInvalidVerificationGate [3.2]
  else if $reason =$ **InvalidEvaluationGate** and $id = id_{evaluator}$ **then**
    parse $message$ as a commitments' set $Comms$, a garbled-circuit's id $id(GC)$, a gate's decommitment (g, π_g), two wire-labels' decommitments $(l_0, \pi_{l_0}), (l_1, \pi_{l_1})$ and two indices b_0, b_1.
    $res \leftarrow$ HandleInvalidEvaluationGate$(Comms, id(GC), (g, \pi_g), (l_0, \pi_{l_0}), (l_1, \pi_{l_1}), b_0, b_1)$ // c.f. protocol HandleInvalidEvaluationGate [3.3]
  else if $reason =$ **InvalidGarblerInputLabel** and $id = id_{evaluator}$ **then**
    parse $message$ as a wire w, wire-label decommitment l_w, π_{l_w} and two commitments c_0, c_1.
    $res \leftarrow$ HandleInvalidGarblerInputLabel$(id(GC), w, (l, \pi_l), comm_0, comm_1)$ // c.f. protocol HandleInvalidGarblerInputLabel [3.4]
  else if $reason =$ **InvalidEvaluatorInputLabel** and $id = id_{evaluator}$ **then**
    parse $message$ as a wire w and a commitment c.
    $res \leftarrow$ HandleInvalidEvaluatorInputLabel$(id(GC), w, Comms)$ // c.f. protocol HandleInvalidEvaluatorInputLabel [3.5]
  else if $reason =$ **InvalidGarbledOutput** and $id = id_{garbler}$ **then**
    parse $message$ as a garbled circuit's id $id(GC)$, a wire w, a label decommitment (l, π_l) and commitments $Comms$.
    $res \leftarrow$ HandleInvalidGarbledOutput$(id(GC), w, (l, \pi_l), Comms)$ // c.f. protocol HandleInvalidGarbledOutput [3.6]
  else if $reason =$ **prove-output** and $id = id_{evaluator}$ **then**
    parse $message$ as a wire w, a label decommitment (l, π_l), a wire-mask decommitment (m, π_m) and a commitments' set $Comms$.
    $res \leftarrow$ HandleProveOutput$(id(GC), w, (l, \pi_l), (m, \pi_m), Comms)$ // c.f. protocol HandleProveOutput [3.7]
  end if
  return res
end function

Protocol VerifyOutputLabelsCorrectness [2.1] Verify that the evaluated output labels are valid output-wires' labels (subroutine of protocol $\Pi^{OS}_{garbler}$ [2.1])

function VERIFYOUTPUTLABELSCORRECTNESS($Comms, outputLabelsDecomms$)
  $Out \leftarrow 0^n$
  for output-label decommitment $(l^w, \pi_{l^w}) \in outputLabelsDecomms$ **do**
    if for all $i \in \{0,1\}$: OPENWIRELABELDECOMMITMENT$((l^w, \pi_{l^w}), Comms, 1-b, w, i) = \perp$ **then** // Check if the given decommitment doesn't any of wire w's labels commitments
      return complaint $\Big(\textbf{InvalidGarbledOutput}, (1-b, w, l^w, \pi_{l^w}, comm(l_0^w), comm(l_1^w))\Big)$
    else
      Let $i \in \{0,1\}$ s.t. OPENWIRELABELDECOMMITMENT$((l^w, \pi_{l^w}), Comms, 1-b, w, i) \neq \perp$
      $Out[w] \leftarrow i$
    end if
  end for
  return Out
end function

Protocol SendInputLabelsDecommitments [2.2] Send input-labels' decommitments to the evaluator (subroutine of protocol $\Pi^{OS}_{garbler}$ [2.1])

function SENDINPUTLABELSDECOMMITMENTS($I_{garbler}, W_{garbler}, W_{evaluator}, Comms, Labels_b, , Masks_b$
  // Send garbled-value to the evaluator
  for $w \in W_{garbler}$ **do**
    Let $v_w^{in} \in I_{garbler}$ the value to-be-evaluated in wire w
    Denote $(l_0^w, l_1^w) \in Labels_b$ the labels of wire w, and $m_w \in Masks_b$ the mask of wire w.
    $index_w \leftarrow (\textbf{wire}, b, w, v_w^{in} \oplus m_w)$
    $\pi_{l_w} \leftarrow$ OPENITEM$(l^w_{v_w^{in} \oplus m_w}, Comms, index_w)$
    Send (l_w, π_{l_w}) to the evaluator
  end for

  // Send the evaluator the required garbled-value via OT
  for $w \in W_{evaluator}$ **do**
    Denote $(l_0^w, l_1^w) \in Labels_b$ the labels of wire w, and $m_w \in Masks$ the mask of wire w.
    $index_{w,i} \leftarrow (\textbf{wire}, b, w, i)$ for $i \in \{0,1\}$
    $\pi_{l_{w,i}} \leftarrow$ OPENITEM$(l_i^w, Comms, index_{w,i})$ for $i \in \{0,1\}$
    Send $(\textbf{transfer}, (l_{w,m_w}, \pi_{l_{w,m_w}}), (l_{w,\overline{m_w}}, \pi_{l_{w,\overline{m_w}}}))$ to $\mathcal{F}^w_{PVOT}$
  end for
end function

Then, the evaluator calls protocol EvaluateGarbledCircuit [2.4] in order to evaluate the evaluation circuit, using the received input labels.

In order to provide a public proof of the evaluator's input or evaluated output, the evaluator executes protocols ProveOutput [2.1] and ProveInput [2.2] respectively.

Protocol ProveOutput [2.1] Pseudo-code for subroutine **prove-output** command – sends output-wire mask and the corresponding evaluated label

1: **function** PROVEOUTPUT(w)
2: **if** already evaluated output and received output-wires' masks **then**
3: Denote $(l_b^w, \pi_{l_b^w}), Comm(l_b^w)$, output-wire w's evaluated label decommitment and commitment (respectively)
4: Denote $(m_w, \pi_{m_w}) \in outputMasks, Comm(m_w)$, output-wire w's mask decommitment and commitment (respectively)
5: Send (**prove-output**, $w, (l_b^w, \pi_{l_b^w}), (m_w, \pi_{m_w}), Comm(l_b^w), Comm(m_w)$) to the arbitrator
6: **end if**
7: **end function**

Protocol ProveInput [2.2] Pseudo-code for subroutine **prove-input** command – sends the evaluator's input value to the arbitrator

1: **function** PROVEINPUT
2: Denote $W_{evaluator}$ the set of input-wires of the evaluator
3: **if** already received inputs from $\mathcal{F}_{PVOT}^w$, for all $w \in W_{evaluator}$ **then**
4: **for** $w \in W_{evaluator}$ **do**
5: Send **prove-output** to $\mathcal{F}_{PVOT}^w$
6: **end for**
7: **end if**
8: **end function**

4.3.3. Auxiliary garbled circuit subroutines

The following auxiliary subroutines are used by several parties.

protocol OpenAllWireLabelsDecommitments [3.1] checks if all wire labels in the verification circuit are valid, each check is done by calling protocol OpenWireLabelDecommitment [3.2].

protocol OpenAllWireMasksDecommitments [3.3] checks if all wire masks in the verification circuit are valid, each check is done by calling protocol OpenWireMaskDecommitment [3.4].

protocol OpenAllGatesDecommitments [3.5] checks if all gates in the verification circuit are valid, each check is done by calling protocol OpenGateDecommitment [3.6].

Protocol VerifyCircuit [2.3] Check the verification circuit's correctness (subroutine of protocol $\Pi^{OS}_{evaluator}$ [2.3])

1: **function** VerifyCircuit($\pi_{GC} = \pi_{Gates} \cup \pi_{Labels} \cup \pi_{Masks}, b, C$)
2: // Verify wire-labels correctness
3: $res \leftarrow$ OpenAllWireLabelsDecommitments($\pi_{Labels}, b, Comms$) // c.f. protocol OpenAllWireLabelsDecommitments [3.1]
4: **if** res is a complaint **then**
5: **return** res
6: **end if**

7: // Verify wire-masks correctness
8: $res \leftarrow$ OpenAllWireMasksDecommitments($\pi_{Masks}, b, Comms$) // c.f. protocol OpenAllWireMasksDecommitments [3.3]
9: **if** res is a complaint **then**
10: **return** res
11: **end if**

12: // Verify gates correctness
13: $res \leftarrow$ OpenAllGatesDecommitments($\pi_{Gates}, b, Comms$) // c.f. protocol OpenAllGatesDecommitments [3.5]
14: **if** res is a complaint **then**
15: **return** res
16: **end if**

17: // Verify gates' output correctness
18: $res \leftarrow$ VerifyAllGatesOutput($\pi_{GC}, b, Comms$) // c.f. protocol VerifyAllGatesOutput [2.7]
19: **if** res is a complaint **then**
20: **return** res
21: **end if**
22: **return**
23: **end function**

Protocol EvaluateGarbledCircuit [2.4] Evaluate the given garbled-circuit, using the provided input-labels (subroutine of protocol $\Pi^{OS}_{evaluator}$ [2.3])

function EvaluateGarbledCircuit($\pi_{Gates}, inputLabels, b, Comms$)
  Denote $inputLabels = \{l^w_{b_w} \mid \text{w is an input-wire}\}$
  // Iterate the gates of GC in a topological order, and evaluate their outputs
  for gate decommitment $(g, \pi_g) \in \pi_{Gates}$ **do**
    // Verify gate's consistency w.r.t. commitments
    $res \leftarrow$ OpenGateDecommitment$((g, \pi_g), Comms, b, g)$
    if res is a complaint **then**
      return res
    end if
    Denote $tuple_g = (w_0, w_1, w_{out}, tt^g)$

    // Verify output-label's correctness
    $l^{w_{out}}, \pi_{l^{w_{out}}} \leftarrow tt^g(l^{w_0}_{b_{w_0}}, l^{w_1}_{b_{w_1}})$
    if for all $i \in \{0,1\}$: OpenWireLabelDecommitment$((l^{w_{out}}, \pi_{l^{w_{out}}}), Comms, b, w_{out}, i) = \perp$ **then** // c.f. protocol OpenWireLabelDecommitment [3.2]
      return complaint $\Big(\textbf{InvalidEvaluationGate}, (1 - b, Comms, (g, \pi_g), (l^{w_0}_{b_{w_0}}, \pi_{l^{w_0}_{b_{w_0}}}), (l^{w_1}_{b_{w_1}}, \pi_{l^{w_1}_{b_{w_1}}}), b_{w_0}, b_{w_1})\Big)$
    end if
    Denote $l^{w_{out}}_i \leftarrow l^{w_{out}}$ for the index i that opens the decommitment correctly
  end for
  return $\{l^w_{b_w}, \pi_{l^w_{b_w}} \mid \text{w is an output wire of GC}\}$
end function

Protocol GetGarblerInputLabels [2.5] Receive the garbler's input labels and check their correctness (subroutine of protocol $\Pi^{OS}_{evaluator}$ [2.3])

function GetGarblerInputLabels($W_{garbler}, Comms, b$)
  $\pi^* \leftarrow \{\}$
  for $w \in W_{garbler}$ **do**
    Wait to receive (l^w, π_{l^w}) from the evaluator
    if $\forall i \in \{0,1\}$: OpenWireLabelDecommitment$((l^w, \pi_{l^w}), Comms, 1 - b, w, i) = \perp$ **then**
      return complaint $\Big(\textbf{InvalidGarblerInputLabel}, ((l^w, \pi_{l^w}), Comms, 1 - b, w)\Big)$
    end if
  end for
  return $\{l^w \mid w \in W_{garbler}\}$
end function

Protocol GetEvaluatorInputLabels [2.6] Receive the evaluator's input labels and check their correctness (subroutine of protocol $\Pi^{OS}_{evaluator}$ [2.3])

function GetEvaluatorInputLabels($x_{evaluator}, m, W_{evaluator}, Comms, b$)
  for $w \in W_{evaluator}$ **do**
    if $w = w_m$ **then**
      $v_w \leftarrow m$
    else
      $v_w \leftarrow x_{evaluator}[w] \oplus m$
    end if
    Send (**choose**, v_w) to $\mathcal{F}^w_{PVOT}$
    Wait to receive $\big($**message**$, l^w, \pi_{l^w}\big)$ from $\mathcal{F}^w_{PVOT}$
    if received **cheat-detected** from $\mathcal{F}^w_{PVOT}$ **then**
      return $\perp$
    end if
    if OpenWireLabelDecommitment$((l^w, \pi_{l^w}), Comms, 1 - b, w, v_w) = \perp$ **then**
      return complaint $\Big($**InvalidEvaluatorInputLabel**$, (Comms, 1 - b, w)\Big)$
    end if
  end for
  return $\{l^w \mid w \in W_{evaluator}\}$
end function

Protocol VerifyAllGatesOutput [2.7] Verify that each gate in the circuit implements a valid NAND gate

function VerifyAllGatesOutput($\pi_{GC} = \pi_{Gates} \cup \pi_{Labels} \cup \pi_{Masks}, b, Comms$)
  for $tuple_g = (w_0, w_1, w_{out}, tt_g)$ s.t. $g \in \pi_{Gates}$ **do**
    Denote $m_{w_0}, m_{w_1}, m_{out} \in \pi_{Masks}$, the masks corresponding to the input and output wires
    Let $L_w \subset \pi_{Labels}$ denote the set of labels of wire w, for $w \in \{w_0, w_1, w_{out}\}$

    // Verify that g yields valid output for any given valid inputs
    for $(l^{w_0}_{b_0}, l^{w_1}_{b_1}) \in L_{w_0} \times L_{w_1}$ **do**
      if VerifyGateOutput$(g, l^{w_0}_{b_0}, l^{w_1}_{b_1}, m_{w_0}, m_{w_1}, m_{out}, L_{w_{out}}) =$ **false then** // c.f. protocol VerifyGateOutput [2.8]
        return complaint $\Big($**InvalidVerificationGate**$, ((g, \pi_g), (l^{w_0}_{b_1}, \pi_{l^{w_0}_{b_1}}), (l^{w_0}_{b_1}, \pi_{l^{w_0}_{b_1}}), (m_{w_0}, \pi_{m_{w_0}}$

      end if
    end for
  end for
end function

Protocol VerifyGateOutput [2.8] Check if the given gate evaluates a valid output (and a corresponding NAND value), given the provided input labels

function VerifyGateOutput($tuple_g = (w_0, w_1, w_{out}, tt_g), l_{b_0}^{w_0}, l_{b_1}^{w_1}, m_{w_0}, m_{w_1}, m_{w_{out}}, L_{w_{out}}$)
  $l_{b_{w_{out}}}^{w_{out}} \leftarrow tt_g(l_{b_0}^{w_0}, l_{b_1}^{w_1})$
  if $l_{b_{w_{out}}}^{w_{out}} \notin L_{w_{out}}$ **then**
    return false
  end if
  $value_{w_i} \leftarrow b_i \oplus m_{w_i}$, for $i \in \{0, 1\}$
  $value_{w_{out}} \leftarrow b_{w_{out}} \oplus m_{w_{out}}$
  if $value_{w_{out}} \neq \overline{value_{w_0} \wedge value_{w_1}}$ **then**
    return false
  end if
  return true
end function

Protocol OpenAllWireLabelsDecommitments [3.1] Extract all wire-labels from the given decommitments

procedure OpenAllWireLabelsDecommitments($\pi_{Labels} = \{(l_w^*, \pi_{l_w^*}) \mid w$ is a wire$\}, b, Comms$)
  for wire w in the circuit **do**
    for $i \in \{0, 1\}$ **do**
      $res \leftarrow$ OpenWireLabelDecommitment$((l_w^*, \pi_{l_w^*}), Comms, b, w, i)$ // c.f. protocol OpenWireLabelDecommitment [3.2]
      if res is a complaint **then**
        return res
      end if
    end for
  end for
  return
end procedure

Protocol OpenWireLabelDecommitment [3.2] Open decommitment corresponding to a wire-label

function OpenWireLabelDecommitment$((l, \pi_l), Comms, b, w, i)$
  $index_{w,i} \leftarrow (\textbf{wire}, b, w, i)$
  Denote $comm(l_w^i) \in Comms$ the commitment corresponding to the label at index i of wire w
  if VerifyDecommitment$(index_{w,i}, (l, \pi_l), comm(l_i^w)) = \perp$ **then**
    return complaint $\Big(\textbf{InvalidOpening}, (index_{w,i}, (l, \pi_l), comm(l_i^w))\Big)$
  end if
  return
end function

Protocol OpenAllWireMasksDecommitments [3.3] Open all decommitments corresponding to the wire-masks

1: **function** OpenAllWireMasksDecommitments($\pi_{Masks} = \{(m^*_w, \pi_{m^*_w}) \mid w \text{ is a wire}\}, b, Comms$)
2: **for** each wire w in the circuit **do**
3: $res \leftarrow$ OpenWireMaskDecommitment($(m^*_w, \pi_{m^*_w}), Comms, b, w$) // c.f. protocol OpenWireMaskDecommitment [3.4]
4: **if** res is a complaint **then**
5: **return** res
6: **end if**
7: **end for**
8: **return**
9: **end function**

Protocol OpenWireMaskDecommitment [3.4] Open decommitment corresponding to a wire-mask

1: **function** OpenWireMaskDecommitment($(m, \pi_m), Comms, b, w$)
2: $index_w \leftarrow (\textbf{wireValue}, b, w)$
3: Denote $comm(m_w) \in Comms$ the commitment corresponding to the mask of wire w
4: **if** VerifyDecommitment($index_w, (m, \pi_m), comm(m_w)$) $= \perp$ **then**
5: **return** complaint $\Big(\textbf{InvalidOpening}, (index_w, (m, \pi_m), comm(m_w))\Big)$
6: **end if**
7: **return**
8: **end function**

Protocol OpenAllGatesDecommitments [3.5] Open all gates' decommitments

1: **function** OpenAllGatesDecommitments($\pi_{Gates} = \{(g^*, \pi_{g^*}) \mid g \text{ is a gate}\}, b, Comms$)
2: **for** each gate g in circuit GC **do**
3: $res \leftarrow$ OpenGateDecommitment($(g^*, \pi_{g^*}), Comms, b, g$) // c.f. protocol OpenGateDecommitment [3.6]
4: **if** res is a complaint **then**
5: **return** res
6: **end if**
7: **end for**
8: **return**
9: **end function**

Protocol OpenGateDecommitment [3.6] Open decommitment corresponding to a gate

function OPENGATEDECOMMITMENT($(g^*, \pi_{g^*}), Comms, b, g$)
  $index_g \leftarrow (\textbf{gate}, b, g)$
  Denote $comm(g) \in Comms$ the commitment corresponding to gate g
  if VERIFYDECOMMITMENT($index_g, (g^*, \pi_{g^*}), comm(g)$) $= \perp$ **then**
    return complaint $\Big(\textbf{InvalidOpening}, (index_g, (g^*, \pi_{g^*}), comm(g))\Big)$
  end if
  return
end function

4.3.4. Arbitration subroutines

The following subroutines are called by the arbitrator while executing its main routine (c.f. protocol Π^{OS}_{arb} [2.5]).

protocol HandleInvalidOpening [3.1] handles an evaluator's complaint where a decommitment of a given value, sent by the garbler, doesn't match the commitment of the corresponding value. The garbler is accused iff the given decommitment and commitment were sent by the garbler, and correspond to the same value, but they don't match.

Protocol HandleInvalidOpening [3.1] Handling the case where an invalid decommitment of component x was received

// This complaint can be sent only by the evaluator!
function HANDLEINVALIDOPENING($index_x, (x, \pi_x), comm$)
  if x, π_x or $comm$ aren't signed by the garbler **then**
    return $\Big(\textbf{cheat-detected}, id_{evaluator}\Big)$
  end if
  if VERIFYDECOMMITMENT($index_x, (x, \pi_x), comm$) $= \perp$ **then**
    return $\Big(\textbf{cheat-detected}, id_{garbler}\Big)$
  else
    return $\Big(\textbf{cheat-detected}, id_{evaluator}\Big)$
  end if
end function

protocol HandleInvalidVerificationGate [3.2] handles an evaluator's complaint where the verification circuit contains an invalid gate — the arbitrator accuse the garbler iff the gate (and its corresponding labels and masks) were sent by the garbler, and the given gate doesn't implement a NAND gate for the given input labels and corresponding wires' masks.

protocol HandleInvalidEvaluationGate [3.3] handles an evaluator's complaint where the evaluation circuit contains an invalid gate — the arbitrator accuse the garbler iff the given input labels are *valid* labels of the corresponding gate's input wires, and the gate

Protocol HandleInvalidVerificationGate [3.2] Handling the case where the garbler sent an invalid verification circuit

// This complaint can be sent only by the evaluator!

function HandleInvalidVerificationGate($id(GC), (g^*, \pi_{g^*}), (l_0, \pi_{l_0}), (l_1, \pi_{l_1}), (l_0^{out}, \pi_{l_0^{out}}), (l_1^{out}, \pi_{l_1^{out}}), (m^*_{w_0}, \pi_{m^*_{w_0}}), (m^*_{w_1}, \pi_{m^*_{w_1}}), (m^*_{w_{out}}, \pi_{m^*_{w_{out}}}), b_0, b_1, Comms^*$)
  if any of the given parameters – except b_0 or b_1 – aren't signed by the garbler **then**
    return $\left(\textbf{cheat-detected}, id_{evaluator}\right)$
  end if
  if OpenGateDecommitment$((g^*, \pi_{g^*}), id(GC), id(g^*)) = \perp$ **then**
    return $\left(\textbf{cheat-detected}, id_{evaluator}\right)$
  end if
  Denote $g = g^*$, and $tuple_g = (w_0, w_1, w_{out}, tt_g)$
  // Check if provided wire-masks' decommitments are valid
  for $w \in \{w_0, w_1, w_{out}\}$ **do**
    if OpenWireMaskDecommitment$((m^*_w, \pi_{m^*_w}), Comms^*, id(GC), w) = \perp$ **then** // c.f. protocol OpenWireMaskDecommitment [3.4]
      return $\left(\textbf{cheat-detected}, id_{evaluator}\right)$
    end if
    Denote $m_w = m^*_w$
  end for
  // Check if provided wire-labels' decommitments are valid
  for $i \in \{0, 1\}$ **do**
    $res_i^{out} \leftarrow$ OpenWireLabelDecommitment$((l_i^{out}, \pi_{l_i^{out}}), Comms^*, id(GC), w_{out}, i)$ // c.f. protocol OpenWireLabelDecommitment [3.2]
    $res_i \leftarrow$ OpenWireLabelDecommitment$((l_i, \pi_{l_i}), Comms^*, id(GC), w_i, b_i)$ // c.f. protocol OpenWireLabelDecommitment [3.2]
    if $res_i = \perp$ or $res_i^{out} = \perp$ **then**
      return $\left(\textbf{cheat-detected}, id_{evaluator}\right)$
    end if
    Denote $l_{b_i}^{w_i} = l_i$
    Denote $l_i^{w_{out}} = l_i^{out}$
  end for
  // Check if the provided gate is valid (represents a garbled NAND gate and evaluates a valid label)
  Denote $L_{w_{out}} \subset \pi_{GC}$ the set of label-decommitments of wire w_{out}, as provided in π_{GC}
  if not VerifyGateOutput$(g, l_{b_0}^{w_0}, l_{b_1}^{w_1}, m_{w_0}, m_{w_1}, m_{w_{out}}, \{l_0^{w_{out}}, l_1^{w_{out}}\})$ **then** // c.f. protocol VerifyGateOutput [2.8]
    return $\left(\textbf{cheat-detected}, id_{garbler}\right)$
  else
    return $\left(\textbf{cheat-detected}, id_{evaluator}\right)$
  end if
end function

doesn't output a valid label while using these inputs.

Protocol HandleInvalidEvaluationGate [3.3] Handling the case where the garbler sent an invalid evaluation circuit

// This complaint can be sent only by the evaluator!
function HandleInvalidEvaluationGate($Comms, id(GC), (g, \pi_g), (l_0, \pi_{l_0}), (l_1, \pi_{l_1}), b_0, b_1$)
  if $(g, \pi_g), (l_0, \pi_{l_0}), (l_1, \pi_{l_1})$ or $Comms$ aren't signed by the garbler **then**
    return $\left(\textbf{cheat-detected}, id_{evaluator}\right)$
  end if
  $tuple_g = (w_0, w_1, w_{out}, tt_g) \leftarrow$ OpenGateDecommitment$((g, \pi_g), Comms, id(GC), g)$
  if $tuple_g = \perp$ **then**
    return $\left(\textbf{cheat-detected}, id_{evaluator}\right)$
  end if
  // Check if provided wire-labels' decommitments are valid
  For all $i \in \{0,1\}$, $res_i \leftarrow$ OpenWireLabelDecommitment$((l_i, \pi_{l_i}), Comms, id(GC), w_i, b_i)$ // c.f. protocol OpenWireLabelDecommitment [3.2]
  if $res_i = \perp$ for any $i \in \{0,1\}$ **then**
    return $\left(\textbf{cheat-detected}, id_{evaluator}\right)$
  end if
  // Check if the provided gate is valid (evaluates a valid label)
  $(l_{out}, \pi_{l_{out}}) \leftarrow tt_g(l_0, l_1)$
  $res_i \leftarrow$ OpenLabelDecommitment$(\pi_{l_{out}}, Comms, id(GC), w_{out}, i)$ // c.f. protocol OpenWireLabelDecommitment [3.2]
  if $res_0 = \perp$ and $res_1 = \perp$ **then**
    return $\left(\textbf{cheat-detected}, id_{garbler}\right)$
  else
    return $\left(\textbf{cheat-detected}, id_{evaluator}\right)$
  end if
end function

protocol HandleInvalidGarblerInputLabel [3.4] handles an evaluator's complaint where one decommitment corresponding to the garbler's input is invalid — the arbitrator accuse the garbler iff the given decommitment was sent from the garbler and doesn't match to any of the corresponding wire's labels commitments.

protocol HandleInvalidEvaluatorInputLabel [3.5] handles an evaluator's complaint where one decommitment corresponding to the evaluator's input is invalid — the arbitrator accuse the garbler iff the given decommitment was the value received by the evaluator in OT, and it doesn't match to any of the corresponding wire's labels commitments.

protocol HandleInvalidGarbledOutput [3.6] handles a garbler's complaint where he

Protocol HandleInvalidGarblerInputLabel [3.4] Handling the case where the garbler sent invalid input-labels to-be-used in the evaluation process

// This complaint can be sent only by the evaluator!
function HandleInvalidGarblerInputLabel($id(GC), w, (l, \pi_l), comm_0, comm_1$)
  if $comm_0, comm_1$ or (l, π_l) aren't signed by the garbler **then**
    return $\left(\textbf{cheat-detected}, id_{evaluator}\right)$
  end if
  For all $i \in \{0,1\}$, denote $index_{w,i} = (\textbf{wire}, id(GC), w, i)$
  if $\forall i \in \{0,1\}$: VerifyDecommitment$(index_{w,i}, l, \pi_l, comm_i) = \perp$ **then**
    return $\left(\textbf{cheat-detected}, id_{garbler}\right)$
  else
    return $\left(\textbf{cheat-detected}, id_{evaluator}\right)$
  end if
end function

Protocol HandleInvalidEvaluatorInputLabel [3.5] Handling the case where the garbler sent invalid input-labels to-be-used in the evaluation process

// This complaint can be sent only by the evaluator!
function HandleInvalidEvaluatorInputLabel$(id(GC), w, Comms)$
  if *comm* isn't signed by the garbler or w isn't an evaluator input wire **then**
    return $\left(\textbf{cheat-detected}, id_{evaluator}\right)$
  end if
  if no message received from $\mathcal{F}^w_{PVOT}$ **then**
    return $\left(\textbf{cheat-detected}, id_{evaluator}\right)$
  end if
  Let $\left(\textbf{prove-output}, b_w, \alpha_{b_w}\right)$ be the message received from $\mathcal{F}^w_{PVOT}$, denote $\alpha_{b_w} = (l^w_{b_w}, \pi_{l^w_{b_w}})$
  if OpenWireLabelDecommitment$((l^w_{b_w}, \pi_{l^w_{b_w}}), Comms, id(GC), w, b_w) = \perp$ **then** // c.f. protocol OpenWireLabelDecommitment [3.2]
    return $\left(\textbf{cheat-detected}, id_{garbler}\right)$
  else
    return $\left(\textbf{cheat-detected}, id_{evaluator}\right)$
  end if
end function

received from the evaluator invalid output wires' labels — the arbitrator accuse the evaluator iff the given decommitment is signed by the evaluator, and it doesn't match to any of the corresponding wire's labels commitments.

Protocol HandleInvalidGarbledOutput [3.6] Handling the case where the evaluator sent invalid output-labels

```
    // This complaint can be sent only by the garbler!
 1: function HandleInvalidGarbledOutput(id(GC), w, (l, π_l), Comms)
 2:     if (π_l, l), comm_0 or comm_1 aren't signed by the garbler then
 3:         return (cheat-detected, id_evaluator)
 4:     else if w doesn't correspond to an output-wire then
 5:         return (cheat-detected, id_evaluator)
 6:     end if
 7:     if for all i ∈ {0,1}: OpenWireLabelDecommitment((l, π_l), Comms, id(GC), w, i) = ⊥ then
 8:         return (cheat-detected, id_evaluator)
 9:     else
10:         return (cheat-detected, id_garbler)
11:     end if
12: end function
```

protocol HandleProveOutput [3.7] provides a public proof of the value evaluated in a given output wire.

Protocol HandleProveOutput [3.7] Handling the case where the evaluator sends proof of the evaluated output

// This routine can only be used by the evaluator!
function HANDLEPROVEOUTPUT($id(GC), w, (l, \pi_l), (m, \pi_m), Comms$)
  if $(l, \pi_l), (m, \pi_m)$ or $Comms$ aren't signed by the garbler **then**
    return $\left(\textbf{cheat-detected}, id_{evaluator}\right)$
  end if
  if w isn't an output wire **then**
    return $\left(\textbf{cheat-detected}, id_{evaluator}\right)$
  end if
  if for all $i \in \{0,1\}$: OPENWIRELABELDECOMMITMENT$((l, \pi_l), Comms, id(GC), w, i) = \perp$ **then** // c.f. protocol OpenWireLabelDecommitment [3.2]
    return $\left(\textbf{cheat-detected}, id_{evaluator}\right)$
  else if OPENWIREMASKDECOMMITMENT$((m, \pi_m), Comms, id(GC), w) = \perp$ **then** // c.f. protocol OpenWireMaskDecommitment [3.4]
    return $\left(\textbf{cheat-detected}, id_{evaluator}\right)$
  end if
  Denote b the index s.t. $res_b \neq \perp$
  Let $v_w^i = m \oplus b$ (s.t. P_i is the complaining party)
  return message $(\textbf{prove-output}, i, w, v_w^i)$
end function

5. One-Shot Protocol Security Analysis

5.1. Simulatability of Corrupt Garbler

Note: In the simulator's description, there are cases where the simulator holds enough information which makes him able to perfectly-simulate the rest of the protocol's execution without additional information. This includes the cases where an abort should be sent by the honest party.

The following describes a simulator for simulating a corrupt-garbler's view, assuming the ability to access ideal functionalities $\mathcal{F}_{PVOT}$ and $\mathcal{F}_{comm}$:

1. $\mathcal{S}$ waits for the garbler to commit (i.e. send to $\mathcal{F}_{comm}$) his inputs: garbled circuits (denoted GC_0, GC_1) and input value (denoted x).
2. $\mathcal{S}$ extracts the garbled circuits from $\mathcal{F}_{comm}$
3. $\mathcal{S}$ observes the sent garbled-circuits:
   a) If exists $i \in \{0,1\}$ s.t. GC_i is invalid (if both are invalid, set $i \xleftarrow{\$} \{0,1\}$), $\mathcal{S}$ sends **cheat** to $\mathcal{F}_{\varepsilon-\text{one-shot}}$, and receives a response r and the evaluator's input $x_{evaluator}$:
      i. If $r =$ **cheat-detected**, $\mathcal{S}$ sends $b = i$ to the garbler.
      ii. Else, $r =$ **cheat-undetected**, $\mathcal{S}$ sends $b = 1 - i$ to the garbler.

      Execute an honest simulation of the evaluator using $x_{evaluator}$, the randomness chosen above for b, and $m \xleftarrow{\$} \{0,1\}$.
   b) Else, (i.e. both garbled-circuits are valid), $\mathcal{S}$ sends a random $b \in \{0,1\}$ and continues as follows.
4. Execute lines 7 to 12 in protocol $\Pi^{OS}_{evaluator}$ [2.3] (i.e. check correctness of verification circuit GC_b)
   a) If a cheating attempt was detected, send **blatant-cheat** to $\mathcal{F}_{\varepsilon-\text{one-shot}}$ and receive the honest evaluator's input $x_{evaluator}$. Then, execute an honest simulation of the evaluator using $x_{evaluator}$, the randomness chosen above for b, and $m \xleftarrow{\$} \{0,1\}$.
5. Denote $W^{input}_{evaluator}$ the evaluator's input wires
6. For all wires $w \in W^{input}_{evaluator}$:
   a) If the garbler sent $(guess, c)$ to $\mathcal{F}^{w}_{PVOT}$ (for an input wire w):

i. $\mathcal{S}$ sends **cheat** to $\mathcal{F}_{\varepsilon-\text{one-shot}}$, and receives a response r and the evaluator's input $x_{evaluator}$:

A. If $r =$ **cheat-undetected**, simulate a message **cheat-undetected** sent from $\mathcal{F}^{w}_{PVOT}$ to the garbler, choose the mask $m = 1 - y_i \oplus c$

B. If $r =$ **cheat-detected**, simulate a message **cheat-detected** sent from $\mathcal{F}^{w}_{PVOT}$ to the garbler, choose the mask $m = y_i \oplus c$

ii. Execute an honest simulation of the evaluator using $x_{evaluator}$, the randomness chosen above for b and m.

b) Else (i.e. the sender sent $(\textbf{transfer}, l^0_w, l^1_w)$ to $\mathcal{F}^{w}_{PVOT}$):

i. If at least one l^c_w $(c \in \{0,1\})$ is an invalid wire label (w.r.t. its corresponding commitment): $\mathcal{S}$ sends **cheat** to $\mathcal{F}_{\varepsilon-\text{one-shot}}$, and receives a response r and the evaluator's input $x_{evaluator}$:

A. If $r =$ **cheat-undetected**, simulate a message $(cheat - undetected)$ sent from $\mathcal{F}_{\varepsilon-\text{one-shot}}$ to the garbler, choose the mask $m = 1 - y_i \oplus c$

B. If $r =$ **cheat-detected**, simulate a message $(cheat - detected)$ sent from $\mathcal{F}_{\varepsilon-\text{one-shot}}$ to the garbler, choose the mask $m = y_i \oplus c$

Execute an honest simulation of the evaluator using $x_{evaluator}$, the randomness chosen above for b and m.

c) $m \xleftarrow{\$} \{0,1\}$

7. For each $w \in W^{input}_{garbler}$:

a) Denote (l_w, π_{l_w}) the label sent by the garbler, c_0, c_1 the commitments corresponding to both labels of wire w (received in item 1), and m_w the mask corresponding to input wire w, as extracted in item 2.

b) If not exists $i \in \{0,1\}$ s.t. $\text{OpenWireLabelDecommitment}((l_w, \pi_{l_w}), c_i, b, w, i) \neq \perp$:

- Send **blatant-cheat** to $\mathcal{F}_{\varepsilon-\text{one-shot}}$ and receives the honest evaluator's input $x_{evaluator}$. Then, execute an honest simulation of the evaluator using $x_{evaluator}$, the randomness chosen above for b and m.

c) Otherwise, set $x_{garbler}[w] \leftarrow i \oplus m_w$

8. $\mathcal{S}$ waits for the garbler to send garbled-circuit GC_{1-b} to the evaluator:

a) If the given values aren't valid decommitments of GC_{1-b}, $\mathcal{S}$ sends **blatant-cheat** to $\mathcal{F}_{\varepsilon-\text{one-shot}}$ and receives the honest evaluator's input $x_{evaluator}$. Then, execute an honest simulation of the evaluator using $x_{evaluator}$, the randomness chosen above for b and m.

9. $\mathcal{S}$ queries $\mathcal{F}_{\varepsilon-\text{one-shot}}$ using $x_{garbler}$, and gets $y = f(x_{garbler}, x_{evaluator})$ as a response

10. Denote W_{output} the output wires of the circuit, and for each $w \in W_{output}$ their corresponding labels (l_w^0, l_w^1) that were extracted in item 2. Send $\{l_w^{y[w]} \mid w \in W_{output}\}$ to the sender.

Claim 5.1.1. *The corrupt-garbler's view in the real-world model and the ideal-world model are indistinguishable.*

Proof. The environment's view consists of a tuple of messages, denoted by the following RVs:

$$B, M, \{OUT^{OT,w}\}_{w \in W_{evaluator}},\ LABELS,\ OUT, VIEW^{arb}$$

where B, M correspond to the randomness for choosing b, m respectively, $OUT^{OT,w}$ corresponds to the output of $\mathcal{F}_{PVOT}^{w}$ (for each $w \in W_{evaluator}$), $LABELS$ corresponds to the output labels sent to the garbler, OUT corresponds to the honest party's output and $VIEW_{arb}$ corresponds to the messages sent during arbitration.

We'll use a hybrid argument in order to prove this claim. The different hybrids are defined as follows:

1. **Hybrid #1 (i.e real world)**

2. **Hybrid #2** — same as hybrid #1, except that the evaluator receives the output from the ideal functionality $\mathcal{F}_{\varepsilon-\text{one-shot}}$.
   The evaluator's inputs $x_{evaluator}, b$ and m are still given to the simulator, as in previous hybrid. In addition, we are able to configure the output sent to the honest evaluator even if a cheating attempt didn't occurred.

   The simulator for this world is described as follows:

   a) Simulate a real-world execution of an evaluator using $x_{evaluator}, b$ and m. Denote the execution's output as *out*.

   b) Configure $\mathcal{F}_{\varepsilon-\text{one-shot}}$ to send the output *out* to the honest evaluator.

3. **Hybrid #3** — same as hybrid #2, except that we don't use the evaluator's randomness b. However, the evaluator's inputs $m, x_{evaluator}$ are still given to the simulator, as in previous hybrid.

   During the cut-and-choose phase, the evaluator chooses a bit b and sends it to the garbler: in hybrid #2, b is picked from a uniformly-distributed random-variable, while in Hybrid #3 it depends on the correctness of the sent garbled-circuits: if both are valid/invalid b is picked using the same randomness as in the real-world. If one of them is invalid, the circuit to-be-verified will be chosen according to $\mathcal{F}_{\varepsilon-\text{one-shot}}$'s decision to detect the cheating attempt or not.

   The simulator for this world is described as follows:

   a) Execute the simulator presented in previous hybrid

      - Instead of executing line 4 in the real-world protocol (i.e. choosing the value of b), execute item 3 in the ideal-world simulator.

4. **Hybrid #4** — same as hybrid #3, except that we don't use the evaluator's randomness m. However, the evaluator's input $x_{evaluator}$ is still given to the simulator, as in previous hybrid.

   During the OT phase (input-labels transmission), the evaluator chooses an input-mask bit m and requests its label from the garbler: in hybrid #3 m is picked from a uniformly-distributed random-variable, while in Hybrid #4 it depends on the wire-labels' correctness that are sent in OT: if all wire-labels are valid m is picked using the same randomness as in hybrid #1, otherwise we choose the value of m in such way that at least one invalid label will be sent to the evaluator (according to $\mathcal{F}_{\varepsilon-\text{one-shot}}$'s decision to detect the cheating attempt or not).

   The simulator for this world is described as follows:

   a) Execute the simulator presented in previous hybrid

      - Instead of executing line 14 in the real-world protocol (i.e. choosing the value of m), choose m as in item 6 in the ideal-world simulator.

5. **Hybrid #5 (i.e. ideal-world)** — same as hybrid #4, except that we change the honest evaluator's output, and the output labels sent to the corrupt garbler.

   The simulator for this world is described as follows:

   a) Execute the simulator presented in previous hybrid

      - Instead of executing line 20 in the real-world protocol, execute items 7 to 9 in the ideal-world simulator (receive the garbler's labels and extract his input $x_{garbler}$).
      - Instead of executing line 27 in the real-world protocol (i.e. evaluating the evaluation circuit) in order to obtain the output, use the output obtained from $\mathcal{F}_{\varepsilon-\text{one-shot}}$ at item 9 in the ideal-world simulator (receive output from $\mathcal{F}_{\varepsilon-\text{one-shot}}$).

For hybrid $\#i$, denote the following RVs:

$$B_i, M_i, \{OUT_i^{OT,w}\}_{w \in W_{evaluator}},\ LABELS_i,\ OUT_i, VIEW_i^{arb}$$

We'll prove indistinguishability between adjacent hybrid worlds:

1. **Hybrid #1 - Hybrid #2** — the simulator perfectly simulates an execution of an honest evaluator, therefore the output sent from the simulator equals to the output obtained from a real execution.

2. **Hybrid #2 - Hybrid #3** — Denote $VIEW$ the set of possible views to be generated in hybrid #2, and for all $i \in \{2, 3\}$ denote $VIEW_i$ a RV of the view generated in hybrid #i. We'll show that for any $view \in VIEW$:

$$\Pr[VIEW_2 = view] = \Pr[VIEW_3 = view]$$

For $i \in \{2,3\}$, denote $VIEW_i^{suff} = (\{OUT_i^{OT,w}\}_{w \in W_{evaluator}}, LABELS_i, OUT_i, VIEW_i^{arbitration})$. Hence, for $i \in \{2,3\}$:

$$\begin{aligned}&\Pr[VIEW_i = view] = \Pr[(B_i, VIEW_i^{suff}) = (b, view^{suff})] \\ &= \Pr[B_i = b] \cdot \Pr[VIEW_i^{suff} = view^{suff} \mid B_i = b]\end{aligned}$$

Assuming that the value of b chosen in both hybrids is the same, the rest of the simulation is the same. Hence,

$$\Pr[VIEW_2^{suff} = view^{suff} \mid B_2 = b] = \Pr[VIEW_3^{suff} = view^{suff} \mid B_3 = b]$$

Therefore,

$$\Pr[VIEW_2 = view] = \Pr[VIEW_3 = view] \iff \Pr[B_2 = b] = \Pr[B_3 = b]$$

We'll prove that $\Pr[B_2 = b] = \Pr[B_3 = b]$, then it can be deduced that both views are indistinguishable.

In hybrid #2 the evaluator selects the value of b using an independent uniformly-distributed random-variable, therefore:

$$\Pr(B_2 = b) = 0.5$$

In hybrid #3, the value of b depends on the correctness of the garbled-circuits that were sent by the garbler. Denote X_{diff} the event that exactly one of GC_0, GC_1 is invalid. Let $p_3 = \Pr(B_3 = b)$, hence:

$$\begin{aligned}p_3 &= \Pr(B_3 = b \wedge \overline{X_{diff}}) + \Pr(B_3 = b \wedge X_{diff}) \\ &= \Pr(\overline{X_{diff}}) \cdot \underbrace{\Pr(B_3 = b \mid \overline{X_{diff}})}_{0.5\ (*)} + \Pr(X_{diff}) \cdot \underbrace{\Pr(B_3 = b \mid X_{diff})}_{0.5\ (**)} \\ &= 0.5 \cdot \Pr(\overline{X_{diff}}) + \Pr(X_{diff}) \cdot \underbrace{\Pr(B_3 = b \mid X_{diff})}_{p^{\text{not-same}}}\end{aligned}$$

We will show that $p^{\text{not-same}} = 0.5$ — assuming X_{diff}, i.e. only one garbled-circuit is valid, $B_3 = b$ iff GC_b is invalid and the cheating attempt is detected, or GC_{1-b} is invalid and the cheating attempt is undetected. Let $p_3^{\text{not same}} = \Pr(B_3 = b \mid X_{diff})$,

and $X_{detected}$ an event indicating that a cheating attempt was detected. hence:

$$\begin{aligned}
p_3^{\text{not same}} &= \Pr(X_{detected} \wedge GC_b \text{ is invalid} \mid X_{diff}) \\
&\quad + \Pr(\overline{X_{detected}} \wedge GC_{1-b} \text{ is invalid} \mid X_{diff}) \\
&= \underbrace{\Pr(X_{detected} \mid X_{diff})}_{0.5} \cdot \Pr(GC_b \text{ is invalid} \mid X_{diff}) \\
&\quad + \underbrace{\Pr(\overline{X_{detected}} \mid X_{diff})}_{0.5} \cdot \Pr(GC_{1-b} \text{ is invalid} \mid X_{diff}) \\
&= 0.5 \cdot \underbrace{\Big(\Pr(GC_b \text{ is invalid} \mid X_{diff}) + \Pr(GC_{1-b} \text{ is invalid} \mid X_{diff}) \Big)}_{1 \text{ (complementary events)}} \\
&= 0.5
\end{aligned}$$

Revisiting p_3, we get that:

$$\begin{aligned}
p_3 &= 0.5 \cdot \Pr(\overline{X_{diff}}) + \Pr(X_{diff}) \cdot \underbrace{\Pr(B_3 = b \mid X_{diff})}_{p_3^{not-same}} \\
&= 0.5 \cdot \Big(\underbrace{\Pr(\overline{X_{diff}}) + \Pr(X_{diff})}_{1 \text{ (complementary events)}} \Big) = 0.5
\end{aligned}$$

Hence, $\Pr(B_2 = b) = \Pr(B_3 = b)$. Therefore, both views are indistinguishable.

(*) If both are valid/invalid (i.e. $\overline{X_{diff}}$) — the simulator sends a random $b \xleftarrow{\$} \{0,1\}$.

3. **Hybrid #3 - Hybrid #4** — Denote $VIEW$ the set of possible views to be generated in hybrid #3. We'll show that for any $view \in VIEW$:

$$\Pr[VIEW_3 = view] = \Pr[VIEW_4 = view]$$

For $i \in \{3,4\}$, denote $VIEW_i^{pref} = (B_i), VIEW_i^{suff} = (LABELS_i, OUT_i, VIEW_i^{arbitration})$. Hence, for $i \in \{3,4\}$:

$$\begin{aligned}
&\Pr[VIEW_i = view] = \Pr[(VIEW_i^{pref}, M_i, OUT_i^{OT}, VIEW_i^{suff}) = (view^{pref}, m, \{out_w^{OT}\}_{w \in W_{evaluator}}, \\
&= \Pr[VIEW_i^{pref} = view^{pref}] \cdot \Pr[M_i = m] \cdot \Pr[OUT^{OT} = \{out_w^{OT}\}_{w \in W_{evaluator}} \mid M_i = m, VIEW_i^{pref} \\
&\quad \cdot \Pr[VIEW_i^{suff} = view^{suff} \mid VIEW_i^{pref} = view^{pref}, OUT^{OT} = \{out_w^{OT}\}_{w \in W_{evaluator}}]
\end{aligned}$$

Both hybrids behave the same way before choosing $\{out_w^{OT}\}_{w \in W_{evaluator}}$. Hence,

$$\Pr[VIEW_3^{pref} = view^{pref}] = \Pr[VIEW_4^{pref} = view^{pref}]$$

Moreover, assuming that the value of $\{out_w^{OT}\}_{w \in W_{evaluator}}$ chosen in both hybrids is the same, the rest of the simulation is the same. Hence,

$$\begin{aligned}
&\Pr[VIEW_3^{suff} = view^{suff} \mid (VIEW_3^{pref}, OUT_3^{OT}) = view^{pref}, \{out_w^{OT}\}_{w \in W_{evaluator}}] \\
&= \Pr[VIEW_4^{suff} = view^{suff} \mid (VIEW_4^{pref}, OUT_4^{OT}) = view^{pref}, \{out_w^{OT}\}_{w \in W_{evaluator}}]
\end{aligned}$$

Therefore, $\Pr[VIEW_3 = view] = \Pr[VIEW_4 = view]$ iff

$$\Pr[OUT_3^{OT} = \{out_w^{OT}\}_{w \in W_{evaluator}}, M_3 = m \mid VIEW_3^{pref} = view^{pref}] = \Pr[OUT_4^{OT} = \{out_w^{OT}\}_{w \in W_{eva}$$

We'll prove that the the latter holds, then it can be deduced that both views are indistinguishable.

Considering the order in which the garbler sends the inputs to the OT instance, we'll denote $W_{evaluator} = (w_0, w_1, ..., w_n)$, and denote i^* the first index where exists $c \in \{0, 1\}$ such that the garbler sent either $(\mathbf{guess}, c)$ or $(\mathbf{transfer}, l_0, l_1)$ (where l_c is invalid label of wire w) to $\mathcal{F}_{PVOT}^{w}$. $i^* = \perp$ if such index doesn't exist.

For $j \in \{3, 4\}$, denote the following RVs:

$$OUT_j^{OT,prefix} = (OUT_j^{OT,w_0}, ..., OUT_j^{OT,w_{i^*-1}})$$
$$OUT_j^{OT,suffix} = (OUT_j^{OT,w_{i^*}}, ..., OUT_j^{OT,w_n})$$

Note that for all $i \in \{0, 1, ..., i^* - 1\}$, $OUT_3^{OT,w_i} = OUT_4^{OT,w_i} = \perp$ — the garbler doesn't get any output in case of no cheating. Hence:

$$\Pr[OUT_3^{OT,prefix} = out_3^{OT,prefix} \mid VIEW_3^{pref} = view^{pref}, M_3 = m]$$
$$= \Pr[OUT_4^{OT,prefix} = out_4^{OT,prefix} \mid VIEW_4^{pref} = view^{pref}, M_4 = m]$$

For readability purposes, we'll merge the RVs $VIEW_i^{pref}$ and $OUT_i^{OT,pref}$.

As for wires $w_{i^*}, ..., w_n$: in hybrid #4 the simulator gets the honest evaluator's input and performs an honest simulation of the evaluator using the value used for b, and also sets the value for randomness m. Hence, for the same value m in both hybrids (in addition to the same prefix of the view), $OUT_3^{OT,suffix}, OUT_4^{OT,suffix}$ are identical, i.e.:

$$\Pr[OUT_3^{OT,suffix} = out_3^{OT,suffix} \mid VIEW_3^{pref} = view^{pref}, M_3 = m]$$
$$= \Pr[OUT_4^{OT,suffix} = out_4^{OT,suffix} \mid VIEW_4^{pref} = view^{pref}, M_4 = m]$$

The last thing remaining to prove is that $\Pr[M_3 = m] = \Pr[M_4 = m]$.
In hybrid #3 the evaluator selects the value of m using an independent uniformly-distributed random-variable, therefore:

$$\Pr(M_3 = m \mid VIEW_3^{pref} = view^{pref}) = 0.5$$

In hybrid #4, the value of m depends on the correctness of the wire labels sent by the garbler.

Denote $X_{detected}$ the event that the adversary was detected and X_{cheat} the event that the garbler attempted cheating. We'll consider only the case where $i^* \neq \perp$ ($i^* = \perp$ is the trivial case where $OUT^{OT} = OUT^{OT,prefix}$). Let y_i denote the

evaluator's i'th input bit. Denote $p_4^{cheat} = \Pr(M_4 = m \mid X_{cheat}, VIEW_4^{pref} = view^{pref})$, hence:

$$\begin{aligned}
p_4^{cheat} &= \Pr(M_4 = m \wedge X_{detected} \mid X_{cheat}, VIEW_4^{pref} = view^{pref}) \\
&+ \Pr(M_4 = m \wedge \overline{X_{detected}} \mid X_{cheat}, VIEW_4^{pref} = view^{pref}) \\
&= \Pr(X_{detected} \mid X_{cheat}, VIEW_4^{pref} = view^{pref}) \\
&\cdot \underbrace{\Pr(M_4 = m \mid X_{detected}, X_{cheat}, VIEW_4^{pref} = view^{pref})}_{0.5\ (*)} \\
&+ \Pr(\overline{X_{detected}} \mid X_{cheat}, VIEW_4^{pref} = view^{pref}) \\
&\cdot \underbrace{\Pr(M_4 = m \mid \overline{X_{detected}}, X_{cheat}, VIEW_4^{pref} = view^{pref})}_{0.5\ (**)} \\
&= 0.5 \cdot \Big(\Pr(\overline{X_{detected}} \mid X_{cheat}, VIEW_4^{pref} = view^{pref}) \\
&\qquad + \Pr(X_{detected} \mid X_{cheat}, VIEW_4^{pref} = view^{pref}) \Big) \\
&\underset{\text{(complementary events)}}{=} 0.5
\end{aligned}$$

Hence, $\Pr(M_3 = m \mid VIEW_3^{pref} = view^{pref}) = \Pr(M_4 = m \mid VIEW_4^{pref} = view^{pref}) = 0.5$ and therefore the views in Hybrid #3 and Hybrid #4 are identically distributed.

(*) If the garbler is detected, $m = y_i \oplus c$ (as specified in item 6(a)iB or item 6(b)iB). While y_i is constant, c is uniformly-distributed. Hence:

$$\begin{aligned}
&\Pr[M_4 = m \mid X_{detected}, X_{cheat}, VIEW_4^{pref} = view^{pref}] \\
&= \Pr[M_4 = c \oplus y_i \mid X_{detected}, X_{cheat}, VIEW_4^{pref} = view^{pref}] \\
&\underset{c \sim U(0,1)}{=} 0.5
\end{aligned}$$

(**) If the garbler isn't detected, $m = 1 - y_i \oplus c$ (as specified in item 6(a)iA or item 6(b)iA). While y_i is constant, c is uniformly-distributed. Hence:

$$\begin{aligned}
&\Pr[M_4 = m \mid \overline{X_{detected}}, X_{cheat}, VIEW_4^{pref} = view^{pref}] \\
&= \Pr[M_4 = 1 - c \oplus y_i \mid \overline{X_{detected}}, X_{cheat}, VIEW_4^{pref} = view^{pref}] \\
&\underset{c \sim U(0,1)}{=} 0.5
\end{aligned}$$

4. **Hybrid #4 - Hybrid #5** — Denote $VIEW$ the set of possible views to be generated in hybrid #4. We'll show that for any $view \in VIEW$:

$$\Pr[VIEW_4 = view] = \Pr[VIEW_5 = view]$$

For $i \in \{4,5\}$, denote $VIEW_i^{pref} = (B_i, \{OUT_i^{OT,w}\}_{w \in W_{evaluator}}), VIEW_i^{suff} = (VIEW_i^{arbitration})$. Hence, for $i \in \{4,5\}$:

$$\Pr[VIEW_i = view] = \Pr[(VIEW_i^{pref}, LABELS_i, OUT_i, VIEW_i^{suff}) = (view^{pref}, labels, out, view^{su}$$
$$= \Pr[VIEW_i^{pref} = view^{pref}] \cdot \Pr[OUT_i = out, LABELS_i = labels \mid VIEW_i^{pref} = view^{pref}]$$
$$\cdot \Pr[VIEW_i^{suff} = view^{suff} \mid VIEW_i^{pref} = view^{pref}, OUT_i = out, LABELS_i = labels]$$

Both hybrids behave the same way before choosing *output*. Hence,

$$\Pr[VIEW_4^{pref} = view^{pref}] = \Pr[VIEW_5^{pref} = view^{pref}]$$

Moreover, assuming that the value of *output* chosen is both hybrids is the same, the rest of the simulation is the same. Hence,

$$\Pr[VIEW_4^{suff} = view^{suff} \mid (VIEW_4^{pref}, LABELS_4, OUT_4) = view^{pref}, labels, output] = \Pr[VIEW_5^{s}$$

Therefore,

$$\Pr[VIEW_4 = view] = \Pr[VIEW_5 = view] \iff \Pr[LABELS_4 = labels, OUT_4 = out \mid VIEW_4^{pref} =$$

Also, the labels sent to the garbler in both hybrids correspond to the output received by the evaluator, hence for all tuples of labels and plaintext values $(labels^*, output^*)$:

$$\Pr[LABELS_4 = labels^* \mid OUT_4 = output^*] = \Pr[LABELS_5 = labels^* \mid OUT_5 = output^*]$$

Hence,

$$\Pr[VIEW_4 = view] = \Pr[VIEW_5 = view] \iff \Pr[OUT_4 = out \mid VIEW_4^{pref} = view^{pref}] = \Pr[OUT$$

We'll prove the latter, then it can be deduced that both views are indistinguishable.

Let X_{cheat} an event indicating that a cheating occurred before item 9 in the ideal

world simulator. Hence,

$$
\begin{aligned}
&\Pr[OUT_5 = output \mid VIEW_5^{pref} \\
&= view^{pref}] = \Pr[OUT_5 = output \wedge X_{cheat} \mid VIEW_5^{pref} = view^{pref}] \\
&\quad + \Pr[OUT_5 = output \wedge \overline{X_{cheat}} \mid VIEW_5^{pref} = view^{pref}] \\
&= \underbrace{\Pr[X_{cheat} \mid VIEW_5^{pref} = view^{pref}]}_{(*)} \\
&\quad \cdot \Pr[OUT_5 = output \mid X_{cheat}, VIEW_5^{pref} = view^{pref}] \\
&\quad + \underbrace{\Pr[\overline{X_{cheat}} \mid VIEW_5^{pref} = view^{pref}]}_{(*)} \\
&\quad \cdot \Pr[OUT_5 = output \mid \overline{X_{cheat}}, VIEW_5^{pref} = view^{pref}] \\
&= \Pr[X_{cheat} \mid VIEW_4^{pref} = view^{pref}] \\
&\quad \cdot \underbrace{\Pr[OUT_5 = output \mid X_{cheat}, VIEW_5^{pref} = view^{pref}]}_{(**)} \\
&\quad + \Pr[\overline{X_{cheat}} \mid VIEW_4^{pref} = view^{pref}] \\
&\quad \cdot \Pr[OUT_5 = output \mid \overline{X_{cheat}}, VIEW_5^{pref} = view^{pref}] \\
&= \Pr[X_{cheat} \mid VIEW_4^{pref} = view^{pref}] \\
&\quad \cdot \Pr[OUT_4 = output \mid X_{cheat}, VIEW_4^{pref} = view^{pref}] \\
&\quad + \Pr[\overline{X_{cheat}} \mid VIEW_4^{pref} = view^{pref}] \\
&\quad \cdot \underbrace{\Pr[OUT_5 = output \mid \overline{X_{cheat}}, VIEW_5^{pref} = view^{pref}]}_{(***)} \\
&= \Pr[X_{cheat} \mid VIEW_4^{pref} = view^{pref}] \\
&\quad \cdot \Pr[OUT_4 = output \mid X_{cheat}, VIEW_4^{pref} = view^{pref}] \\
&\quad + \Pr[\overline{X_{cheat}} \mid VIEW_4^{pref} = view^{pref}] \\
&\quad \cdot \Pr[OUT_4 = output \mid \overline{X_{cheat}}, VIEW_5^{pref} = view^{pref}] \\
&= \Pr[OUT_4 = output \mid VIEW_4^{pref} = view^{pref}]
\end{aligned}
$$

Therefore, $\Pr[OUT_5 = output \mid VIEW_5^{pref} = view^{pref}] = \Pr[OUT_4 = output \mid VIEW_4^{pref} = view^{pref}]$

(*) $VIEW_5^{pref}$ is identically distributed as $VIEW_4^{pref}$, hence the probability of cheating in both hybrids is identically distributed

(**) Assuming a cheating attempt occurred, the simulator receives the honest party's inputs and in this case a perfect honest simulation is generated

(***) Assuming no cheating attempt occurred, the sender sent a valid evaluation garbled circuit, hence the honest evaluator evaluates the correct value of

$f(x_{garbler}, x_{evaluator})$ in hybrid #4, while in hybrid #5 he receives the same value, but from $\mathcal{F}_{\varepsilon-\text{one-shot}}$

□

5.2. Simulatability of Corrupt Evaluator

The following describes a simulator for simulating a corrupt-evaluator's view, assuming the ability to access ideal functionalities $\mathcal{F}_{PVOT}$ and $\mathcal{F}_{comm}$:

1. Simulate execution of a real-world garbler (protocol $\Pi^{OS}_{garbler}$ [2.1]), using input $x^*_{garbler} \leftarrow 0^n$, until reaching line 16 (i.e. sending evaluation circuit)
2. $\mathcal{S}$ extracts the evaluator's input $x_{evaluator}$ by observing the evaluator's messages $(\textbf{choose}, v_w)$ to $\mathcal{F}^w_{PVOT}$ (for each $w \in W_{evaluator}$)
3. $\mathcal{S}$ sends $x_{evaluator}$ to $\mathcal{F}_{\varepsilon-\text{one-shot}}$, and receive $y = f(x_{garbler}, x_{evaluator})$ as a response.
4. $\mathcal{S}$ builds a fake garbled-circuit (denoted GC_{fake}) that outputs $f(x_{garbler}, x_{evaluator})$ for any given inputs
   - GC_{fake} is constructed from a valid garbled-circuit by replacing the output gates' truth table.
5. $\mathcal{S}$ sends $Open(GC_{1-b}, GC_{fake})$ to the evaluator (i.e. equivocates the evaluation circuit decommitment to be GC_{fake})
6. Receive the output labels' decommitment (denoted $outputDecommitments$) from the evaluator.
7. $res \leftarrow$ VerifyOutputLabelsCorrectness$(Comms, outputDecommitments)$
   - If res indicates an abort, send **blatant-cheat** to $\mathcal{F}_{\varepsilon-\text{one-shot}}$ and receive the honest garbler's input $x_{garbler}$.

Claim 5.2.1. *The corrupt-evaluator's view in the real-world model and the ideal-world model are indistinguishable.*

Proof. The environment's view consists of a tuple of messages, denoted by the following RVs:

$$\left(COMMS, GC^{ver}, \{L^w\}_{w\in W_{evaluator}}, \{L^w\}_{w\in W_{garbler}}, GATES^{eval}, OUT, VIEW^{arb}\right)$$

which correspond to the messages received by the receiver in protocol $\Pi^{OS}_{evaluator}$ [2.3] — $COMM$ correspond to the garbled circuits' commitments sent in line 3, GC^{ver} corresponds to the opening of the verification circuit in line 6, L^w correspond to the output obtained from $\mathcal{F}^w_{PVOT}$ received at line 15 (for each $w \in W_{evaluator}$), $\{L^w\}_{w\in W_{garbler}}$ corresponds to the garbler's input wires' labels sent in line 20, $GATES^{eval}$ corresponds to the

evaluation circuit's gates opening that is sent in line 6, OUT corresponds to the honest party's output and $VIEW^{arb}$ corresponds to the messages sent during arbitration.

We'll use a hybrid argument in order to prove this claim. The different hybrids are defined as follows:

1. **Hybrid #1 (i.e real world)**
2. **Hybrid #2** — same as hybrid #1, except that the simulator doesn't receive the evaluator's input. The garbler's input $x_{garbler}$ is given to the simulator, same as previous hybrid. The simulator for this world is described as follows:
   a) Simulate a real-world execution using the honest receiver's inputs
      - After line 15 in protocol $\Pi^{OS}_{evaluator}$ [2.3] (receive inputs from OT), execute item 2 from the ideal-world simulator (extract evaluator's input).
3. **Hybrid #3** — same as hybrid #2, except that we equivocate the evaluation garbled circuit. The garbler's input $x_{garbler}$ is given to the simulator, same as previous hybrid. The simulator for this world is described as follows:
   a) Run the simulator presented in previous hybrid
      - Instead of sending the opening of GC_{1-b} (i.e. the evaluation circuit) in the real-world protocol at line 16, execute items 3 to 5 in the ideal-world simulator (equivocate the circuit's opening to a hardcoded-output circuit).
4. **Hybrid #4 (i.e. ideal world)** — same as hybrid #3, except that we don't receive the garbler's input $x_{garbler}$. Instead, we set its value to 0^n.

For hybrid $\#i$, denote the following RVs:

$$\left(COMMS_i, GC_i^{ver}, \{L_i^w\}_{w\in W_{evaluator}}, \{L_i^w\}_{w\in W_{garbler}}, GATES_i^{eval}, OUT_i, VIEW_i^{arb}\right)$$

We'll prove indistinguishability between adjacent hybrid worlds:

1. **Hybrid #1 - Hybrid #2** — Denote $VIEW_i = (VIEW_i^{pref}, VIEW^{suff})$ (for $i \in \{1,2\}$), where $VIEW_i^{pref}, VIEW^{suff}$ correspond to the messages received before and after line 15, respectively.

   The evaluator's inputs aren't used in the real world protocol before line 15, hence

   $$\Pr[VIEW_1^{pref} = view^{pref}] = \Pr[VIEW_2^{pref} = view^{pref}]$$

   Moreover, both hybrids are identical before reaching line 15, therefore we need to prove that

   $$\Pr[VIEW_1^{suff} = view^{suff} \mid VIEW_1^{pref} = view^{pref}] = \Pr[VIEW_2^{suff} = view^{suff} \mid VIEW_2^{pref} = vie$$

In hybrid #2, the simulator extracts the corrupt evaluator's inputs w.p. 1 before sending the messages in $VIEW^{suff}$. Meaning that the simulator case perform an honest simulation of the evaluator in hybrid #1, i.e.:

$$\Pr[VIEW_1^{suff} = view^{suff} \mid VIEW_1^{pref} = view^{pref}] = \Pr[VIEW_2^{suff} = view^{suff} \mid VIEW_2^{pref} = vie$$

2. **Hybrid #2 - Hybrid #3** — We rely on the fact that for each gate in the circuit, the evaluator is able to decrypt only a single entry of the gate (otherwise, he breaks the encryption scheme).

   Denote $g_1, ..., g_k$ the gates of the evaluation garbled circuit, topologically sorted (w.r.t. evaluation order). We'll define a series of hybrids $H_{2,1}, ..., H_{2,k}$, where in each hybrid we change a single gate in the evaluation circuit. We'll denote hybrid #2 as $H_{2,0}$ and hybrid #3 as $H_{2,k}$.

   Formally, in hybrid $H_{2,i}$ (for all $i \in [1,k]$) we replace the outputs at gates $g_1, ...g_i$ to be 0^n, at all entries except the one that can be evaluated by the evaluator. Also, denote $VIEW_{2,i}$ the RV corresponding to the view generated in hybrid $H_{2,i}$, and G_j^i the RV corresponding to the value of gate g_j in hybrid i (for all $j \in [1,k]$).

   Therefore, for all $i \in [1,k]$, hybrids $H_{2,i-1}$ and $H_{2,i}$ differ only at the value of gate g_i. This means that $VIEW_{2,i-1}$ and $VIEW_{2,i-1}$ are indistinguishable iff G_i^{i-1} and G_i^i are indistinguishable. Denote $(K_0^0, K_1^0), (K_0^1, K_1^1), (K_0^{out}, K_1^{out})$ the RVs corresponding to the gate's input and output wires' labels. Hence,

$$G_i^{i-1} = \{E_{K_i^0}(E_{K_j^1}(K_{i\wedge j}^{out}))\}_{i,j\in\{0,1\}^2}$$

   Assuming that the evaluator can open only the first entry of G_i^{i-1}, then:

$$G_i^i = \{E_{K_0^0}(E_{K_0^1}(K_1^{out}))\} \cup \{E_{K_i^0}(E_{K_j^1}(0^n))\}_{(i,j)\in\{0,1\}^2\setminus\{0^2\}}$$

   For all $(i,j) \in \{0,1\}^2 \setminus \{0^2\}$, at least one of K_i^0, K_j^1 is uniformly distributed. Using the encryption scheme's semantic security property we can deduce that

$$E_{K_i^0}(E_{K_j^1}(K_{i\wedge j}^{out})) \approx E_{K_i^0}(E_{K_j^1}(0^n))$$

   Therefore $G_i^{i-1} \approx G_i^i$ and also $VIEW_{2,i-1} \approx VIEW_{2,i}$ for all $i \in [1,k]$.

3. **Hybrid #3 - Hybrid #4** — We'll define a series of hybrids, where in each hybrid we change a single input gate in the evaluation circuit — change all entries' encrypted wire labels to be the same as the entry that can be decrypted by the evaluator's inputs.
   Two adjacent hybrids are indistinguishable due to the encryption scheme's semantic security property.

□

5.3. Accountability and Defamation-Freeness

Lemma 5.3.1. *Assuming the input-wire labels are valid, if the evaluator reaches an invalid gate g in the evaluation garbled-circuit then its two inputs are valid*

Proof. By induction using the invalid gate's depth (denoted k)

$\underline{k = 1}$: Gate g is using the input-wires' labels that were sent by the garbler. If these labels contained an invalid label, the protocol's execution would halt at line 16. Hence, gate g has two valid input labels.

$\underline{k < n}$: Assuming the claim's correctness for depth $k < n$

$\underline{k = n}$: Let's falsely assume the claim is false – i.e. g has an input label l^* corresponding to wire w which is an invalid label. Wire w can be either an input-wire or an output-wire of another gate g':

- If w is an input-wire, l^* is an invalid input-wire label that was sent by the garbler. This contradicts the claim's assumption that the input-wire labels are valid.
- If w is an output-wire of gate g', it means g' is a gate that evaluated an invalid wire-label. Therefore g' is an invalid gate located at depth $< n$, which contradicts the definition of g (being an invalid gate at the minimal depth n).

We reached a contradiction in both possibilities, therefore our false assumption was wrong and hence the claim is correct. □

Claim 5.3.2. *Our one-shot protocol satisfies the "Accountability" definition (c.f. item 2 in definition A.3.1).*

Proof. We'll prove that for any abort triggered by the honest party, the arbitrator accuses the adversary. We'll do it by enumerating the cases where the honest party aborts and show that in each case he is able to generate a proof-of-malfeasance that's used by the arbitrator in order to accuse the adversary.

First, we'll consider the case where the protocol's execution halted because the garbler claimed the evaluator is corrupt. By observing the garbler's protocol, the only conditions where he aborts are as follows:

1. **Invalid evaluated output-wires labels (c.f. line 23 in protocol $\Pi^{OS}_{garbler}$ [2.1])** – the garbler reached this line in the protocol iff the output-labels' verification process (c.f. protocol VerifyOutputLabelsCorrectness [2.1]) returned a complaint. Inspecting this subroutine, a complaint is returned at line 5 iff an invalid output-wire label $\pi_{l^w_{b_w}}$ (for output-wire w) decommitment is detected by the garbler. Hence, the garbler aborts iff he holds an invalid output-wire label $\pi_{l^w_{b_w}}$ (for output-wire w) that was sent by the evaluator. Then, the garbler sends the invalid decommitment with the corresponding commitment (which is signed by both parties) to the arbitrator.

   The arbitrator executes protocol HandleInvalidGarbledOutput [3.6] for the given complaint. He checks that the decommitment is signed by the evaluator and the

commitment is signed by both parties. This check passes due to the inputs sent by the garbler, as described above.

Finally, the arbitrator checks at line 8 if the provided decommitment is an invalid decommitment of the specified output-wire. This check fails because an honest garbler sends an invalid output-wire label. Then, the arbitrator blames the corrupt evaluator.

Next, we'll consider the case where the protocol's execution halted because the evaluator claimed the garbler is corrupt. By observing the evaluator's protocol, the only conditions where he aborts are as follows:

1. **Invalid verification garbled-circuit (c.f. line 11 in protocol $\Pi^{OS}_{evaluator}$ [2.3])** – The evaluator aborts **iff** the verification garbled-circuit's checking subroutine (c.f. protocol VerifyCircuit [2.3]) returns a complaint.

   In this subroutine, a complaint is returned in these lines:

   - line 5 – the evaluator reaches this line if protocol OpenAllWireLabelsDecommitments [3.1] returns a complaint, which happens iff the evaluator received an invalid wire-label decommitment $\pi_{l^w_b}$ (for wire w and wire-value b). In this case, the evaluator sends an **InvalidOpening** complaint containing the invalid decommitment and the corresponding component's commitment.
   - line 10 – the evaluator reaches this line if protocol OpenAllWireMasksDecommitments [3.3] returns a complaint, which happens iff the evaluator received an invalid wire-mask decommitment π_{m_w} (for wire w). In this case, the evaluator sends an **InvalidOpening** complaint containing the invalid decommitment and the corresponding component's commitment.
   - line 15 – the evaluator reaches this line if protocol OpenAllGatesDecommitments [3.5] returns a complaint, which happens iff the evaluator received an invalid gate decommitment π_g (for gate g). In this case, the evaluator sends an **InvalidOpening** complaint containing the invalid decommitment and the corresponding component's commitment.
   - line 20 – the evaluator reaches this line iff the protocol VerifyAllGatesOutput [2.7] returns a complaint, which happens iff there exists a gate g, two gate input-labels $l^{w_0}_{b_0}, l^{w_1}_{b_1}$ (and their corresponding masks m_{w_0}, m_{w_1}) that outputs either an invalid output label or an invalid NAND output.
     In this case, the evaluator sends an **InvalidVerificationGate** containing the invalid gate g, both used inputs $l^{w_0}_{b_0}, l^{w_1}_{b_1}$, the wires' corresponding masks m_{w_0}, m_{w_1} and the corresponding components' commitments.

   As described above, two types of complaints are sent to the arbitrator. We'll describe how the arbitrator handles each complaint and why the corrupt garbler is accused:

   - **InvalidOpening**– In all cases above where the evaluator sends this complaint, he provides an invalid component decommitment (either wire-label,

wire-mask or gate) and its corresponding commitment, both signed by the garbler.

The arbitrator executes protocol HandleInvalidOpening [3.1] for the given complaint. He verifies whether both commitment and decommitment were signed by the same party, this check passes because both are signed by the garbler. Then, the arbitrator checks in line 5 if the given decommitment is invalid – as we described in each of the three relevant cases above, the evaluator held an invalid decommitment and sent it to the arbitrator as part of the complaint. Therefore the check in line 5 returns $\perp$, therefore the garbler is accused.

- **InvalidVerificationGate**– Following the abort occurred at line 20 (as described above), the arbitrator receives a valid gate, wire-masks are wire-labels (otherwise, an "InvalidDecommitment" complaint should have been sent before). Hence, the checks at line 6, line 11 and line 19 are skipped. However, the provided gate evaluates an invalid output, given the provided inputs. Hence, the check at line 26 (which is the same check as in the evaluator's main subroutine, line 20) returns $\perp$ and therefore the garbler is accused.

2. **Invalid input-labels for evaluation circuit (c.f. line 18 in protocol $\Pi^{OS}_{evaluator}$ [2.3])** – The evaluator aborts **iff** protocol GetEvaluatorInputLabels [2.6] returns a complaint.

   Inspecting the relevant subroutine, a complaint is returned at either line 6 or line 14 iff the evaluator received an invalid input-wire label decommitment $\pi_{l^w_{b_w}}$ (for wire w) that corresponds to the garbler or evaluator, respectively. If a complaint returned in line 6, the evaluator sends an **InvalidGarblerInputLabel** complaint, containing the invalid decommitment $\pi_{l^w_{b_w}}$ and the commitments of both wire-labels of w.

   The arbitrator executes protocol HandleInvalidGarblerInputLabel [3.4] for the given complaint. He checks that both commitments and decommitment are signed by the garbler. Then, the check at line 7 passes because the evaluator passed a decommitment which doesn't match either the first or second labels associated with the wire w. Therefore, the garbler is accused.

   If a complaint returned in line 14, the evaluator sends an **InvalidEvaluator-InputLabel** complaint, containing the wire w, its corresponding value and the commitment of the corresponding label.

   The arbitrator executes protocol HandleInvalidEvaluatorInputLabel [3.5] for the given complaint. He checks that the commitmentis signed by the garbler. Then, the check at line 10 passes because the evaluator passed a decommitment which doesn't match either the first or second labels associated with the wire w. Therefore, the garbler is accused.

3. **Invalid gate in evaluation circuit (c.f. line 31 in protocol $\Pi^{OS}_{evaluator}$ [2.3])** – the evaluator aborts **iff** the garbled-circuit's evaluation subroutine (c.f. proto-

col EvaluateGarbledCircuit [2.4]) returned a complaint.
Inspecting the relevant subroutine, a complaint is returned in two lines:

- line 8 – the evaluator reaches this case iff he holds an invalid decommitment π_g of gate g that was sent by the garbler.

  The evaluator sends an **InvalidOpening** complaint, containing the invalid gate decommitment and its corresponding commitment. Because π_g is invalid, the check at line 3 fails and therefore the garbler is accused.

- line 14 – the evaluator reaches this case iff the garbler reached a gate g that evaluates an invalid output-label, given two inputs $l^{w_0}_{b_{w_0}}, l^{w_1}_{b_{w_1}}$ (where the gate was sent by the gabler).

  The evaluator sends an **InvalidEvaluationGate** complaint, containing $\pi_g, \pi_{l^{w_0}_{b_{w_0}}}, \pi_{l^{w_1}_{b_{w_1}}}$ and the commitments of the corresponding components.

  The arbitrator executes protocol HandleInvalidEvaluationGate [3.3] for the given complaint and checks if the provided decommitments are valid – the check at line 7 passes (if π_g was invalid, the evaluator would abort at line 8 at the evaluator's main subroutine). The checks at line 10 fails, by lemma 5.3.1.

  As defined above, g evaluates an invalid output-label decommitment, using inputs $l^{w_0}_{b_{w_0}}, l^{w_1}_{b_{w_1}}$. Therefore, the check at line 15 passes and therefore the garbler is accused.

□

Claim 5.3.3. *Our one-shot protocol maintains the "Defamation-Free" definition (c.f. item 3 in definition A.3.1).*

Proof. We'll prove that the honest party can't be falsely accused by showing that all complaints (considering the sender and the complaint type) sent by the adversary to the arbitrator can't cause a false accusation of the honest party.

Due to claim 5.3.2, the arbitrator will always accuse the adversary for complaints presented by the honest party and therefore defamation-freeness is guaranteed in this case.

Hence, if the corrupt party wants to frame the honest party it would falsely claim one of the following:

1. **Invalid decommitment (c.f. protocol HandleInvalidOpening [3.1])** – in order to blame the honest party, the adversary must send the arbitrator two messages (signed by the honest party) including the commitment and its corresponding invalid decommitment. Because the honest party follows the protocol, he will never send an invalid decommitment.

   We assume the signature scheme is secure, hence the adversary might frame the honest party with negligible probability.

2. **Invalid verification-circuit component (i.e. gate or wire-label) (c.f. protocol HandleInvalidVerificationGate [3.2])** – in order to blame the garbler, the corrupt evaluator must provide a gate that outputs an invalid output given two specific inputs (which are also provided by him). The garbler follows the protocol, therefore all gates generated by him are valid NAND gates – i.e. each combination of valid inputs would output a valid label and its corresponding value equals to $NAND(input_0, input_1)$.
   We assume the signature scheme is secure, hence the adversary might frame the garbler with negligible probability.

3. **Invalid evaluation-circuit component (i.e. gate or wire-label) (c.f. protocol HandleInvalidEvaluationGate [3.3])** – in order to blame the garbler, the corrupt evaluator must provide a gate that outputs an invalid output given two specific inputs (which are also provided by him). The garbler follows the protocol, therefore all gates generated by him are valid – i.e. each combination of valid inputs would output a valid label.
   We assume the signature scheme is secure, hence the adversary might frame the garbler with negligible probability.

4. **Invalid garbler's input-wire label (c.f. protocol HandleInvalidGarblerInputLabel [3.4])** – in order to blame the garbler, the corrupt evaluator must provide a message (sent by the garbler) including an invalid input-wire label decommitment. An honest garbler follows the protocol and therefore will always send valid decommitments.

   We assume the signature scheme is secure, hence the adversary might frame the garbler with negligible probability.

5. **Invalid evaluator's input-wire label (c.f. protocol HandleInvalidEvaluatorInputLabel [3.5])** – in order to blame the garbler, the corrupt evaluator must provide a wire w and an invalid input-wire label decommitment (that was sent by $\mathcal{F}^w_{PVOT}$). An honest garbler follows the protocol and therefore will always send valid decommitments.

   We assume the signature scheme is secure, hence the adversary might frame the garbler with negligible probability.

6. **Invalid evaluated output (c.f. protocol HandleInvalidGarbledOutput [3.6])** – in order to blame the evaluator, the corrupt garbler must send the arbitrator a message (signed by the evaluator) including an invalid output-wire label. By inspecting the evaluator's protocol, an honest evaluator will check the correctness of the evaluated output-labels before sending them back to the garbler. If the evaluated labels are invalid, the honest party will abort and won't send them to the garbler. Otherwise, the honest party will send valid labels *only* to the garbler.

   We assume the signature scheme is secure, hence the adversary might frame the evaluator with negligible probability.

□

6. Dual-Execution Protocol: Overview

We define a new protocol, in the $\mathcal{F}_{\varepsilon-\text{one-shot}}$-hybrid model, where the parties run two parallel executions of a protocol modeled by $\mathcal{F}_{\varepsilon-\text{one-shot}}$, each execution in a different role. After both executions end (unless a **cheat-detected** message is returned from $\mathcal{F}_{\varepsilon-\text{one-shot}}$), the parties compare their evaluated outputs. The honest user detects cheating if the outputs differ.

Note that $\mathcal{F}_{\varepsilon-\text{one-shot}}$ provides asymmetrical security guarantees — only one of the parties is able to cheat, and the honest party can generate a proof of the inputs he used. Hence, by performing a dual execution of $\mathcal{F}_{\varepsilon-\text{one-shot}}$, each party acts as an honest party in *at least* one computation and can prove which input he used (in the corresponding instance of $\mathcal{F}_{\varepsilon-\text{one-shot}}$), if needed to. This suffices to guarantee output integrity in our dual-execution protocol, since the adversary *cannot* cheat in both execution.

The function evaluated in each instance of $\mathcal{F}_{\varepsilon-\text{one-shot}}$ (denoted g) outputs, in addition to $f(x_0, x_1)$, two (keyed) hashes corresponding to the parties' inputs where the hashes' keys are secretly shared between the parties, and each party will also receive the key corresponding to their input's evaluated hash.

$$\begin{aligned} g(x_0, a_0^0, a_0^1, r_0; x_1, a_1^0, a_1^1, r_1) &= \underbrace{f(x_0, x_1)}_{y} || H_{a_0^0 \oplus a_1^0}(x_0) || H_{a_0^1 \oplus a_1^1}(x_1) || (\underbrace{a_0^0 \oplus a_1^0}_{k_0} \oplus r_0) || (\underbrace{a_0^1 \oplus a_1^1}_{k_1} \oplus r_1) \\ &= y || \underbrace{H_{k_0}(x_0)}_{m_0} || \underbrace{H_{k_1}(x_1)}_{m_1} || \underbrace{(k_0 \oplus r_0)}_{k_0'} || \underbrace{(k_1 \oplus r_1)}_{k_1'} \end{aligned}$$

where inputs a_i^0, a_i^1, r_i are picked by party P_i (for both $i \in \{0, 1\}$), and H is the keyed-hash function.

We prove that g has the property that different inputs of the adversary result in different evaluated outputs with high-enough probability to detect cheating and satisfy ε-covert security.

In case of different outputs in both executions, the accused party will have to send a plaintext evaluation of g to the honest party (using the inputs broadcast from both $\mathcal{F}_{\varepsilon-\text{one-shot}}$ instances). If the outputs differ, while the honest party used the same inputs and the circuit evaluated by the adversary corresponds to the function g (because the adversary couldn't alter the output in this execution), therefore either (1) the adversary used inconsistent inputs or (2) the output evaluated by the honest party doesn't correspond to the function g. This means that the plaintext circuit sent by the adversary, which outputs a value that differs from the value evaluated by the honest party, will contain either (1) an invalid gate, or (2) an output wire value that doesn't correspond to

the output received by the honest party, or (3) an input value that doesn't correspond to the adversary's input.

Hence, the honest party provides a proof containing the corrupt component (either a gate, or an input/output wire and its two inconsistent values that were sent by the adversary) in order to accuse the adversary.

For a full, formal specification of the protocol, see protocol $\Pi_{P_i}^{DE}$ [3.1].

6.1. Ideal functionality $\mathcal{F}_{dual-exec}$

Functionality $\mathcal{F}_{dual-exec}$ guarantees output correctness, and allows cheating w.p. ε. The formal description of the functionality appears in fig. 6.1.1

Figure 6.1.1.: Functionality $\mathcal{F}_{dual-exec}$

Inputs: Party P_i sends a message $(\textbf{input}, x_i)$ (for his input x_i) to $\mathcal{F}_{dual-exec}$, while $\mathcal{A}$ sends input on behalf of the corrupted party.
Attempted Cheat: If $\mathcal{F}_{dual-exec}$ receives (**cheat**) from corrupt party P_i, $\mathcal{F}_{dual-exec}$ decides whether to reveal the honest party's inputs or not:

- **cheat-detected:** With probability $1 - \varepsilon$, $\mathcal{F}_{dual-exec}$ sends $(\textbf{cheat-detected}, P_i)$ to all parties and aborts.
- **cheat-undetected:** With probability ε, $\mathcal{F}_{dual-exec}$ sends the honest party's inputs to P_i.

6.2. Security Analysis

Our main theorem is as follows:

Theorem 6.2.1. *The protocol Π^{DE} realizes $\mathcal{F}_{dual-exec}$ functionality with malicious security.*

$\mathcal{F}_{dual-exec}$ is an extension of covert security — because $\mathcal{F}_{dual-exec}$ doesn't allow the adversary to choose the honest party's output in the case of an undetected cheating attempt — therefore theorem 6.2.1 implies the following:

Corollary 6.2.2. *For any two-party function f, Π_{DE}^{f} realizes $\mathcal{F}_f$ with ε-covert security, public interactive arbitration and output correctness.*

Proof Sketch of theorem 6.2.1

The full proof of theorem 6.2.1 appears in chapter 8. Below we give a high-level overview of our security proof. We construct a simulator $\mathcal{S}$ that runs $\mathcal{A}$ internally and interacts

with the trusted party that computes f. The protocol is symmetric, therefore w.l.o.g. we can assume party P_0 is honest and P_1 is corrupt.

We need to show that (1) cheating is detected with probability at least $1 - \varepsilon$ even if the output is *not* different (this is required to satisfy "standard" ε-covert security) and (2) that the arbitrator (not just the honest user) will detect cheating whenever the honest user does (this is required to satisfy covert security with interactive arbitration).

Cheating is detected

To do this, we modify the function f that is computed by $\mathcal{F}_{\varepsilon-\text{one-shot}}$, and instead compute a function g that includes, in addition to the output of f, two keyed-hash values that correspond to the parties' input values. The key corresponding to each hash value is secret-shared between the parties. By using a keyed-hash function that realizes definition 7.1.1, we ensure that if he adversary attempts to give different inputs the hash values will be different with probability at least $1-\varepsilon$. (In order to prevent the adversary from making the input of one execution dependent on the other, we make use of the input-commitment phase provided by $\mathcal{F}_{\varepsilon-\text{one-shot}}$).

Of course, by adding to the inputs of g the key-shares for the keyed hash functions, we add an additional avenue for cheating: the adversary can give consistent inputs, but inconsistent key shares. We handle this by giving, in the output of g, the full key for the hash corresponding to the party's own input (i.e., party P_0 gets the full key used to hash x_0, and party P_1 the full key used to hash x_1). This means that giving inconsistent key shares for x_0 will *always* be detected (since the honest party will receive a different full key in each execution). On the other hand, we can simulate ourselves whether or not the adversary is caught in this case, because we can fully simulate the hash computation (we can sample a key-share, and then run the hash on the actual input x_1).

Detected cheating is provable

Now, we will describe the mechanism ensuring that an honest party is always able to generate a complaint against the adversary in case of different outputs, and on the other hand an adversary cannot falsely accuse the honest party. We rely on the guarantee, provided by $\mathcal{F}_{\varepsilon-\text{one-shot}}$, that the evaluator *can't* cheat, and he can provide a publicly-verifiable proof of his used input.

For all $i \in \{0,1\}$, denote x_i^g, x_i^e the inputs of party i to the instances of $\mathcal{F}_{\varepsilon-\text{one-shot}}$ where P_i acted as a garbler and evaluator, respectively, and f_i is the function evaluated in $\mathcal{F}^i_{\varepsilon-\text{one-shot}}$.

If the evaluated output of f is different in both executions of $\mathcal{F}_{\varepsilon-\text{one-shot}}$, it means that:

$$\underbrace{f_1(x_0^e, x_1^g)}_{\text{evaluated by } P_0} \neq \underbrace{f_0(x_0^g, x_1^e)}_{\text{evaluated by } P_1}$$

We assumed P_0 is honest, therefore it can be deduced that $x_0^g = x_0^e$ and $f_0 = f$. Hence,

the following are true:

$$\underbrace{f_0(x_0^g, x_1^e)}_{\text{evaluated by } P_1} = f_0(x_0^e, x_1^e) = f(x_0^e, x_1^e) \tag{6.2.1}$$

$$\underbrace{f_1(x_0^e, x_1^g)}_{\text{evaluated by } P_0} \neq \underbrace{f(x_0^e, x_1^e)}_{\text{evaluated by } P_1} \tag{6.2.2}$$

Thus, the corrupt party either used inconsistent inputs (i.e. $x_1^g \neq x_1^e$), or the honest party evaluated a corrupt circuit (i.e. $f_1 \neq f$).

In case P_i complains for output inconsistency, P_{1-i} should send him (privately, not through the arbitrator) a plaintext evaluation of f using the inputs x_0^e, x_1^e and gets accused iff the circuit is invalid or the output of this circuit doesn't match the output evaluated by P_i:

- If P_0 complains, he receives from P_1 a plaintext evaluation of $f(x_0^e, x_1^e)$ which differs from P_0's output $f_1(x_0^e, x_1^g)$ (by eq. (6.2.2)). Therefore the plaintext evaluated circuit must contain at least one invalid evaluated NAND gate, or an invalid input bit (i.e., that is different from the corresponding bit in x_0^e or x_1^e) or an invalid output bit (i.e., that is different from the corresponding output bit of $f_1(x_0^e, x_1^g)$).

  Therefore, P_0 rebuts using the invalid component (i.e., invalid NAND gate or invalid input/output bit) and as a result P_1 (i.e. the corrupt party) will be accused.

- If P_1 complains (i.e., falsely accuses P_0), he receives from P_0 a plaintext evaluation of $f(x_0^e, x_1^e)$ which equals to P_1's output $f_0(x_0^g, x_1^e)$ (by eq. (6.2.1)). Therefore the plaintext evaluated circuit provided by P_0 will be valid, and will have the same output as evaluated by P_1.

  Therefore, P_1 can't respond with a rebuttal that falsely accuses P_0 and as a result P_1 (i.e. the corrupt party) will be accused.

7. Full Description of the Dual-Execution Protocol

In this chapter we give a full, formal description of our protocol to realize, for any function f, the functionality $\mathcal{F}_f$ with $\frac{1}{2}$-Covert security and public interactive arbitration.

7.1. Notations

Our construction makes use of hash functions that satisfy a slightly-stronger form of pairwise independence, and an additional "invertibility".

with two special properties.

Formally, let $\mathcal{H}^{n_I,n_O,n_K} = \{H_k\}_{k\in\{0,1\}^{n_K}}$ be a family of hash functions, mapping n_I input bits to n_O output bits.

Definition 7.1.1 (Key-XOR pairwise independence)**.** We say $\mathcal{H}^{n_I,n_O,n_K}$ is *Key-XOR pairwise independent* if for all $m, m' \in \{0,1\}^{n_I}$, $z \in \{0,1\}^{n_K}$, if $m \neq m'$ then for all $t, t' \in \{0,1\}^{n_O}$

$$\Pr_{k\xleftarrow{\$}\{0,1\}^{n_K}}\left[H_k(m) = t \wedge H_{k\oplus z}(m') = t'\right] = 2^{-2n}$$

(i.e., $H_k(m)$ and $H_{k\oplus z}(m')$ are uniformly and independently distributed)

Note that this is strictly a stronger than pairwise independent (which is a special case of this definition when $z = 0$).

Definition 7.1.2 (Invertibility)**.** We say $\mathcal{H}^{n_I,n_O,n_K}$ is *invertible* if there exists an efficient sampler **Samp** such that for any $m' \in \{0,1\}^{n'}$, the distributions

$$(U_n, H_{U_n}(m)) \approx_C (\mathbf{Samp}(m, U_{n'}), U_{n'})$$

are computationally indistinguishable (where U_j denotes the uniform distribution on j bits).

Corollary 7.1.3. *We note that when n' is logarithmic in the security parameter, any pairwise-independent hash functions will satisfy definition 7.1.2 (i.e. the invertibility property), since **Samp** can use rejection sampling (i.e., **Samp**(m,t) repeatedly chooses a uniform key k until $H_k(m) = t$; this will require, in expectation, $2^{n'}$ attempts).*

7.2. Direct-Product Hash

We define the direct-product family of hash functions $\mathcal{H}_{\text{direct}}^{n_I,n_O,(n_I+1)\cdot n_O}$ as follows: The space of keys consists of all $(n_I+1)\times n_O$ binary matrices (we can think of K as a boolean string of length $(n_I+1)\cdot n_O$), and $H_K(m) = K\cdot(m||1)^T$ (this is matrix multiplication over $GF[2]$; think of m as a binary vector of length n_I which we extend to n_I+1 by adding an extra 1 element).

Lemma 7.2.1. *$\mathcal{H}_{direct}^{n_I,n_O,(n_I+1)\cdot n_O}$ satisfies definition 7.1.1.*

Proof. Denote $x = m \oplus m'$. Note that since $m_{n_I+1} = 1 = m'_{n_I+1}$, $x \neq m$. Let $t, t' \in \{0,1\}^{n_O}$ Let $p = \Pr_{K\xleftarrow{\$}\{0,1\}^{n\cdot n'}}\left[H_K(m) = t \wedge H_{K\oplus Z}(m') = t'\right]$

$$
\begin{aligned}
p &= \Pr_{K\xleftarrow{\$}\{0,1\}^{n\cdot n'}}\left[H_K(m) = t \wedge H_{K\oplus Z}(m') = t'\right] \\
&= \Pr_{K\xleftarrow{\$}\{0,1\}^{n\cdot n'}}\left[K\cdot m^T = t \wedge K\cdot m^T \oplus K\cdot x^T \oplus Z\cdot m^T \oplus Z\cdot x^T = t'\right] \\
&= \Pr_{K\xleftarrow{\$}\{0,1\}^{n\cdot n'}}\left[K\cdot m^T = t\right]\cdot \Pr\left[K\cdot m^T \oplus K\cdot x^T \oplus Z\cdot m^T \oplus Z\cdot x^T = t' | K\cdot m^T = t\right] \\
&= 2^{-n}\cdot \Pr\left[t \oplus K\cdot x^T \oplus Z\cdot m^T \oplus Z\cdot x^T = t' | K\cdot m^T = t\right] \\
&= 2^{-n}\cdot \Pr\left[K\cdot x^T = t \oplus t' \oplus Z\cdot m^T \oplus Z\cdot x^T | K\cdot m^T = t\right]
\end{aligned}
$$

Since $x \neq m$, $K\cdot m^T$ and $K\cdot x^T$ are independent, hence

$$
\begin{aligned}
&= 2^{-n}\cdot \Pr\left[K\cdot x^T = t \oplus t' \oplus Z\cdot m^T \oplus Z\cdot x^T\right] \\
&= 2^{-2n}\ .
\end{aligned}
$$

□

7.3. Dual-execution protocol main routines

The main routine of party P_i is described in protocol $\Pi_{P_i}^{DE}$ [3.1], where protocol $\Pi_{P_i\text{-rebuttal}}^{DE}$ [3.2] is invoked if the other party broadcasts a complaint.

The arbitrator's main routine is described in protocol Π_{arb}^{DE} [3.3]. It uses protocol HandleDEComplaint [3.4] for handling the parties' complaints (if such are published).

7.4. Dual-execution protocol auxiliary routines

The parties perform two executions of $\mathcal{F}_{\varepsilon-\text{one-shot}}$ and compare their outputs. If the output differ, P_i asks from P_{1-i} a plaintext evaluation of the circuit that was supposed to be evaluated in $\mathcal{F}_{\varepsilon-\text{one-shot}}$.

Then, P_i calls protocol CheckIfConsistent [3.1] in order to check that (1) the given plaintext evaluation of the circuit is valid (i.e., each gate evaluates a valid NAND gate),

Protocol $\Pi_{P_i}^{DE}$ **[3.1]** Implementing $\mathcal{F}_{dual-exec}$ in the $\mathcal{F}_{\varepsilon-\text{one-shot}}$-hybrid model

function $\Pi_{P_i}^{DE}(x_i)$
  // Send inputs to one-shot functionality
  P_i samples random values $a_i^0, a_i^1 \leftarrow \{0,1\}^*$
  Let $z_i = \bar{z}_i = (x_i, a_i^0, a_i^1)$, the inputs to $\mathcal{F}_{\varepsilon-\text{one-shot}}^{1-i}, \mathcal{F}_{\varepsilon-\text{one-shot}}^{i}$ respectively.
  Send (**input**, z_i) to $\mathcal{F}_{\varepsilon-\text{one-shot}}^{1-i}$, and (**input**, $\bar{z}_i$) to $\mathcal{F}_{\varepsilon-\text{one-shot}}^{i}$
  Wait to receive (**committed**) messages from $\mathcal{F}_{\varepsilon-\text{one-shot}}^{0}$ and $\mathcal{F}_{\varepsilon-\text{one-shot}}^{1}$

  // Handle cheating attempts in one-shot protocol
  if received **cheat-detected** from $\mathcal{F}_{\varepsilon-\text{one-shot}}^{1-i}$ **then**
    return $\perp$
  end if
  Send **reveal** to both $\mathcal{F}_{\varepsilon-\text{one-shot}}^{0}$ and $\mathcal{F}_{\varepsilon-\text{one-shot}}^{1}$
  Wait to receive $out^j \leftarrow$ (**output**, $\{v_w^j \mid w$ is an output-wire of $C_f\}$) from $\mathcal{F}_{\varepsilon-\text{one-shot}}^{j}$ (for all $j \in \{0,1\}$)
  For all $j \in \{0,1\}$, denote $out^j = y^j||t_0^j||t_1^j||k_0'^j||k_1^j$

  // Handle inconsistent outputs
  if $y^0||t_0^0||t_1^0||k_i^0 \neq y^1||t_0^1||t_1^1||k_i^1$ **then** //
    Send **prove-input** to $\mathcal{F}_{\varepsilon-\text{one-shot}}^{i}$
    Wait to receive (**prove-input**, $\overline{z_{1-i}}$) from $\mathcal{F}_{\varepsilon-\text{one-shot}}^{1-i}$
    Wait to receive $V(C_f) = \{v_w, \sigma(v_w) \mid v_w$ is the evaluated value of wire $w\}$ from P_{1-i}
    $res \leftarrow$ CheckIfConsistent$(V(C_f), \overline{z_0}, \overline{z_1}, Comm(\overline{z_{1-i}}), out_0)$ // c.f. protocol CheckIfConsistent [3.1]
    if res is a complaint **then**
      Broadcast complaint res
      return $\perp$
    end if
  end if
  if prove-input message was sent to $\mathcal{F}_{\varepsilon-\text{one-shot}}^{1-i}$ **then** // i.e., P_{1-i} asks for a plaintext evaluation of C_f
    Send **prove-input** to $\mathcal{F}_{\varepsilon-\text{one-shot}}^{i}$
    Wait to receive (**prove-input**, $\overline{z_{1-i}}$) from $\mathcal{F}_{\varepsilon-\text{one-shot}}^{1-i}$
    Evaluate C_f using inputs $\overline{z_0}, \overline{z_1}$
    Send $\{v_w, \sigma(v_w) \mid v_w$ is the evaluated value of wire $w\}$ to P_{1-i}
  end if
  return Out_0 // doesn't matter which output we choose, both Out_0, Out_1 are equal
end function

Protocol $\Pi^{DE}_{P_i\text{-rebuttal}}$ [3.2] The rebuttal routine of the dual-execution protocol

function $\Pi^{DE}_{P_i\text{-REBUTTAL}}$
end function

Protocol Π^{DE}_{arb} [3.3] The arbitration routine of the dual-execution protocol

function Π^{DE}_{arb}
    Wait for a complaint $\Big(reason, (message)\Big)$. If received (**cheat-detected**, id) from $\mathcal{F}^0_{\varepsilon-\text{one-shot}}$ or $\mathcal{F}^1_{\varepsilon-\text{one-shot}}$ then **return** (**cheat-detected**, id).
    if received a complaint **then**
        Denote P_i the sender of the complaint
        $res \leftarrow$ HandleDEComplaint$(P_i, reason, message)$
        if $res \neq \perp$ **then**
            return res
        end if
    end if
end function

Protocol HandleDEComplaint [3.4] The arbitration routine of the dual-execution protocol

function $HandleDEComplaint(id, reason, message)$
    if $reason =$ **invalid-plain-wire then**
        parse $message$ as a wire-value v, a signature σ and a wire w.
        $res \leftarrow$ HandleInvalidPlainWire(v^*_w, w) // c.f. protocol HandleInvalidPlainWire [4.1]
    else if $reason =$ **invalid-plain-gate** and $id = id_{evaluator}$ **then**
        parse $message$ as a plain gate g, and a set containing tuples of wire-values and signatures.
        $res \leftarrow$ HandleInvalidPlainGate$(g, \{v_w \mid w \in g\})$ // c.f. protocol HandleInvalidVerificationGate [3.2]
    end if
    return res
end function

(2) the plaintext evaluated output matches the output evaluated by $\mathcal{F}^{1-i}_{\varepsilon-\text{one-shot}}$, and (3) the used inputs in the plaintext circuit match the inputs used by P_0 and P_1 in $\mathcal{F}^{1}_{\varepsilon-\text{one-shot}}$ and $\mathcal{F}^{0}_{\varepsilon-\text{one-shot}}$ (respectively).

Protocol CheckIfConsistent [3.1] Check correctness of plain circuit's evaluation (subroutine of protocol $\Pi^{DE}_{P_i}$ [3.1])

function CheckIfConsistent($V(C_f), \overline{z_0}, \overline{z_1}, Comm(\overline{z_{1-i}}), out_0$) // Checks P_{1-i}'s innocence
  for $i \in \{0,1\}$ **do**
    for P_i's input wire $w \in C_f$ **do**
      Denote $v_w, \sigma(v_w) \in V(C_f)$ the plaintext value of wire w evaluated in C_f by P_{1-i} and its corresponding signature
      if $v_w \neq \overline{z_i}[w]$ **then**
        Send (**prove-input**, w) to $\mathcal{F}^{i}_{\varepsilon-\text{one-shot}}$
        return complaint $\Big($**invalid-plain-wire**$, (w, v_w)\Big)$
      end if
    end for
  end for
  for output wire $w \in C_f$ **do**
    Denote $v_w, \sigma(v_w) \in V(C_f)$ the plaintext value of wire w evaluated in C_f by P_{1-i} and its corresponding signature
    if $v_w \neq out_0[w]$ **then**
      Send (**prove-output**, w) to $\mathcal{F}^{i}_{\varepsilon-\text{one-shot}}$
      return complaint $\Big($**invalid-plain-wire**$, (w, v_w)\Big)$
    end if
  end for
  for gate $g \in C_f$ **do**
    Denote $w_{in_0}, w_{in_1}, w_{out}$ the input and output wires of gate g
    Denote $v_w, \sigma(v_w) \in V(C_f)$ the plaintext value of wire w evaluated in C_f by P_{1-i} and its corresponding signature (for all $w \in \{w_{in_0}, w_{in_1}, w_{out}\}$)
    if $v_{w_{out}} \neq \overline{v_{w_{in_0}} \oplus v_{w_{in_1}}}$ **then**
      return complaint $\Big($**invalid-plain-gate**$, (g, \{v_{w_{in_0}}, v_{w_{in_1}}, v_{w_{out}}\})\Big)$
    end if
  end for
  return $\perp$
end function

7.4.1. Arbitration subroutines

protocol HandleInvalidPlainWire [4.1] handles a complaint of party P_i, where either (1) a plaintext value of an input wire doesn't correspond to P_0 and P_1's input values that

were used while executing $\mathcal{F}^{1}_{\varepsilon-\text{one-shot}}$ and $\mathcal{F}^{0}_{\varepsilon-\text{one-shot}}$ (respectively), or (2) a plaintext value of an output wire doesn't correspond to the output evaluated by $\mathcal{F}^{1-i}_{\varepsilon-\text{one-shot}}$.

Protocol HandleInvalidPlainWire [4.1] Handle complaint of an invalid plaintext wire in case of inconsistent outputs in dual execution

// Any party can access this routine
// Denote P_i the party sending the complaint
function HandleInvalidPlainWire(v^*_w, w)
  if v^*_w isn't signed by P_{1-i} and w is neither an output wire nor one of P_i's input wires **then**
    return $\left(\textbf{cheat-detected}, P_i\right)$
  end if
  if w is an output wire and didn't received message $(\textbf{prove-output}, w, v_w)$ from $\mathcal{F}^{i}_{\varepsilon-\text{one-shot}}$ **then**
    return $\left(\textbf{cheat-detected}, P_i\right)$
  else if w is an input wire and didn't received message $(\textbf{prove-input}, w, v_w)$ from $\mathcal{F}^{i}_{\varepsilon-\text{one-shot}}$ **then**
    return $\left(\textbf{cheat-detected}, P_i\right)$
  end if
  Denote v_w the value sent in the corresponding message from $\mathcal{F}^{i}_{\varepsilon-\text{one-shot}}$ // either $(\textbf{prove-output}, w, v_w)$ or $(\textbf{prove-input}, w, v_w)$
  if $v^*_w \neq v_w$ **then**
    return $\left(\textbf{cheat-detected}, P_{1-i}\right)$
  else
    return $\left(\textbf{cheat-detected}, P_i\right)$
  end if
end function

protocol HandleInvalidPlainGate [4.2] handles a complaint of party P_i, where a plaintext values correspoding to a gate's input and output wires don't correspond to a NAND gate. P_{1-i} is accused iff the given plaintext wires' values are signed by P_{1-i}, and they don't correspond to a NAND gate.

Protocol HandleInvalidPlainGate [4.2] Handle complaint of an invalid plaintext gate in case of inconsistent outputs in dual execution

```
    // Any party can access this routine
    // Denote P_i the party sending the complaint
 1: function HandleInvalidPlainGate(g, {v_w | w ∈ g})
 2:     Denote W^g = {w_in0, w_in1, w_out} the input and output wires of gate g
 3:     for w ∈ W^g do
 4:         if v_w isn't signed by P_{1-i} as a value of wire w then
 5:             return (cheat-detected, P_i)
 6:         end if
 7:     end for
 8:     if v_wout ≠ NAND(v_win0, v_win1) then
 9:         return (cheat-detected, P_{1-i})
10:     else
11:         return (cheat-detected, P_i)
12:     end if
13: end function
```

8. Dual-Execution Protocol Security Analysis

8.1. Simulatability

For clarity purposes, we'll denote x_i the private input of party P_i that was transmitted to $\mathcal{F}^{1-i}_{\varepsilon-\text{one-shot}}$, and by $\bar{x}_i$ the input that was transmitted to $\mathcal{F}^{i}_{\varepsilon-\text{one-shot}}$.

The simulator $\mathcal{S}$ works as follows (w.l.o.g. we assume P_0 is corrupt):

1. Until receiving P_0's input messages to $\mathcal{F}^{0}_{\varepsilon-\text{one-shot}}, \mathcal{F}^{1}_{\varepsilon-\text{one-shot}}$, do the following upon receiving new message from P_0:

Case 1: If P_0 sent (**input**, z_0) to $\mathcal{F}^{0}_{\varepsilon-\text{one-shot}}$, simulates $\mathcal{F}^{0}_{\varepsilon-\text{one-shot}}$ sending (**committed**) to P_0

Case 2: If P_0 sent (**input**, $\overline{z_0}$) to $\mathcal{F}^{1}_{\varepsilon-\text{one-shot}}$, simulates $\mathcal{F}^{1}_{\varepsilon-\text{one-shot}}$ sending (**committed**) to P_0

Case 3: If sent **cheat** to $\mathcal{F}_{\varepsilon-\text{one-shot}}$, set $a^0_1, a^1_1 \xleftarrow{\$} \{0,1\}^n$ and simulate an honest execution of P_1

Case 4: If received any other message, ignore

2. Observe P_0's inputs $z_0 = (x_0, a^0_0, a^1_0)$ and $\overline{z_0} = (\overline{x_0}, \overline{a^0_0}, \overline{a^1_0})$:

Case 1: $z_0 = \overline{z_0}$, set $a^0_1 \xleftarrow{\$} \{0,1\}^n$ and continue to item 3 in the simulator

Case 2: $(a^1_0, x_0) = (\overline{a^1_0}, \overline{x_0})$

i. Set $a^0_1 \xleftarrow{\$} \{0,1\}^n$, calculate $k_0 \leftarrow a^0_0 \oplus a^0_1$ and $\overline{k_0} \leftarrow \overline{a^0_0} \oplus \overline{a^0_1}$

ii. If $H_{k_0}(x_0) \neq H_{\overline{k_0}}(x_0)$, send **blatant-cheat** and simulate an honest execution of P_1 using randomness a^0_1 and $a^1_1 \xleftarrow{\$} \{0,1\}^n$.

iii. Otherwise, continue to item 3 in the simulator

Case 3: $(a^1_0, x_0) \neq (\overline{a^1_0}, \overline{x_0})$, send **cheat** to $\mathcal{F}_{dual-exec}$, receive a response r and the honest party's input x_1:

i. If $f(x_0, x_1) \neq f(\overline{x_0}, x_1)$ or $a^1_0 \neq \overline{a}^1_0$, send **blatant-cheat** to $\mathcal{F}_{dual-exec}$ and set $a^0_1 \xleftarrow{\$} \{0,1\}^*$

ii. Otherwise (i.e. $f(x_0, x_1) = f(\overline{x_0}, x_1)$ and $a^0_0 = \overline{a^0_0}$), observe the response r received from $\mathcal{F}_{dual-exec}$:

Case 1: $r =$ **cheat-detected**

- Repeatedly sample $k_0 \xleftarrow{\$} \{0,1\}^{n_K}$ until $H_{k_0}(x_0) \neq H_{k_0 \oplus a_0^0 \oplus \overline{a_0^0}}(\overline{x_0})$.
- Let $a_1^0 \leftarrow k_0 \oplus a_0^0$.

Case 2: $r =$ **cheat-undetected**

- For $i \in [0,l]$: sample $k_{0,i} \xleftarrow{\$} \{0,1\}^{n_K}$.
- If $\forall i \in [0,l]$: $H_{k_{0,i}}(x_0) \neq H_{k_{0,i} \oplus a_0^0 \oplus \overline{a_0^0}}(\overline{x_0})$, send **blatant-cheat** to $\mathcal{F}_{dual-exec}$ and set $a_1^0 \leftarrow k_{0,0} \oplus a_0^0$.
- Otherwise, let $i_{\neq}, i_{=} \in [0,l]$ the first indices s.t.

$$H_{k_{0,i_{\neq}}}(x_0) \neq H_{k_{0,i_{\neq}} \oplus a_0^0 \oplus \overline{a_0^0}}(\overline{x_0})$$
$$H_{k_{0,i_{=}}}(x_0) = H_{k_{0,i_{=}} \oplus a_0^0 \oplus \overline{a_0^0}}(\overline{x_0})$$

- w.p. l^* send **blatant-cheat** to $\mathcal{F}_{dual-exec}$ and set $a_1^0 \leftarrow k_{0,i_{\neq}} \oplus a_0^0$.
- w.p. $1 - l^*$ set $a_1^0 \leftarrow k_{0,i_{=}} \oplus a_0^0$ and continue to item 3 in the simulator.

Simulate an honest execution of P_1 using randomness a_1^0 as picked above, and $a_1^1 \xleftarrow{\$} \{0,1\}^n$.

3. Set $t_0 \leftarrow H_{a_0^0 \oplus a_1^0}(x_0)$
4. Calculate $t_1 \xleftarrow{\$} \{0,1\}^{n'}$
5. Denote $z_0 = (x_0, a_0^0, a_0^1)$. Send x_0 to $\mathcal{F}_{dual-exec}$ and receive the response $y = f(x_0, x_1)$.
6. Until receiving P_0's **reveal** messages to $\mathcal{F}^0_{\varepsilon-\text{one-shot}}, \mathcal{F}^1_{\varepsilon-\text{one-shot}}$, do the following upon receiving new message from P_0:

Case 1: If P_0 sent **reveal** to $\mathcal{F}^0_{\varepsilon-\text{one-shot}}$, simulates $\mathcal{F}^0_{\varepsilon-\text{one-shot}}$ sending (**output**, (y, t_0, t_1)) to P_0

Case 2: If P_0 sent **reveal** to $\mathcal{F}^1_{\varepsilon-\text{one-shot}}$, simulates $\mathcal{F}^1_{\varepsilon-\text{one-shot}}$ sending (**output**, (y, t_0, t_1)) to P_0

Case 3: If received any other message, ignore

7. If P_0 sent a complaint during arbitration phase, send **blatant-cheat** to $\mathcal{F}_{dual-exec}$ and receive the honest party's input x_1:
   - Compute $k^* \leftarrow \textbf{Samp}(x_1, t_1)$ using the input x_1, and set $a_1^1 \leftarrow k^* \oplus a_0^1$
   - Simulate an honest execution using a_1^0, a_1^1.

Claim 8.1.1. *The corrupt party's view in the real-world model and the ideal-world model are indistinguishable.*

Proof. The environment's view consists of a tuple of messages, denoted by the following RVs:

$$C, \overline{C}, RES^{OS}, \overline{RES^{OS}}, OUT, VIEW^{arb}$$

where $C, \overline{C}$ are indicators of the events that **committed** message was sent from each instance of $\mathcal{F}_{\varepsilon-\text{one-shot}}$, RES^{OS} and $\overline{RES^{OS}}$ correspond to the output of both instances of $\mathcal{F}_{\varepsilon-\text{one-shot}}$, OUT corresponds to the honest party's output, and $VIEW^{arb}$ corresponds to the arbitration's view.
The output of $\mathcal{F}_{\varepsilon-\text{one-shot}}$ (denoted OUT^{OS}) consists of a tuple of RVs: $OUT^{OS} = (Y, TAG^0, TAG^1, K^0)$, where Y corresponds to $f(x_0^*, x_1^*)$ (where P_0, P_1 sent inputs x_0^*, x_1^*), TAG^j corresponds to the hash of input x_j^* (for all $j \in \{0,1\}$), and K^0 is the hash key used for calculating TAG^0. The same construction is made also for $\overline{OUT^{OS}}$. Also, denote $VIEW_{arb} = (X_1, A_1^0, A_1^1)$, RVs corresponding to P_1's input, and the hash keys' shares.

We'll use a hybrid argument in order to prove this claim. The different hybrids are defined as follows:

1. **Hybrid #1 (i.e. real world)**

2. **Hybrid #2** — same as previous hybrid, except that we modify the one-shot protocol to output a uniformly random value instead of P_1's input hash. The honest party's input x_1 is given to the simulator, same as previous hybrid.

   The simulator for this world is described as follows:

   a) Execute the real world protocol, with the following changes:

      i. Change the operation of $\mathcal{F}_{\varepsilon-\text{one-shot}}$ to output $t_1 \xleftarrow{\$} \{0,1\}^n$ instead of $H_{a_0^1 \oplus a_1^1}(x_1)$

      ii. If **cheat** was sent, calculate $k^* \leftarrow \textbf{Samp}(x_1, t_1)$ and set $a_1^1 \leftarrow k^* \oplus a_0^1$ (the value of a_1^1 is revealed only during arbitration).

3. **Hybrid #3** — same as previous hybrid, except that we extract the corrupt party's inputs. The honest party's input x_1 is given to the simulator, same as previous hybrid.

   The simulator for this world is described as follows:

   a) Execute the simulator of previous hybrid, using the honest party's input, with the following changes:

      - Instead of lines 5 to 10 in protocol $\Pi_{P_i}^{DE}$ [3.1] (sending inputs to $\mathcal{F}_{\varepsilon-\text{one-shot}}$), execute item 1 in the ideal world simulator (extract P_0's input). Denote $z_0, \overline{z_0}$ the corrupt party's inputs.
      - Execute the real world protocol protocol $\Pi_{P_i}^{DE}$ [3.1] starting at line 11 using input x_1 and randomness a_1^0, a_1^1 as set in line 3 in protocol $\Pi_{P_i}^{DE}$ [3.1].

4. **Hybrid #4** — same as previous hybrid, except that we modify the honest party's randomness a_1^0 (corresponding to x_0's hash key share). The honest party's input x_1 is given to the simulator, same as previous hybrid.

   The simulator for this world is described as follows:

   a) Execute the simulator of previous hybrid, using the honest party's input, with the following changes:

      - Instead of sampling a_1^0 at line 3 in protocol $\Pi_{P_i}^{DE}$ [3.1], execute item 2 in the ideal world simulator (after extracting corrupt inputs at item 1).
      - Execute the real world protocol protocol $\Pi_{P_i}^{DE}$ [3.1] starting at line 11 using input x_1 and randomness a_1^0, a_1^1 as set in item 1 in the ideal world simulator.

5. **Hybrid #5** — same as previous hybrid, except that we change the operation of $\mathcal{F}_{\varepsilon-\text{one-shot}}$: instead of computing $f(x_0, x_1)$, we use the output of $\mathcal{F}_{dual-exec}$. The honest party's input x_1 is given to the simulator, as in previous hybrid.

   The simulator for this world is described as follows:

   a) Execute the simulator of previous hybrid, using the honest party's input, with the following changes:

      - Instead of lines 11 to 12 in protocol $\Pi_{P_i}^{DE}$ [3.1] (sending **reveal** messages to $\mathcal{F}_{\varepsilon-\text{one-shot}}$ and waiting to receive output), execute items 5 to 6 in the ideal world simulator (receive output from $\mathcal{F}_{dual-exec}$).
      - Execute the real world protocol protocol $\Pi_{P_i}^{DE}$ [3.1] starting at line 13 using input x_1 and randomness a_1^0, a_1^1 as set in item 1 in the ideal world simulator.

6. **Hybrid #6** — same as previous hybrid, except that we don't receive the honest party's input.

   The simulator for this world is described as follows:

   a) Execute the simulator of previous hybrid, setting the honest party's input to be 0^n.

   b) If sent **cheat**, use the honest party's *real* input, as received from $\mathcal{F}_{dual-exec}$

7. **Hybrid #7 (i.e. ideal world)** — the only difference between this hybrid and the previous is that, instead of sending the output of the simulated honest party and simulated arbitrator to the corresponding ideal-world parties, the output they receive in the ideal world is from $\mathcal{F}_{dual-exec}$.

For each hybrid $\#i$, denote $VIEW_i$ a RV of the view generated in hybrid #i, which consists of the following RVs:

$$VIEW_i = \left(C_i, \overline{C_i}, \underbrace{(Y_i, TAG_i^0, TAG_i^1, K_i^0)}_{OUT_i^{OS}}, \underbrace{(\overline{Y_i}, \overline{TAG_i^0}, \overline{TAG_i^1}, \overline{K_i^0})}_{\overline{OUT_i^{OS}}}, OUT_i, VIEW_i^{arb} \right)$$

We'll prove indistinguishability between adjacent hybrid worlds:

1. **Hybrid #1 – Hybrid #2** —Denote $VIEW$ the set of possible views to be generated in hybrid #1. We'll show that for any $view \in VIEW$:

   $$\Pr[VIEW_1 = view] = \Pr[VIEW_2 = view]$$

   The only thing changed from hybrid #1 are the values of $TAG_2^1, \overline{TAG_2^1}$ and $A_{1,2}^1$.

   Considering the event $\neg X_{same}$, the simulator simulates an honest execution of P_1 and therefore

   $$\Pr[VIEW_1 = view \mid \neg X_{same}] = \Pr[VIEW_2 = view \mid \neg X_{same}]$$

   Considering the event X_{same}, $TAG_1^1 = \overline{TAG_1^1}$ because in hybrid #1 the same hash key is used in both executions of $\mathcal{F}_{\varepsilon-\text{one-shot}}$ (both parties use consistent key shares) and P_1 sends x_1 to both instances. $TAG_2^1 = \overline{TAG_2^1}$ by definition at **??** in the simulator. Hence,

   $$\begin{aligned}
   &|\Pr[VIEW_1 = view \mid \neg X_{same}] \\
   &\quad - \Pr[VIEW_2 = view \mid \neg X_{same}]| < neg(n) \\
   &\iff \\
   &|\Pr[TAG_1^1 = t_1, A_{11}^1 = a_1^1 \mid \neg X_{same}] \\
   &\quad - \Pr[TAG_2^1 = t_1, A_{12}^1 = a_1^1 \mid \neg X_{same}]| < neg(n)
   \end{aligned}$$

   In hybrid #1 (TAG_1^1, A_{11}^1) are calculated by generating A_{11}^1's value and setting TAG_1^1 accordingly, and in hybrid #2 (TAG_2^1, A_{12}^1) are set using the simulator $\mathcal{S}_H$.

   The hashing scheme in the protocol realizes definition 7.1.2, therefore it holds that $(U_n, H_{U_n}(m)) \approx (\mathbf{Samp}(m, U_{n'}), U_{n'})$, hence

   $$(TAG_1^1, A_{11}^1) \approx (TAG_2^1, A_{12}^1)$$

   therefore:

   $$\left|\Pr[TAG_1^1 = t_1, A_{11}^1 = a_1^1 \mid \neg X_{same}] - \Pr[TAG_2^1 = t_1, A_{12}^1 = a_1^1 \mid \neg X_{same}]\right| < neg(n)$$

2. **Hybrid #2 – Hybrid #3** — Denote $VIEW$ the set of possible views to be generated in hybrid #2. We'll show that for any $view \in VIEW$:

   $$\Pr[VIEW_2 = view] = \Pr[VIEW_3 = view]$$

   For hybrid $\#i$ $(i \in \{2, 3\})$, denote $VIEW_i^{suf} = (OS_i, \overline{OS_i}, OUT_i, VIEW_i^{arb})$. The only thing changed from hybrid #2 are the values of $C, \overline{C}$.

   In hybrid #2 we simulate sending x_1 to $\mathcal{F}_{\varepsilon-\text{one-shot}}$, and then — by the definition of $\mathcal{F}_{\varepsilon-\text{one-shot}}$ — $C_2 \neq \perp$ iff P_0 sends his input.

In hybrid #3 we change the operation of $\mathcal{F}_{\varepsilon-\text{one-shot}}$ to set $C_3 \neq \perp$ iff P_0 sends his input. Hence, $C_2 = \perp \iff C_3 = \perp$.
The same argument holds for the RVs $\overline{C_2}, \overline{C_3}$. Hence,

$$\Pr[C_2 = c, \overline{C_2} = \overline{c}] = \Pr[C_3 = c, \overline{C_3} = \overline{c}]$$

Given the same prefix $c, \overline{c}$ in these hybrids, both simulators run the same code, using the same information information determined so far (in this particular case — none). Hence, the rest of the simulation is identically distributed. Hence,

$$\begin{aligned}&\Pr[VIEW_2^{suf} = view^{suf} \mid C_2 = c, \overline{C_2} = \overline{c}] \\ &\quad = \Pr[VIEW_3^{suf} = view^{suf} \mid C_3 = c, \overline{C_3} = \overline{c}]\end{aligned}$$

3. **Hybrid #3 – Hybrid #4** — Denote $VIEW$ the set of possible views to be generated in hybrid #3. We'll show that for any $view \in VIEW$:

$$\Pr[VIEW_3 = view] = \Pr[VIEW_4 = view]$$

Let $z_0 = (x_0, a_0^0, a_0^1)$ and $\overline{z_0} = (\overline{x_0}, \overline{a_0^0}, \overline{a_0^1})$ the inputs sent from P_0 to $\mathcal{F}_{\varepsilon-\text{one-shot}}, \overline{\mathcal{F}_{\varepsilon-\text{one-shot}}}$.
Denote the following events:

- X_{P_0-same} — $x_0 = \overline{x_0}$ and $a_0^0 = \overline{a_0^0}$.
- $X_{diff-blatent}$ — $x_0 \neq \overline{x_0}$ or $a_0^0 \neq \overline{a_0^0}$, and $f(x_0, x_1) \neq f(\overline{x_0}, x_1)$ or $a_0^1 \neq \overline{a}_0^1$
- X_{P_0-diff} — $x_0 \neq \overline{x_0}$ or $a_0^0 \neq \overline{a_0^0}$, and $f(x_0, x_1) = f(\overline{x_0}, x_1)$ and $a_0^1 = \overline{a}_0^1$
- $X_{no-cheat}$ — P_0 sent the same input to $\mathcal{F}^0_{\varepsilon-\text{one-shot}}, \mathcal{F}^1_{\varepsilon-\text{one-shot}}$ and no cheating attempt was made.

Considering the event X_{P_0-same}, P_0 is detected in hybrid #3 iff x_1's hash key is different in both executions, i.e. $a_0^1 \oplus A_{1,3}^1 \neq \overline{a_0^1} \oplus A_{1,3}^1$. Hence,

$$\Pr[P_0 \text{ is detected} \mid X_{P_0-same}] = \Pr[a_0^1 \oplus A_{1,3}^1 \neq \overline{a_0^1} \oplus \overline{A_{1,3}^1} \mid X_{P_0-same}]$$

the honest party sets $A_{1,3}^1, \overline{A_{1,3}^1}$ s.t. $A_{1,3}^1 = \overline{A_{1,3}^1}$

$$= \Pr[a_0^1 \neq \overline{a_0^1} \mid X_{P_0-same}]$$

In hybrid #4, P_0 is detected iff $a_0^1 \neq \overline{a_0^1}$, hence:

$$\begin{aligned}\Pr[P_0 \text{ is detected in 4} \mid X_{P_0-same}] &= \Pr[a_0^1 \neq \overline{a_0^1} \mid X_{P_0-same}] &(8.1.1)\\ &= \Pr[P_0 \text{ is detected in 3} \mid X_{P_0-same}] &(8.1.2)\end{aligned}$$

Considering the event $X_{diff-blatent}$, P_0 is detected in hybrid #3 w.p. 1 because either $Y_3 \neq \overline{Y_3}$ (if $f(x_0, x_1) \neq f(\overline{x_0}, x_1)$) or the honest party's hash key, which is

part of the output, is different in both executions (if $a_0^1 \neq \overline{a}_0^1$, while $a_0^1 = \overline{a}_1^1$, as they are set by the honest party). Hence,

$$\Pr[P_0 \text{ is detected} \mid X_{diff-blatent}] = 1$$

and in hybrid #4, P_0 is detected w.p. 1, hence

$$\Pr[P_0 \text{ is detected in } 4 \mid X_{diff-blatent}] = 1 \tag{8.1.3}$$
$$= \Pr[P_0 \text{ is detected in } 3 \mid X_{diff-blatent}] \tag{8.1.4}$$

Considering the event X_{P_0-diff}, P_0 is detected in hybrid #3 iff x_0's hash is different in both executions, i.e. $H_{a_0^0 \oplus A_{1,3}^0}(x_0) \neq H_{\overline{a_0^0 \oplus A_{1,3}^0}}(\overline{x_0})$. Hence,

$$\Pr[P_0 \text{ is detected} \mid X_{P_0-diff}] = \Pr[H_{a_0^0 \oplus A_{1,3}^0}(x_0) \neq H_{\overline{a_0^0 \oplus A_{1,3}^0}}(\overline{x_0})] = p^*$$

In hybrid #4, the simulator sends **cheat** to $\mathcal{F}_{dual-exec}$ and receives a response r. Denote $X_{\text{DE-detected}}$ the event of $r =$ **cheat-detected**. In the case of $X_{\text{DE-detected}}$, the simulator performs rejection sampling of $k_0 \xleftarrow{\$} \{0,1\}^{n_K}$ until $H_{k_0}(x_0) \neq H_{k_0 \oplus a_0^0 \oplus \overline{a_0^0}}(\overline{x_0})$.

$$\Pr[sampling - different - hashes] = p^*$$

Hence, the expected sampling amount is $\frac{1}{1-p^*}$, which is $poly(...)$.

Then, the simulator sets $A_{1,4}^0 = \overline{A_{1,4}^0} = k_0 \oplus a_0^0$ which holds that:

$$H_{a_0^0 \oplus A_{1,4}^0}(x_0) = H_{a_0^0 \oplus (k_0 \oplus a_0^0)}(x_0) = H_{k_0}(x_0)$$

using the property of the chosen value of k_0

$$\neq H_{k_0 \oplus a_0^0 \oplus \overline{a_0^0}}(\overline{x_0}) = H_{\overline{A_{1,4}^0} \oplus \overline{a_0^0}}(\overline{x_0})$$

In the case of $\neg X_{\text{DE-detected}}$ the simulator samples l keys where $k_0^i \xleftarrow{\$} \{0,1\}^{n_K}$, and if exists a key k_0^i s.t. $H_{k_0^i}(x_0) = H_{k_0^i \oplus a_0^0 \oplus \overline{a_0^0}}(\overline{x_0})$. Denote $X_{\text{sampling-fail}}$ the event

$$\forall i \in [0,l] : H_{k_0^i}(x_0) \neq H_{k_0^i \oplus a_0^0 \oplus \overline{a_0^0}}(\overline{x_0})$$

Hence,

$$\Pr[X_{\text{sampling-fail}}] = (p^*)^l$$

In case of $X_{\text{sampling-fail}}$, the simulator sends **blatant-cheat** to $\mathcal{F}_{dual-exec}$ and sets the honest party's randomness $A_1^0 = \overline{A_1^0} = k_0^i \oplus a_0^0$ for a random $i \in [0,l]$ (the hashes of x_0 and $\overline{x_0}$ are different for the same argument as in case of the event $X_{\text{DE-detected}}$). Otherwise, let $i_{\neq}, i_{=} \in [0,l]$ s.t.

$$H_{k_0^{i_=}}(x_0) = H_{k_0^{i_=} \oplus a_0^0 \oplus \overline{a_0^0}}(\overline{x_0})$$

and

$$H_{k_0^{i\neq}}(x_0) \neq H_{k_0^{i\neq} \oplus a_0^0 \oplus \overline{a_0^0}}(\overline{x_0})$$

W.p. l^* the simulator sets the honest party's randomness $A_1^0 = \overline{A_1^0} = k_0^{i=} \oplus a_0^0$ and continues the simulation, denote this event by $X_{\text{trail-undetected}}$ (i.e. $\Pr[X_{\text{trail-undetected}}] = l^*$).

W.p. $1 - l^*$ the simulator sends **blatant-cheat** to $\mathcal{F}_{dual-exec}$ and sets the honest party's randomness $A_1^0 = \overline{A_1^0} = k_0^{i\neq} \oplus a_0^0$. Hence,

$$\begin{aligned}
\Pr[P_0 \text{ is undetected}] &= \Pr[\neg X_{\text{DE-detected}} \wedge \neg X_{\text{sampling-fail}} \wedge X_{\text{trail-undetected}}] \\
&= \underbrace{\Pr[\neg X_{\text{DE-detected}}]}_{\varepsilon} \cdot \Pr[\neg X_{\text{sampling-fail}} \wedge X_{\text{trail-undetected}} \mid \neg X_{\text{DE-detected}}] \\
&= \varepsilon \cdot \underbrace{\Pr[\neg X_{\text{sampling-fail}} \mid \neg X_{\text{DE-detected}}]}_{1-(p^*)^l} \\
&\quad \cdot \underbrace{\Pr[X_{\text{trail-undetected}} \mid \neg X_{\text{DE-detected}}, \neg X_{\text{sampling-fail}}]}_{l^*} \\
&= \varepsilon \cdot (1 - (p^*)^l) \cdot l^*
\end{aligned}$$

We need to choose l, l^* s.t. $\Pr[P_0 \text{ is detected}] = 1 - p^*$:

$$\begin{aligned}
\Pr[P_0 \text{ is detected}] &= 1 - p^* \\
\varepsilon \cdot (1 - (p^*)^l) \cdot l^* &= 1 - p^* \\
l^* &= \frac{1 - p^*}{\varepsilon \cdot (1 - (p^*)^l)}
\end{aligned}$$

Also, $0 \leq l^* \leq 1$ because l^* is a probability. Therefore:

$$\begin{aligned}
\frac{1 - p^*}{\varepsilon \cdot (1 - (p^*)^l)} &\leq 1 \\
\frac{1 - p^*}{\varepsilon} &\leq (1 - (p^*)^l) \\
(p^*)^l &\leq \underbrace{1 - \frac{1 - p^*}{\varepsilon}}_{\leq 1 (\text{because } \varepsilon \leq p^*)} \\
l \cdot \log_2(p^*) &\leq \log_2\left(1 - \frac{1 - p^*}{\varepsilon}\right) \\
l &\geq \frac{1}{\log_2(p^*)} \cdot \log_2\left(1 - \frac{1 - p^*}{\varepsilon}\right)
\end{aligned}$$

Hence, we'll set $l = \left\lceil \frac{1}{\log_2(p^*)} \cdot \log_2\left(1 - \frac{1-p^*}{\varepsilon}\right) \right\rceil$.

4. **Hybrid #4 – Hybrid #5** — Denote $VIEW$ the set of possible views to be generated in hybrid #4. We'll show that for any $view \in VIEW$:

$$\Pr[VIEW_4 = view] = \Pr[VIEW_5 = view]$$

For hybrid $\#i$ ($i \in \{4,5\}$), denote $VIEW_i^{pref} = (C, \overline{C})$ and $VIEW_i^{suf} = (OUT_i, VIEW_i^{arb})$.

Both hybrids execute the same code for generating $VIEW_4^{pref}$ and $VIEW_5^{pref}$, hence both views are identically distributed:

$$\Pr[VIEW_4^{pref} = view_4^{pref}] = \Pr[VIEW_5^{pref} = view^{pref}]$$

The only thing changed from hybrid #4 are the values of $Y, \overline{Y}$ (where Y is a RV of the output $f(x_0, x_1)$, and is part of $\mathcal{F}_{\varepsilon-\text{one-shot}}$'s output OS, as defined at the beginning of the proof).

In hybrid #4 Y holds the value of $f(x_0, x_1)$ if P_0 sends **reveal**, and $\perp$ otherwise.

In hybrid #5 we change the operation of $\mathcal{F}_{\varepsilon-\text{one-shot}}$ to set Y to be the output of $\mathcal{F}_{dual-exec}$ if P_0 sends **reveal**, and $\perp$ otherwise.

Denote X_{cheat} the event that cheating attempt occurred before item 2 in the ideal world simulator. Considering the event X_{cheat}, the simulator receives the honest party's inputs and in this case a perfect honest simulation is generated, using x_1 and the randomness a_1^0, a_1^1. Hence,

$$\begin{aligned}&\Pr[Y_5 = y, \overline{Y_5} = \overline{y} \mid VIEW_5^{pref} = view^{pref}, X_{cheat}] \\ &\quad = \Pr[Y_4 = y, \overline{Y_4} = \overline{y} \mid VIEW_4^{pref} = view^{pref}, X_{cheat}]\end{aligned}$$

Considering the event $\neg X_{cheat}$, P_0 sent *the same input* to both $\mathcal{F}_{\varepsilon-\text{one-shot}}, \overline{\mathcal{F}_{\varepsilon-\text{one-shot}}}$ (i.e. $x_0 = \overline{x_0}$), hence in hybrid #4 both outputs $Y_4, \overline{Y_4}$ have the same value of $f(x_0, x_1)$, while in hybrid #5 the simulator sets both $Y_5, \overline{Y_5}$ to the the output of $\mathcal{F}_{dual-exec}$ which is, by definition, $f(x_0, x_1)$. Hence,

$$\begin{aligned}&\Pr[Y_5 = y, \overline{Y_5} = \overline{y} \mid VIEW_5^{pref} = view^{pref}, \neg X_{cheat}] \\ &\quad = \Pr[Y_4 = y, \overline{Y_4} = \overline{y} \mid VIEW_4^{pref} = view^{pref}, \neg X_{cheat}]\end{aligned}$$

5. **Hybrid #5 – Hybrid #6** — the only places where the value of x_1 is used is in $\mathcal{F}_{\varepsilon-\text{one-shot}}$ and in arbitration.

For $i \in \{5,6\}$, let $VIEW_i^{before-arb} = (C_i, \overline{C_i}, OS_i, \overline{OS_i})$ denote the environment's view before arbitration.

In hybrid #5 $\mathcal{F}_{\varepsilon-\text{one-shot}}$ use x_1 only during arbitration and therefore the generated view before arbitration is independent of this value. Because both hybrids send

cheat to $\mathcal{F}_{dual-exec}$ when P_0 cheats, P_1's input is always accessible to the simulator during arbitration. Hence,

$$\begin{aligned}&\Pr[VIEW_5^{before-arb} = view^{before-arb}]\\&\quad = \Pr[VIEW_6^{before-arb} = view^{before-arb}]\end{aligned}$$

During arbitration, both hybrids execute the same code using the honest party's input x_1 and randomness a_1^0, a_1^1, hence both arbitration views are identical, given that the view before arbitration is the same in both hybrids:

$$\begin{aligned}&\Pr[VIEW_5^{arb} = view^{arb} \mid VIEW_5^{before-arb} = view^{before-arb}]\\&\quad = \Pr[VIEW_6^{arb} = view^{arb} \mid VIEW_6^{before-arb} = view^{before-arb}]\end{aligned}$$

6. **Hybrid #6 – Hybrid #7** — Denote $VIEW$ the set of possible views to be generated in hybrid #6. We'll show that for any $view \in VIEW$:

$$\Pr[VIEW_6 = view] = \Pr[VIEW_7 = view]$$

For hybrid $\#i$ ($i \in \{6, 7\}$), denote $VIEW_i^{pref} = (C_i, \overline{C_i}, OS_i, \overline{OS_i}, VIEW_i^{arb})$ Both hybrids execute the same code for generating $VIEW_6^{pref}$ and $VIEW_7^{pref}$, hence both views are identically distributed:

$$\Pr[VIEW_6^{pref} = view_6^{pref}] = \Pr[VIEW_7^{pref} = view^{pref}]$$

Next, we need to prove that

$$\begin{aligned}&\Pr[OUT_6 = out \mid VIEW_6^{pref} = view_6^{pref}]\\&\quad = \Pr[OUT_7 = out \mid VIEW_7^{pref} = view^{pref}]\end{aligned}$$

Using claim 8.2.3 (i.e. *Accountability* property), $OUT_6 = id_0 \rightarrow OUT_7 = id_0$.

Using claim 8.2.4 (i.e. *Defamation freeness* property), $OUT_7 = id_0 \rightarrow OUT_6 = id_0$.

Therefore, $OUT_7 = id_0 \iff OUT_6 = id_0$. Hence,

$$\begin{aligned}&\Pr[OUT_6 = out \mid VIEW_6^{pref} = view_6^{pref}]\\&\quad = \Pr[OUT_7 = out \mid VIEW_7^{pref} = view^{pref}]\end{aligned}$$

□

8.2. Accountability and Defamation-Freeness

Note that the protocol is symmetric, therefore w.l.o.g. we assume P_1 is the corrupt party.

Lemma 8.2.1. *If P_0 is honest, then $out_0 = C_f(x_0^e, x_1^e)$, where x_i^e is the input used by P_i as evaluator and f is the evaluated function (an equivalent claim: If $out_0 \neq C_f(x_0^e, x_1^e)$ then P_0 is corrupt)*

Proof. For all $i \in \{0,1\}$, denote f_i be the function evaluated at $\mathcal{F}_{OS}^i$ (in which P_{1-i} acted as evaluator).

If P_0 is honest, he uses consistent input values in both executions (i.e. $x_0^e = x_0^g$) and $f_0 = f$ because he acts as honest garbler in $\mathcal{F}_{OS}^i$ and as such he sends a valid circuit for evaluating function f. Hence: $out_0 = C_{f_0}(x_0^g, x_1^e) \underset{f_0=f}{=} C_f(x_0^g, x_1^e) \underset{x_0^g=x_0^e}{=} C_f(x_0^e, x_1^e)$ □

Lemma 8.2.2. *Let $V(C_f)$ a set of wire-values corresponding to the wires in circuit C_f, where out' is the output value. If $out' \neq C_f(x_0^e, x_1^e)$ then one of the following is true:*

- *There's an input-wire w of P_i ($i \in \{0,1\}$) s.t. $v_w \neq x_i^e[w]$ where $v_w \in V(C_f)$*
- *There's a gate $g = (w_{in_0}, w_{in_1}, w_{out}) \in C_f$ s.t. $v_{w_{out}} \neq NAND(v_{w_{in_0}}, v_{w_{in_1}})$, for $v_{w_{out}}, v_{w_{in_0}}, v_{w_{in_1}} \in V(C_f)$*

Proof. Denote $V^*(C_f)$ the set of wire-values of wires in C_f corresponding to an evaluation of C_f using inputs x_0^e, x_1^e. It can be observed that the output-wires' values in $V^*(C_f)$ correspond to $C_f(x_0^e, x_1^e)$.

Let w be a wire in C_f s.t. $v_w \neq v_w^*$ for $v_w \in V(C_f), v_w^* \in V^*(C_f)$ (there's at least one such wire because $out' \neq C_f(x_0^e, x_1^e)$).
Denote w_{min} the first occurrence of such wire, using topological order of the wires. Hence one of the following is correct:

- w_{min} is an input-wire – by $V^*(C_f)$'s definition, the used inputs in $V^*(C_f)$ are x_0^e, x_1^e and therefore $v_{w_{min}}^* = x_i^e[w_{min}]$ where w_{min} is input-wire of party P_i.
  Hence, $v_{w_{min}} \neq v_{w_{min}}^* = x_i^e[w_{min}]$.
- w_{min} is an output-wire of gate g located at depth k – denote w_0, w_1 the input-wires of gate g. $v_{w_i} = v_{w_i}^*$ for all $i \in \{0,1\}$ by w_{min}'s definition.
  By $V^*(C_f)$'s definition, the evaluated circuit is C_f and therefore $v_{w_{min}}^* = NAND(v_{w_0}^*, v_{w_1}^*) = NAND(v_{w_0}, v_{w_1})$.
  Hence, $v_{w_{min}} \neq NAND(v_{w_0}, v_{w_1})$

□

Claim 8.2.3. *Our dual-execution protocol maintains the "Accountability" definition (c.f. item 2 in definition A.3.1).*

Proof. We'll prove that for any abort triggered by the honest party, the arbitrator accuses the adversary. We'll do it by enumerating the cases where the honest party aborts and show that in each case he is able to generate a proof-of-malfeasance that's used by the arbitrator in order to accuse the adversary.

We'll consider the case where the protocol's execution halted because P_0 claimed P_1 is corrupt. By observing the dual-execution's protocol (c.f. protocol $\Pi_{P_i}^{DE}$ [3.1]), the only conditions where P_0 aborts are as follows:

1. **One-shot functionality abort (c.f. line 9 in protocol $\Pi^{DE}_{P_i}$ [3.1])** – an abort occurs iff a **cheat-detected** message is received from a one-shot functionality. Observing the functionality's interface (c.f. section 3.1), this message is sent iff a cheating attempt was detected and contains also a proof-of-malfeasance φ that proves the adversary's corruption.

2. **Inconsistent outputs (c.f. line 21 in protocol $\Pi^{DE}_{P_i}$ [3.1])** – P_0 reaches this line iff different output was evaluated by the two executions and the check at protocol CheckIfConsistent [3.1] returned a complaint. Moreover, by the time P_0 reached line 21 he already holds the plaintext evaluated wire-values of C_f (denoted $V(C_f)$) and the arbitrator holds both parties' inputs as they used as evaluators in the corresponding $\mathcal{F}^0_{OS}, \mathcal{F}^1_{OS}$.

   Denote out' the plaintext output of $V(C_f)$:

   a) If $out' = out_0$, therefore $out' \neq out_1$ (because $out_0 \neq out_1$) – i.e. exists output-wire w s.t. $out'[w] \neq out_1[w]$.

   b) If $out' \neq out_0$, by lemma 8.2.1 $out' \neq C_f(x^e_0, x^e_1)$ and by lemma 8.2.2 one of the following is true:

      - There's an input-wire w of P_i ($i \in \{0,1\}$) s.t. $v_w \neq x^e_i[w]$ where $v_w \in V(C_f)$ and x^e_i is the input used by P_i as evaluator
      - There's a gate $g = (w_{in_0}, w_{in_1}, w_{out}) \in C_f$ s.t. $v_{w_{out}} \neq NAND(v_{w_{in_0}}, v_{w_{in_1}})$, for $v_{w_{out}}, v_{w_{in_0}}, v_{w_{in_1}} \in V(C_f)$

   We'll inspect protocol CheckIfConsistent [3.1], considering the possible cases as described above, and prove that P_0 returns a complaint in each of these cases:

   a) There's an output-wire w s.t. $out_1[w] \neq out'[w]$ – P_0 will ask $\mathcal{F}^1_{\varepsilon-\text{one-shot}}$ to broadcast the evaluated output in wire w (line 14), then he aborts at line 15 in protocol CheckIfConsistent [3.1] and sends the corresponding plaintext wire-values of w to the arbitrator.

      The arbitrator executes protocol HandleInvalidPlainWire [4.1] for the corresponding complaint. The check at line 2 fails because the provided wire value $out'[w]$ is properly-signed. The check at line 5 fails because P_0 sent $\mathcal{F}^1_{\varepsilon-\text{one-shot}}$ a message $(\textbf{prove-output}, w)$ (that's followed by $\mathcal{F}^1_{\varepsilon-\text{one-shot}}$ broadcasting the message $(\textbf{prove-output}, w, out_1[w])$). The check at line 11 passes because $out_1[w] \neq out'[w]$ and therefore the adversary is accused.

   b) There's an input-wire w of P_i ($i \in \{0,1\}$) s.t. $v_w \neq x^e_i[w]$ where $v_w \in V(C_f)$ and x^e_i is the input used by P_i as evaluator – in this case P_0 aborts in line 7 in protocol CheckIfConsistent [3.1] and submits the (signed) invalid plaintext value v_w to the arbitrator, which already holds $x^e_i[w]$.

      The arbitrator executes protocol HandleInvalidPlainWire [4.1] for the corresponding complaint. The check at line 2 fails because the provided wire value is properly-signed and he holds the value of input-wire w. The check at line 11 passes because $x^e_i[w] \neq v_w$ and therefore the adversary is accused.

c) There's a gate $g = (w_{in_0}, w_{in_1}, w_{out}) \in C_f$ s.t. $v_{w_{out}} \neq NAND(v_{w_{in_0}}, v_{w_{in_1}})$, for $v_{w_{out}}, v_{w_{in_0}}, v_{w_{in_1}} \in V(C_f)$ – in this case P_0 will abort at line 22 in protocol CheckIfConsistent [3.1]. P_0 submits the (signed) values of the input and output wires corresponding to gate g to the arbitrator.

The arbitrator executes protocol HandleInvalidPlainGate [4.2] for the corresponding complaint. He reaches line 8 because all wire values are valid (if some of them were not, the honest party would complain earlier in protocol CheckIfConsistent [3.1] at line 7 or line 15). Finally, the check at line 8 passes because the given wire values don't represent a valid NAND evaluation of gate g and therefore he accuses the adversary.

□

Claim 8.2.4. *Our dual-execution protocol maintains the "Defamation-Free" definition (c.f. item 3 in definition A.3.1).*

Proof. We'll prove that the honest party (i.e. P_0) can't be falsely accused by the arbitrator by showing that all complaints (considering the sender and the complaint type) sent to the arbitrator can't cause a false accusation of the honest party.
Due to claim 8.2.3, the arbitrator will always accuse the adversary for complaints presented by the honest party and therefore defamation-freeness is guaranteed in this case. Hence, if the corrupt party wants to frame the honest party it would falsely claim one of the following:

1. **Inconsistent plaintext input/output wire (c.f. protocol HandleInvalidPlainWire [4.1])** – Let $V(C_f)$ denote the evaluated wires of C_f and out' the corresponding output value, as sent by P_0 after being blamed by the adversary. By lemma 1, $out' = out_0$ and therefore he can't be accused for having inconsistent output-wire's values in out', out_0. In addition, P_0 use x_0^e, x_1^e as inputs in $V(C_f)$ and therefore he can't be accused for using inconsistent input-wire's values in $V(C_f)$ and x_i^e (for all $i \in \{0,1\}$).

   In order to frame the honest party, the adversary must send the arbitrator a message containing a value of wire w (denoted v_w) signed by the honest party, s.t. $v_w \neq v_w^*$, where v_w^* is the value of w which was used by the honest party in $\mathcal{F}_{OS}^i$. The honest party follows the protocol, hence the value of w he used in $\mathcal{F}_{OS}^i$ is the same as the one used to evaluate the plaintext circuit C_f. Therefore, the adversary frames the honest party with negligible probability.

2. **Inconsistent plaintext gate (c.f. protocol HandleInvalidPlainGate [4.2])**– Let $V(C_f)$ denote the evaluated wires of C_f and out' the corresponding output value, as sent by P_0 after being blamed by the adversary.

   P_0 acts honestly, therefore he evaluates all gates in C_f correctly and hence for all gates $g = (w_{in_0}, w_{in_1}, w_{out}) \in C_f$ the following holds: $v_{w_{out}} \neq NAND(v_{w_{in_0}}, v_{w_{in_1}})$, for $v_{w_{out}}, v_{w_{in_0}}, v_{w_{in_1}} \in V(C_f)$.
   Hence, P_0 can't be accused having an invalid gate evaluation in $V(C_f)$.

In order to frame the honest party, the adversary must send the arbitrator a message containing a gate $g \in C_f$ and the values of the corresponding gate's input and output wires, as evaluated by the honest party, s.t. the wire values don't resemble a valid NAND-gate evaluation.

The honest party follows the protocol, hence he will evaluate the plaintext circuit C_f correctly. All gates in C_f are NAND gates, therefore all wire-values that are passed to the adversary represent valid evaluation of all NAND gates in C_f. Therefore, the adversary frames the honest party with negligible probability.

□

9. PVOT (Online) Protocol: Overview

In this section we'll describe our publicly-verifiable OT protocol, which guarantees security against covert adversaries and selective-failure attacks.
Our protocol is based on a selective-failure OT protocol defined in Doerner, Kondi, Lee, and abhi shelat [9]. Their protocol is secure against a malicious adversary, but requires access to an ideal functionality $\mathcal{F}_{ZK}^{R_{DL}}$ modeling a zero-knowledge proof-of-knowledge-of-discrete-logarithm protocol.

In order to make it publicly-verifiable and improve verification efficiency, we added signatures to the messages exchanged between the parties and replaced the usage of $\mathcal{F}_{ZK}^{R_{DL}}$ by a simpler mechanism.

The original protocol involves calculating two pads ρ_0, ρ_1 — based on randomness generated by the parties and the receiver's input ω — and includes two parts: (1) the parties perform a "two-way handshake", where each party checks if the other party calculated ρ_ω correctly, and (2) the sender encrypts the inputs using the generated pads — the receiver is able to calculate only a single pad, and therefore he can't decrypt both messages.

In the original protocol, $\mathcal{F}_{ZK}^{R_{DL}}$ is used for checking that the randomness used by the sender corresponds to a message sent to the receiver, without compromising its value. In our protocol, instead, we require the sender to reveal the randomness used in case of a complaint. (This doesn't violate security, since the sender's inputs haven't been used yet at this point in the protocol.) For a full, formal specification of the protocol, see protocols Π_{sender}^{PVOT} [5.1] and $\Pi_{receiver}^{PVOT}$ [5.3].

9.1. Security Analysis

Our main theorem is as follows:

Theorem 9.1.1. *The protocol Π^{OT} realizes $\mathcal{F}_{PVOT}$ functionality with malicious security.*

Proof Sketch of theorem 9.1.1

The full proof of theorem 9.1.1 appears in chapter 11. Below we give a high-level overview of our security proof.

We describe a simulator $\mathcal{S}$ that runs $\mathcal{A}$ internally and interacts with the trusted party that computes f.

We consider the two possible corruption scenarios separately (sender and receiver).

Sender is corrupt: $\mathcal{S}$ simulates an honest receiver (using a random value for A), until he needs to respond to the challenge ξ – until that point, $\mathcal{S}$ receives the pads h_0, h_1

and challenge ξ. Note that an honest sender checks, upon receiving a response, that it equals to $H(H(b \cdot A))$. Considering that $\mathcal{S}$ doesn't hold the receiver's input ω, he attempts to extract the value b and use it to create a valid response.

$\mathcal{S}$ extracts the value b by observing the queries made by the sender to the random oracle — an honest sender should compute the pads $\rho^0 = H(b \cdot A)$ and $\rho^1 = H(b \cdot (A - B))$, where $B = b \cdot G$. If a query q^* exists s.t. $q^* = b \cdot A$ or $q^* = b \cdot (A - B)$, then we will be able to extract the value of b as follows — $\frac{q^*}{A} \cdot G = b \cdot G = B$ or $\frac{q^*}{A-B} \cdot G = b \cdot G = B$:

Case 1: **(no such query q^* exists)**, the check at line 11 would fail in a real execution and an honest receiver will abort, therefore $\mathcal{S}$ simulates an abort in this case.

Case 2: **(such query q^* exists)**, $\mathcal{S}$ computes b and then computes the sender's expected view (i.e. both pads and challenge ξ as should be calculated by an *honest* sender) and continues and follows.

Then, $\mathcal{S}$ compares the values of received pads and their expected values (as calculated above, using b): if none of the pads were calculated correctly the check at line 11 will pass for any $\omega \in \{0, 1\}$ and an honest receiver will abort w.p. 1, hence $\mathcal{S}$ simulates an abort in that case. If a single pad was computed correctly then the sender's cheating attempt relies on the value of ω in order to be undetected — $\mathcal{S}$ performs a coin-toss in order to decide whether to detect it or not:

Case 1: If the cheating attempt is detected, the honest receiver will check the correctness of the *invalid* pad in line 11, and as a result an honest receiver will abort. Therefore $\mathcal{S}$ simulates an abort in this case.

Case 2: If the cheating attempt is undetected, the honest receiver will check the correctness of the *valid* pad in line 11 and hence no abort will be triggered.

Next, $\mathcal{S}$ checks if the sender's challenge is valid (i.e. the receiver's check in line 14) — if it is invalid, an abort will be triggered in a real execution by the receiver and therefore $\mathcal{S}$ simulates an abort. Otherwise, $\mathcal{S}$ continues as follows. Then, $\mathcal{S}$ perfectly-simulates the rest of the protocol — he waits for the garbler to send the inputs $\tilde{\alpha}^0, \tilde{\alpha}^1$, and sends α_0, α_1 to $\mathcal{F}_{PVOT}$ (where α_i is obtained by decrypting $\tilde{\alpha}^i$ using the corresponding pad).

Receiver is corrupt: Note that an honest sender uses his inputs (i.e. messages α_0, α_1) only at the final step of the protocol, therefore $\mathcal{S}$ can perfectly simulate the preceding steps.

For simulate the final step, we need to extract ω – otherwise, we might generate a view where a receiver asked for message ω but received message $\bar{\omega}$ which is distinguishable from a real execution. In order to extract the receiver's input ω, it suffices to observe the queries to the random oracle. The receiver can't query both ρ^0 and ρ^1 (look at the full proof for an explanation why the receiver can't query both of them).

Case 1: **(neither ρ^0 nor ρ^1 were queried by the receiver)** then the challenge's response is invalid with overwhelming probability and therefore the check at line 17 will pass and a following abort will be triggered. Therefore, $\mathcal{S}$ simulates an abort in that case.

Case 2: **(ρ^0 or ρ^1 were queried by the receiver)** $\mathcal{S}$ sets ω to be the value $i \in \{0,1\}$ s.t. ρ^i was queried and continue as follows.

Finally, $\mathcal{S}$ asks $\mathcal{F}_{PVOT}$ for message ω and receives α_ω as a response. Then he sends the corrupt receiver m_0, m_1 s.t.

$$m_\omega \leftarrow E_{\rho^\omega}(\alpha_\omega)$$
$$m_{\bar{\omega}} \xleftarrow{\$} E_{\rho^{\bar{\omega}}}(0^n)$$

This is indistinguishable from the real view because the encryption scheme is semantically secure, and adversary has no information at all about $\rho^{\bar{\omega}}$.

10. Full Description of the PVOT (online) Protocol

In this chapter we give a full, formal description of our protocol to realize $\mathcal{F}_{PVOT}$.

10.1. Notations

This protocol is parameterized by the Elliptic curve $(\mathbb{G}, G, q)$ (using a group $\mathbb{G}$, generator G and order q) and symmetric security parameter $k = |q|$. It makes use of a hash function H and a symmetric encryption scheme (G_{enc}, E, D). It takes as input a choice bit $\omega \in \{0, 1\}$ from the receiver, and two messages $\alpha_0, \alpha_1 \in \mathbb{Z}_q$ from the sender.

10.2. PVOT (online) protocol main routines

The sender's main routine is described in protocol Π^{PVOT}_{sender} [5.1], where protocol $\Pi^{PVOT}_{\text{sender-rebuttal}}$ [5.2] is invoked if the other party broadcasts a complaint.

The receiver's main routine is described in protocol $\Pi^{PVOT}_{receiver}$ [5.3], where protocol $\Pi^{PVOT}_{\text{receiver-rebuttal}}$ [5.4] is invoked if the other party broadcasts a complaint.

The arbitrator's main routine is described in protocol Π^{PVOT}_{arb} [5.5]. It uses protocol HandleOTComplaint [5.6] for handling the parties' complaints (if such are published).

10.3. PVOT (online) protocol auxiliary routines

10.3.1. Arbitration subroutines

protocol HandleProveOutput [6.1] provides a publicly verifiable proof of the receiver's output in the OT protocol.

protocol HandleInvalidChallenge [6.2] handles a receiver's complaint where the challenge ξ sent by the sender is invalid — the arbitrator accuse the sender iff the challenge ξ and h_0, h_1 are signed by the sender, and ξ is invalid.

protocol HandleInconsistentPadding [6.3] handles a receiver's complaint where the challenge ξ sent by the sender is invalid — the arbitrator waits for the sender's rebuttal. If the sender rebuts, protocol HandleInconsistentPaddingRebut [6.4] is called, and otherwise the sender is accused.

protocol HandleInvalidResponseDecommitment [6.5] handles a sender's complaint where the decommitment corresponding to the receiver's response to the challenge ξ is invalid.

protocol HandleInvalidChallengeResponse [6.6] handles a sender's complaint where the receiver's response to challenge ξ is invalid — the arbitrator waits for the receiver's

Protocol Π^{PVOT}_{sender} **[5.1]** Publicly-verifiable OT pseudo-code

1: **function** $\Pi^{PVOT}_{sender}(\alpha^0, \alpha^1)$
2: // Public Key
3: Sample $b \xleftarrow{R} \mathbb{Z}_q$, computes $B = b \cdot G$ and sends B to the receiver
4: // Pad Transfer
5: Wait to receive A from receiver
6: Compute the values of $\rho^0 \leftarrow H(b \cdot A), \rho^1 \leftarrow H(b \cdot (A - B))$
7: // Verification
8: Compute the challenge's value $\xi \leftarrow H(H(\rho^0)) \oplus H(H(\rho^1))$ and send it to the receiver
9: Wait to receive c from the receiver
10: Send $H(\rho^0), H(\rho^1)$ to the receiver
11: Wait to receive (π_{ψ^*}, ψ^*) from the receiver
12: **if** VerifyDecommitment$(c, \pi_{\psi^*}, \psi^*) = \perp$ **then**
13: // $(InvalidOpening, c, \pi_{\psi^*}, \psi^*)$
14: **Broadcast** complaint $\Big(\textbf{invalid-opening}, (c, (\psi^*, \pi_{\psi^*}))\Big)$
15: **return** $\perp$
16: **end if**
17: **if** $\psi^* \neq H(H(\rho^0))$ **then**
18: **Broadcast** complaint $\Big(\textbf{invalid-challenge-response}, (c, (\psi^*, \pi_{\psi^*}), H(\rho^0))\Big)$
19: **return** $\perp$
20: **end if**
21: // Message transfer
22: Calculate (for all $i \in \{0, 1\}$) $\tilde{\alpha}^i \leftarrow E_{\rho^i}(\alpha^i)$
23: Send $\tilde{\alpha}^0, \tilde{\alpha}^1$ to receiver
24: **return**
25: **end function**

Protocol $\Pi^{PVOT}_{\textbf{sender-rebuttal}}$ **[5.2]** Sender's rebuttal routine

1: **function** $\Pi^{PVOT}_{\text{SENDER-REBUTTAL}}$
2: **if invalid-padding** complaint was broadcasted by the receiver **then**
3: Denote b the value generated for calculating B at line 3 in protocol Π^{PVOT}_{sender} [5.1]
4: Denote A the value received at line 5 in protocol Π^{PVOT}_{sender} [5.1]
5: Broadcast message $(\textbf{rebut}, \textbf{invalid-padding}, A, b)$
6: **end if**
7: **end function**

Protocol $\Pi^{PVOT}_{receiver}$ **[5.3]** Publicly-verifiable OT pseudo-code

1: **function** $\Pi^{PVOT}_{receiver}(\omega)$
2: // Public Key
3: Receive B from the sender
4: // Pad Transfer
5: Sample $a \xleftarrow{R} \mathbb{Z}_q$ and compute $A = a \cdot G + \omega \cdot B$ and $\rho^\omega \leftarrow H(a \cdot B)$
6: Send A to the receiver
7: // Verification
8: Wait to receive the challenge's value ξ from the sender
9: Calculate $\psi \leftarrow H(H(\rho^\omega)) \oplus (\omega \cdot \xi)$ and send $Comm(\psi)$ to the sender
10: Wait to receive h_0, h_1 from the sender // an honest sender will send $H(\rho^0), H(\rho^1)$
11: **if** $H(\rho^\omega) \neq h_\omega$ **then**
12: **Broadcast** complaint $\Big(\textbf{invalid-padding}, (B, h_0, h_1)\Big)$
13: **return** $\perp$
14: **else if** $\xi \neq H(h_0) \oplus H(h_1)$ **then**
15: **Broadcast** complaint $\Big(\textbf{invalid-challenge}, (\xi, h_0, h_1)\Big)$
16: **return** $\perp$
17: **end if**
18: Send (ψ, π_ψ) to the sender
19: // Message transfer
20: Wait to receive $\tilde{\alpha}^0, \tilde{\alpha}^1$ from the sender
21: Calculate $\alpha^\omega \leftarrow D_{\rho^\omega}(\tilde{\alpha}^\omega)$
22: **return** α^ω
23: **end function**

24: **function** ProveOutput()
25: Denote ρ^ω the value calculated at line 5 in protocol $\Pi^{PVOT}_{receiver}$ [5.3].
26: Denote (h_0, h_1) and $(\tilde{\alpha}^0, \tilde{\alpha}^1)$ the messages received at line 10 and line 20 in protocol $\Pi^{PVOT}_{receiver}$ [5.3], respectively.
27: Send $(\textbf{prove-output}, (h_0, h_1), \rho^\omega, (\tilde{\alpha}^0, \tilde{\alpha}^1))$ to the arbitrator
28: **end function**

Protocol $\Pi^{PVOT}_{\textbf{receiver-rebuttal}}$ **[5.4]** Receiver's rebuttal routine

1: **function** $\Pi^{PVOT}_{\text{RECEIVER-REBUTTAL}}$
2: **if invalid-challenge-response** complaint was broadcasted by the sender **then**
3: Denote h_0 the value received at line 10 in protocol $\Pi^{PVOT}_{receiver}$ [5.3]
4: Broadcast message $(\textbf{rebut}, \textbf{invalid-challenge-response}, h_0)$
5: **end if**
6: **end function**

Protocol $\Pi^{PVOT}_{\mathbf{arb}}$ [5.5] The arbitration routine of the OT (online) protocol

function Π^{PVOT}_{ARB}
  Wait for a complaint $\Big(reason, (message)\Big)$.
  if received a complaint **then**
    Denote P_i the sender of the complaint
    $res \leftarrow$ HandleOTComplaint$(P_i, reason, message)$
    if $res \neq \perp$ **then**
      return res
    end if
  end if
end function

rebuttal. If the sender rebuts, protocol HandleInvalidChallengeResponseRebut [6.7] is called, and otherwise the sender is accused.

Protocol HandleOTComplaint [5.6] Arbitration protocol for OT

1: **function** HANDLEOTCOMPLAINT($id, reason, message$)
2: **if** $reason =$ **prove-output** and $id = id_{receiver}$ **then**
3: parse message as $\Big((h_0, h_1), \rho^\omega, (\tilde{\alpha}^0, \tilde{\alpha}^1)\Big)$
4: $res \leftarrow$ HANDLEPROVEOUTPUT$((h_0, h_1), \rho^\omega, (\tilde{\alpha}^0, \tilde{\alpha}^1))$ // c.f. protocol HandleProveOutput [6.1]
5: **else if** $reason =$ **invalid-challenge** and $id = id_{receiver}$ **then**
6: parse message as (ξ, h_0, h_1)
7: $res \leftarrow$ HANDLEINVALIDCHALLENGE(ξ, h_0, h_1) // c.f. protocol HandleInvalidChallenge [6.2]
8: **else if** $reason =$ **invalid-padding** and $id = id_{receiver}$ **then**
9: parse message as (B, h_0^*, h_1^*)
10: $res \leftarrow$ HANDLEINVALIDPADDING(B, h_0^*, h_1^*) // c.f. protocol HandleInconsistentPadding [6.3]
11: **if** $res = \Big(\textbf{cheat-detected}, id_{sender}\Big)$ **then**
12: **if** a rebut $message_{rebut}$ was received from the sender **then**
13: Parse $message_{rebut}$ as a tuple (A, b)
14: $res \leftarrow$ HANDLEINVALIDPADDINGREBUT(B, h_0^*, h_1^*, A, b) // c.f. line 1
15: **end if**
16: **end if**
17: **else if** $reason =$ **invalid-opening** and $id = id_{sender}$ **then**
18: parse message as $\Big(c, (\psi^*, \pi_{\psi^*})\Big)$
19: $res \leftarrow$ HANDLEINVALIDCHALLANGERESPONSEDECOMMITMENT$(c, (\psi^*, \pi_{\psi^*}))$ // c.f. protocol HandleInvalidResponseDecommitment [6.5]
20: **else if** $reason =$ **invalid-challenge-response** and $id = id_{sender}$ **then**
21: parse message as $\Big(c, (\psi^*, \pi_{\psi^*}), h\Big)$
22: $res \leftarrow$ HANDLEINVALIDCHALLANGERESPONSE$(c, (\psi^*, \pi_{\psi^*}), h)$ // c.f. protocol HandleInvalidChallengeResponse [6.6]
23: **if** $res = \Big(\textbf{cheat-detected}, id_{receiver}\Big)$ **then**
24: **if** a rebut message $message_{rebut}$ was received from the receiver **then**
25: Parse $message_{rebut}$ as a h'
26: $res \leftarrow$ HANDLEINVALIDCHALLANGERESPONSEREBUT$(c, (\psi^*, \pi_{\psi^*}), h, h')$ // c.f. line 1
27: **end if**
28: **end if**
29: **end if**
30: **return** message res
31: **end function**

Protocol HandleProveOutput [6.1] Handle complaint of an invalid challenge sent during the verification process in the OT protocol

```
    // This routine can only be used by the receiver!
 1: function HandleProveOutput((h_0, h_1), ρ^ω, (α̃^0, α̃^1))
 2:     if (h_0, h_1) or (α̃^0, α̃^1) aren't signed by the sender then
 3:         return (cheat-detected, id_receiver)
 4:     end if
 5:     if H(ρ^ω) ≠ h_ω then
 6:         return (cheat-detected, id_receiver)
 7:     end if
 8:     return message (prove-output, ω, D_{ρ^ω}(α̃^ω))
 9: end function
```

Protocol HandleInvalidChallenge [6.2] Handle complaint of an invalid challenge sent during the verification process in the OT protocol

```
    // This routine can only be used by the receiver!
 1: function HandleInvalidChallenge(ξ, h_0, h_1)
 2:     if ξ, h_0 or h_1 aren't signed by the sender then
 3:         return (cheat-detected, id_receiver)
 4:     end if
 5:     if ξ ≠ H(h_0) ⊕ H(h_1) then
 6:         return (cheat-detected, id_sender)
 7:     else
 8:         return (cheat-detected, id_receiver)
 9:     end if
10: end function
```

Protocol HandleInconsistentPadding [6.3] Handle complaint of an inconsistent padding calculated by the sender and receiver during the Pad-Transfer process in the OT protocol

```
    // This routine can only be used by the receiver!
 1: function HandleInconsistentPadding(B, h_0^*, h_1^*)
 2:     if B, h_0^* or h_1^* aren't signed by the sender then
 3:         return (cheat-detected, id_receiver)
 4:     end if
 5:     return ⊥
 6: end function
```

Protocol HandleInconsistentPaddingRebut [6.4] Handle the rebut of a complaint of an inconsistent padding calculated by the sender and receiver during the Pad-Transfer process in the OT protocol

function HandleInconsistentPaddingRebut(A, b, B, h_0^*, h_1^*)
  if A isn't signed by the receiver **then**
    return $\left(\textbf{cheat-detected}, id_{sender}\right)$
  end if
  if $b \cdot G \neq B$ **then**
    return $\left(\textbf{cheat-detected}, id_{sender}\right)$
  else if $h_0^* \neq H(H(b \cdot A))$ **then**
    return $\left(\textbf{cheat-detected}, id_{sender}\right)$
  else if $h_1^* \neq H(H(b \cdot (A - B)))$ **then**
    return $\left(\textbf{cheat-detected}, id_{sender}\right)$
  else
    return $\left(\textbf{cheat-detected}, id_{receiver}\right)$
  end if
end function

Protocol HandleInvalidResponseDecommitment [6.5] Handle complaint of an invalid challenge response decommitment that was sent to the sender during the OT protocol

// This routine can only be used by the sender!
function HandleInvalidResponseDecommitment$(c, (\psi^*, \pi_{\psi^*}))$
  if $c, (\psi^*, \pi_{\psi^*})$ aren't signed by the receiver **then**
    return $\left(\textbf{cheat-detected}, id_{sender}\right)$
  end if
  if VerifyDecommitment$(c, \pi_{\psi^*}, \psi^*) = \perp$ **then**
    return $\left(\textbf{cheat-detected}, id_{receiver}\right)$
  else
    return $\left(\textbf{cheat-detected}, id_{sender}\right)$
  end if
end function

Protocol HandleInvalidChallengeResponse [6.6] Handle complaint of an invalid challenge response sent during the verification process in the OT protocol

// This routine can only be used by the sender!
function HandleInvalidChallengeResponse($c, \pi_{\psi^*}, \psi^*, h$)
  if c, π_{ψ^*}, ψ^* aren't signed by the receiver or h isn't signed by the sender **then**
    return $\left(\textbf{cheat-detected}, id_{sender}\right)$
  end if
  if VerifyDecommitment$(c, \pi_{\psi^*}, \psi^*) = \perp$ or $\psi^* = H(h)$ **then**
    return $\left(\textbf{cheat-detected}, id_{sender}\right)$
  end if
  return $\left(\textbf{cheat-detected}, id_{receiver}\right)$
end function

Protocol HandleInvalidChallengeResponseRebut [6.7] Handle the rebut for a complaint of an invalid challenge response sent during the verification process in the OT protocol

function HandleInvalidChallengeResponseRebut($h', c, \pi_{\psi^*}, \psi^*, h$)
  if $h' \neq h$ and h is signed by the sender **then**
    return $\left(\textbf{cheat-detected}, id_{sender}\right)$
  else
    return $\left(\textbf{cheat-detected}, id_{receiver}\right)$
  end if
end function

11. PVOT (Online) Protocol Security Analysis

11.1. Simulatability of Corrupt sender

Lemma 11.1.1. *Denote Q the corrupt sender's set of queries to the random oracle before the receiver reached line 10 in protocol $\Pi^{PVOT}_{receiver}$ [5.3], and $Fail_{extract}$ a RV indicating the event where $\nexists q \in Q$ s.t. $q = bA$ or $q = b(A - B)$. Then $\Pr[h_0 \neq H(\rho_0) \vee h_1 \neq H(\rho_1) \mid Fail_{extract}] = 1 - neg(n)$.*

Proof. We show with full details the case where $\omega = 0$. Essentially, the same argument holds for $\omega = 1$.

We'll consider a lazy evaluation of the random oracle, where the receiver queries aB and $H(aB)$ *after* receiving h_0, h_1 from the sender. Hence, it can be deduced that the random oracle received a query of aB or $H(ab)$ for the *first time* in line 11. Denote $Fail_1, Fail_2$ RVs indicating if the sender queried the random oracle for aB and $H(aB)$ respectively. Therefore:

$$\begin{aligned}\Pr[Fail_1 \vee Fail_2 \mid Fail_{extract}] &\underset{\text{union bound}}{\leq} \Pr[Fail_2 \mid Fail_{extract}] \\ &+ \Pr[Fail_2 \mid Fail_{extract}] = neg(n)\end{aligned}$$

Therefore, $h_0 \neq H(\rho_0)$ with overwhelming probability. □

The following describes the ideal-world simulator for simulating a corrupt sender's view, assuming the ability to access ideal functionality $\mathcal{F}_{comm}$:

(1) Receive B from sender

(2) Wait to receive (**chosen**) from $\mathcal{F}_{PVOT}$, then generate uniformly $A \in G$ and send A to sender

(3) Receive ξ from sender

(4) Simulate a message (**commited**) sent to sender from $\mathcal{F}_{comm}$

(5) Receive h_0, h_1 from sender. Denote Q the set of queries made by the sender to the random-oracle.

(6) If exists $q^* \in Q$ s.t. $\frac{q^*}{A} \cdot G = B$ then set $b \leftarrow \frac{q^*}{A}$. Otherwise, if exists $q^* \in Q$ s.t. $\frac{q^*}{A-B} \cdot G = B$ then set $b \leftarrow \frac{q^*}{A-B}$. Else:

a) Send **blatant-cheat** to $\mathcal{F}_{PVOT}$

b) Simulate a broadcast of a complaint $Blame(id_{receiver}, \textbf{invalid-padding})$ and halt the execution.

(7) Calculate both pads and the expected challenge:

$$\rho^0 = H(b \cdot A)$$
$$\rho^1 = H(b \cdot (A - B))$$

(8) If $h_0 \neq H(\rho^0)$ and $h_1 \neq H(\rho^1)$:

a) Send **blatant-cheat** to $\mathcal{F}_{PVOT}$

b) Simulate a broadcast of a complaint $Blame(id_{receiver}, \textbf{invalid-padding})$ and halt the execution.

Else, if exists a single $i \in \{0, 1\}$ s.t. $h_i \neq H(\rho^i)$ then set $\omega \leftarrow i$. Otherwise, set $\omega \leftarrow \perp$.

(9) If $\omega \neq \perp$, send $(\textbf{guess}, \omega)$ to $\mathcal{F}_{PVOT}$

a) If received (**cheat-detected**) from $\mathcal{F}_{PVOT}$, simulate a broadcast of a complaint $Blame(id_{receiver}, \textbf{invalid-padding})$ and halt the execution.

b) If received (**cheat-undetected**) from $\mathcal{F}_{PVOT}$, continue as follows.

(10) If $\xi \neq H(h_0) \oplus H(h_1)$:

a) Send **blatant-cheat** to $\mathcal{F}_{PVOT}$

b) Simulate a broadcast of a complaint $Blame(id_{receiver}, \textbf{invalid-challenge})$ and halt the execution.

(11) Simulate a message $(\textbf{open}, H(H(\rho^0)))$ sent to the sender from $\mathcal{F}_{comm}$

(12) If received $(\textbf{accused}, \textbf{invalid-challenge-response})$ from $\mathcal{F}_{PVOT}$, simulate sending
a message $(\textbf{rebut}, \textbf{invalid-challenge-response}, h_0)$ to arbitrator and halt the execution.

(13) Upon receiving $\tilde{\alpha}^0, \tilde{\alpha}^1$, compute the sender's inputs:

$$\alpha^0 = D_{\rho^0}(\tilde{\alpha}^0)$$
$$\alpha^1 = D_{\rho^1}(\tilde{\alpha}^1)$$

(14) Send $(\textbf{transfer}, \alpha^0, \alpha^1)$ to $\mathcal{F}_{PVOT}$

Claim 11.1.2. *The corrupt-sender's view in the real-world model and the ideal-world model are indistinguishable*

Proof. The sender's view consists of a 4-tuple of messages: $\Big(A, (\pi_{\psi^*}, \psi^*), output, VIEW_{arb}\Big)$ (the first three messages are received in line 5, line 9 and line 11 in protocol Π^{PVOT}_{sender} [5.1], respectively), *output* holds either the output or $\perp$, in case of a detected cheating attempt. $VIEW_{arb}$ includes the messages published during arbitration.
We'll use a hybrid argument in order to prove this claim. The different hybrids are defined as follows:

1. **Hybrid #1 (i.e. real world)**

2. **Hybrid #2** – same as hybrid #1, except that we add the ideal functionality $\mathcal{F}_{PVOT}$ and replace the real world output with the ideal world output. The honest party's inputs are given to the simulator, same as in previous hybrid.

   The simulator for this world works as follows:

   (1) Simulate a real world execution using the honest party's inputs.

   (2) If the honest party received an output α_ω: send m_0, m_1 to $\mathcal{F}_{PVOT}$ where

   $$m_\omega \leftarrow \alpha_\omega$$
   $$m_{1-\omega} \leftarrow 0^n$$

   (3) If the honest party detected a cheating attempt in the real-world execution, send **blatant-cheat** to $\mathcal{F}_{PVOT}$

3. **Hybrid #3** – same as hybrid #2, except that we change the commitment and opening of the challenge's response. The honest party's inputs are given to the simulator, same as in previous hybrid.

   The simulator looks like the previous simulator, but instead of sending the challenge's commitment and corresponding opening (lines 9, 16 in protocol $\Pi^{PVOT}_{receiver}$ [5.3] respectively) the simulator sends $c \leftarrow Comm(0)$ and $Open(c, H(h_0))$ (i.e. equivocate the decommitment to be $H(h_0)$)

4. **Hybrid #4** – same as hybrid #3, except that we try to extract the corrupt sender's input. The honest party's inputs are given to the simulator, same as in previous hybrid.

   The simulator for this world works as follows:

   (1) Simulate the real world protocol protocol $\Pi^{PVOT}_{receiver}$ [5.3], until line 20 (i.e. receiving encrypted messages from the sender)

   - During the execution, after receiving h_0, h_1 from the sender, execute items (5) to (8) from section 11.1 (note that if b could not be extracted, we send **blatant-cheat** to $\mathcal{F}_{PVOT}$)

   (2) Execute lines items (13) to (14) from section 11.1 (decrypt messages using sender's extracted inputs)

5. **Hybrid #5 (i.e. ideal world)** – it is the same as hybrid #4, except that we don't use the honest party's input.

   The simulator includes the following changes w.r.t. the simulator in previous hybrid:

   - Instead of using ω to calculate A, the simulator sets $A \leftarrow a^* \cdot G$ for $a^* \xleftarrow{\$} \mathbb{Z}_G$
   - Using the ideal functionality $\mathcal{F}_{PVOT}$ (instead of ω) in order to guess the receiver's input

We'll prove indistinguishability between adjacent hybrid worlds:

1. **Hybrid #1 - Hybrid #2** — the simulator perfectly simulates an execution of an honest receiver, therefore the output sent from the simulator equals to the output obtained from a real execution.

2. **Hybrid #2 - Hybrid #3** — Using the ideal commitment functionality, it is perfectly indistinguishable and also perfectly equivocable (the same argument can be extended to work for using standard indistinguishablity and equivocability).

3. **Hybrid #3 - Hybrid #4** — If the simulator extracts the value of b, he perfectly simulates the view of the previous hybrid.
   Otherwise, if we failed to extract b (i.e. event $Fail_{extract}$ occurs, as defined in lemma 11.1.1) then a **blatant-cheat** is sent by the simulator to $\mathcal{F}_{PVOT}$. By lemma 11.1.1, $\Pr[h_0 \neq H(\rho_0) \vee h_1 \neq H(\rho_1) \mid Fail_{extract}] = 1 - neg(n)$ and hence the simulated receiver in previous hybrid detects that $h_\omega \neq H(\rho_\omega)$ with all-but-negligible probability. Hence, both views in this case are indistinguishable.
   In total, the views in both cases (whether b was extracted or not) are indistinguishable and hence both hybrids are indistinguishable.

4. **Hybrid #4 - Hybrid #5** — Both values of A are identically distributed, and it doesn't affect the rest of the execution, as we only use the value of A from this point onward.

□

11.2. Simulatability of Corrupt receiver

Lemma 11.2.1. *(c.f. Lemma D.2 in Doerner, Kondi, Lee, and abhi shelat [9]) Let q be the order of a group $\mathbb{G}$ generated by G, whose elements are represented in $\kappa = |q|$ bits. If there exists a PPT algorithm $\mathcal{A}$ s.t.:*

$$\Pr\left[\mathcal{A}(1^\kappa, x \cdot G) = x \cdot x \cdot G : x \leftarrow \mathbb{Z}_q\right] = \varepsilon$$

where the probability is taken over the choice of x, then there exists an algorithm $\mathcal{A}'$ which solves the Computational Diffie-Hellman problem in $\mathbb{G}$ with advantage ε^2.

Lemma 11.2.2. *Let q be the order of a group $\mathbb{G}$ generated by G, whose elements are represented in $\kappa = |q|$ bits. Given a, A, B as calculated at lines 3 and 5 in protocol $\Pi^{PVOT}_{receiver}$ [5.3], the values of both $b \cdot A$ and $b \cdot (A - B)$ can be calculated with negligible probability.*

Proof. Falsely assuming the receiver is able to calculate $b \cdot A$ and $b \cdot (A - B)$, it means he's able to calculate $b \cdot A - b \cdot (A - B) = b \cdot B = b \cdot b \cdot G$.

By lemma 11.2.1, successfully computing $x \cdot x \cdot G$ given $X = x \cdot G$ with non-negligible probability implies breaking the Computational Diffie-Hellman assumption. Therefore this case is contradictory to the Computational Diffie-Hellman assumption and therefore not possible — the receiver can't calculate both $b \cdot A$ and $b \cdot (A - B)$. □

Lemma 11.2.3. *Denote Q the corrupt receiver's set of queries made to the random oracle before the simulated sender in Hybrid #2 reached line 3 of the ideal-world protocol, and $Bad_{extract}$ a RV indicating the event where $\nexists q \in Q : q = bA \vee q = b(A - B)$.*

Also let $Fail_1, Fail_2$ denote RVs indicating the events of failure at the decommitment verification in line 12 or the response verification in line 17 (both in protocol Π^{PVOT}_{sender} [5.1]), respectively. Then $\Pr[Fail_1 \vee Fail_2 \mid Bad_{extract}] = 1 - neg(n)$

Proof. Note that none of the queries $bA, b(A - B)$ are sent to the random oracle by the sender until he reaches line 3. That, including the assumption of event $Bad_{extract}$, implies that the value of $H(H(\rho^0))$ is determined only after receiving the challenge's response commitment. Hence, let $p = \Pr[Fail_1 \vee Fail_2 \mid Bad_{extract}]$:

$$\begin{aligned} p &= \Pr[Fail_1 \mid Bad_{extract}] \\ &\quad + \Pr[\overline{Fail_1} \mid Bad_{extract}] \cdot \Pr[Fail_2 \mid Bad_{extract}, \overline{Fail_1}] \\ &\underset{\text{union bound}}{\leq} \Pr[Fail_1 \mid Bad_{extract}] \\ &\quad + \Pr[Fail_2 \mid Bad_{extract}, \overline{Fail_1}] = neg(n) \end{aligned}$$

□

Lemma 11.2.4. *Denote Q the corrupt receiver's set of queries made to the random oracle before the sender reached item (4) in the ideal-world simulator.*

Denote $Valid$ a RV indicating that the sender reached item (4) in the ideal-world, and denote $Queried_0, Queried_1$ RVs indicating whether bA and $b(A - B)$ (respectively) were sent to the random oracle. Also, denote RV $Queried_{both} = Queried_0 + Queried_1$. Hence, $\Pr[Queried \neq 1 \mid Valid] = neg(n)$

Proof. By lemma 11.2.3 $\Pr[Queried = 0 \mid Valid] = neg(n)$, and by lemma 11.2.2 $\Pr[Queried = 2 \mid Valid] = neg(n)$. Therefore, using a union-bound:

$$\begin{aligned} \Pr[Queried \neq 1 \mid Valid] &\leq \Pr[Queried = 0 \mid Valid] \\ &\quad + \Pr[Queried = 2 \mid Valid] = neg(n) \end{aligned}$$

□

The following describes the ideal-world simulator for simulating a corrupt receiver's view, assuming the ability to access ideal functionality $\mathcal{F}_{comm}$:

(1) Execute the real-world protocol protocol Π^{PVOT}_{sender} [5.1] until reaching line 6 (i.e. generate B and receive A)

(2) Compute the challenge's value $\xi \xleftarrow{\$} \mathbb{Z}_G$ and send it to the receiver.

(3) Wait to receive c from the receiver

(4) Denote Q the set of queries made by the receiver to the random-oracle

(5) Denote query $q \in Q$ s.t. $H(q) = \rho^i$ where $i \in \{0,1\}$. set $\omega = i$. If no such query exists, send **blatant-cheat** to $\mathcal{F}_{PVOT}$

(6) Send $(\textbf{choose}, \omega)$ to $\mathcal{F}_{PVOT}$ and wait to receive $(\textbf{message}, \alpha^\omega)$ as a response.

(7) Compute the values ρ^ω, and $H(\rho^\omega)$ the same as in the real-world protocol

(8) Denote $h^\omega \leftarrow H(\rho^\omega)$ and $h^{\bar{\omega}} \xleftarrow{\$} \mathbb{Z}_G$, and configure the random oracle s.t. $H(h^{\bar{\omega}}) \leftarrow \xi \oplus H(H(\rho^\omega))$

(9) Send h^0, h^1 to the receiver

(10) Execute the real-world protocol's verification routines of the sender (using protocol Π^{PVOT}_{sender} [5.1] lines 11 to 21) (receiving challenge decommitment and checking its correctness and validity).

(11) If a cheating attempt was detected during execution, send **blatant-cheat** to $\mathcal{F}_{PVOT}$. Otherwise, continue as follows.

(12) Denote $\rho^* \xleftarrow{\$} \mathbb{Z}_G$. Calculate the following and send $\tilde{\alpha}^0, \tilde{\alpha}^1$ to the receiver:

$$\tilde{\alpha}^\omega \leftarrow E_{\rho^\omega}(\alpha^\omega)$$
$$\tilde{\alpha}^{\bar{\omega}} \leftarrow E_{\rho^*}(0^n)$$

(13) If received $(\textbf{accused}, \textbf{invalid-padding})$ from $\mathcal{F}_{PVOT}$, simulate sending a message $(\textbf{rebut}, \textbf{invalid-padding}, A, b)$ to arbitrator.

Claim 11.2.5. *The corrupt-receiver's view in the real-world model and the ideal-world model are indistinguishable*

Proof. The receiver's view consists of a 6-tuple:

$$\left(B, \xi, (h_0, h_1), (\tilde{\alpha}^0, \tilde{\alpha}^1), output, VIEW_{arb}\right)$$

where the first four messages were received in lines 3, 8, 10 and 20 respectively, *output* holds either the honest party's output, and $VIEW_{arb}$ includes the messages published during arbitration.

We'll use a hybrid argument in order to prove this claim. The different hybrids are defined as follows:

1. **Hybrid #1 (i.e. real world)**

2. **Hybrid #2** — the same as hybrid #1, except that we replace the random oracle with a (configurable) lazy-evaluating random oracle , and change the calculation of ξ to be picked at random.

   The simulator for this world is described as follows:

   a) Execute the real-world protocol protocol Π^{PVOT}_{sender} [5.1]
      - Instead of lines 6 to 10 in protocol Π^{PVOT}_{sender} [5.1], execute items (2) to (9) in the ideal-world simulator.

3. **Hybrid #3** — the same as hybrid #2, except that we extract the receiver's input ω and the sender's input α_ω. The honest sender's input $\alpha_{\bar{\omega}}$ is given to the simulator, same as in previous hybrid.

   The simulator for this world works the same as the real-world protocol, except that we execute items (4) to (6) in the ideal-world simulator before line 22 in the real-world protocol protocol Π^{PVOT}_{sender} [5.1] (i.e. encrypting α_0, α_1), in order to extract α_ω.

4. **Hybrid #4** — the same as hybrid #3, except that we pick the value of of the encryption key $\rho_{\bar{\omega}}$ at random (we use ρ^* instead of $\rho_{\bar{\omega}}$, where $\rho^* \xleftarrow{\$} \mathbb{Z}_G$).

5. **Hybrid #5 (i.e. ideal world)** — the same as hybrid #4, except that we set the value of $\alpha_{\bar{\omega}}$ to be 0.

We'll prove indistinguishability between adjacent hybrid worlds:

1. **Hybrid #1 - Hybrid #2** — The value of ξ that's sent by the simulator is indistinguishable from its value in previous hybrid — its value is uniformly distributed (as in previous hybrid) and it holds the same property as in previous hybrid (i.e. $\xi = H(H(\rho^0)) \oplus H(H(\rho^1))$, and the rest of the execution is perfectly simulated. Therefore both views are identically distributed.

2. **Hybrid #2 - Hybrid #3** — If the simulator fails to extract b (i.e. either none or both queries $bA, b(A-B)$ were sent to the random oracle), a **blatant-cheat** command is sent to $\mathcal{F}_{PVOT}$.

   By lemma 11.2.4, the receiver queries *exactly one* of bA and $b(A-B)$ with all-but-negligible probability and hence the simulator extracts the correct value of ω (depends on which of these two queries was sent by the receiver) with all-but-negligible probability. The rest of the execution, given ω, is perfectly simulated and therefore both hybrids are indistinguishable.

3. **Hybrid #3 - Hybrid #4** — We'll consider a lazy evaluation of the random oracle, where the sender evaluates $\rho_{\bar{\omega}} = H(x_{\bar{\omega}})$ (where $x_{\bar{\omega}} \in \{bA, b(A-B)\}$, depends on $\bar{\omega}$) only at line 22.
   - Note that the sender doesn't use $\rho_{\bar{\omega}}$ before that line.

By lemma lemma 11.2.4, the receiver queries only x_ω (i.e. a single $x_i \in \{x_0, x_1\}$), therefore $\rho_{\bar{\omega}} = H(x_{\bar{\omega}})$ is uniformly distributed, and therefore also identically distributed as $\rho^* \leftarrow \mathbb{Z}_G$. Hence, $E_{\rho_{\bar{\omega}}}(0^n)$ and $E_{\rho^*}(0^n)$ are indistinguishable and therefore both views are indistinguishable as well.

4. **Hybrid #4 - Hybrid #5** — the value of $\alpha_{\bar{\omega}}$ is only used when sending the encryptions of α_0, α_1 to the receiver (line 20 in the real-world protocol). Therefore, the only difference is whether the value of $\alpha_{\bar{\omega}}$ or 0^n is encrypted. We use semantically secure encryption scheme, and the encryption key is uniformly distributed, therefore both encryptions are indistinguishable and hence both views are indistinguishable.

□

11.3. Accountability and Defamation-Freeness

Claim 11.3.1. *Our OT protocol maintains the "Accountability" definition (c.f. item 2 in definition A.3.1).*

Proof. We'll prove that for any abort triggered by the honest party, the arbitrator accuses the adversary. We'll do it by enumerating the cases where the honest party aborts and show that in each case he is able to generate a proof-of-malfeasance that's used by the arbitrator in order to accuse the adversary.
We'll consider the case where the protocol's execution halted because the sender claimed the receiver is corrupt. By observing the sender's main subroutine(c.f. protocol Π^{PVOT}_{sender} [5.1]), the only conditions where the sender aborts are as follows:

1. **Invalid decommitment (c.f. line 14)** – The sender reaches this line iff the challenge's response decommitment that was received from the receiver is invalid. Then, the sender sends the (signed) messages containing c, π_{ψ^*} to the arbitrator.

   The arbitrator executes protocol HandleInvalidOpening [3.1] for the given complaint. The receiver is accused by the arbitrator because both components are signed by him, but the decommitment component is invalid.

2. **Invalid challenge response (c.f. line 18)** – the sender reaches this line iff the response ψ^* received from the receiver doesn't satisfy the equality $\psi^* = H(H(\rho^0))$.

   Then, the sender sends to the arbitrator c, π_{ψ^*} (that were received in line 9, line 11) and the value $H(\rho^0)$ that was sent to the receiver in protocol Π^{PVOT}_{sender} [5.1] at line 10. The arbitrator executes protocol HandleInvalidChallengeResponse [6.6] for the given complaint (considering an honest sender who complained, it's safe to assume that $h = H(\rho^0)$). The provided decommitment is valid (otherwise, the sender should have complained earlier at line 12 in protocol Π^{PVOT}_{sender} [5.1]) and also $\psi^* \neq H(H(\rho^0)) = H(h)$, therefore the check at line 5 fails.

   Next, the arbitrator notifies the corrupt receiver he's being accused and then waits for his response:

- If the receiver refuses to rebut, he gets accused by the arbitrator in **??**.
- If the receiver rebuts with message h', the arbitrator executes line 1. The receiver will be accused iff the message h' isn't equal to h (where $h = H(\rho^0)$, as explained above).
  The receiver holds the message h_0, which was sent by the honest sender (and therefore $h_0 = H(\rho^0) = h$.
  Therefore he's either rebuts using h_0, or forges a new message with the sender's signature. In either way, the receiver is accused with overwhelming probability.

Next, we'll consider the case where the protocol's execution halted because the receiver claimed the sender is corrupt. By observing the receiver's main subroutine(c.f. protocol $\Pi^{PVOT}_{receiver}$ [5.3]), the only conditions where the receiver aborts are as follows:

1. **Inconsistent shared padding (c.f. line 12)** – denote ρ^ω_r the value of ρ^ω calculated by the receiver, and h_0, h_1 the values of ρ^ω calculated by the sender (which equals to $H(\rho^0), H(\rho^1)$ if the sender is honest). The receiver reaches this line iff $h_\omega \neq H(\rho^\omega_r)$ (which equals to $H(H(aB))$ by ρ^w's definition).
   This case occurs due to one of the following reasons:
   a) $\omega = 0$ and $h_0 \neq H(H(a \cdot B))$
   b) $\omega = 1$ and $h_1 \neq H(H(a \cdot B))$

   The receiver sends a, B to the arbitrator (using the value of B received form the sender). The arbitrator executes protocol HandleInconsistentPadding [6.3] for the given complaint. He waits to receive b, A from the sender – if the sender doesn't rebut, he gets accused.

   Now we will handle separately the case where $\omega = 0$ and $\omega = 1$:

Case 1: ($\omega = 0$) The receiver complains because $h_0 \neq H(H(aB))$

i. If the check at line 2 does not pass, the sender is accused. Otherwise, either the adversary forged the receiver's signature (which cannot happen with more than negligible probability) or $A^* = A = aG$, since the receiver does not sign anything else).

ii. If the check at line line 5 does not pass, the sender is accused. Otherwise we can assume $B = bG$.

iii. Since $A = aG$ and $B = bG$, it follows that $aB = abG = baG = bA$, and therefore $H(aB) = H(bA)$.

iv. Since $h_0 \neq H(H(aB)) = H(H(bA))$, the check at line 7 passes and the sender is accused.

Case 2: ($\omega = 1$) The receiver complains because $h_1 \neq H(H(aB))$

i. If the check at line 2 does not pass, the sender is accused. Otherwise, either the adversary forged the receiver's signature (which cannot happen with more than negligible probability) or $A^* = A = aG + \omega B = aG + \omega B$, since the receiver does not sign anything else).

ii. If the check at line line 5 does not pass, the sender is accused. Otherwise we can assume $B = bG$ and therefore $A = aG + bG$.

iii. Since $A = aG$ and $B = bG$, it follows that:

$$aB = abG = baG = b(\underbrace{aG + bG}_{A} - bG) = b(A - B)$$

Therefore, $H(aB) = H(b(A - B))$.

iv. Since $h_1 \neq H(H(aB)) = H(H(b(A - B)))$, the check at line 9 passes and the sender is accused.

2. **Invalid challenge (c.f. line 15)** – the receiver reaches this line iff $\xi \neq H(h_0) \oplus H(h_1)$, for challenge ξ and pads h_0, h_1 sent by the sender.
Then, he sends to the arbitrator the signed messages, that were sent by the sender, containing the values of ξ, h_0, h_1. The arbitrator executes protocol HandleInvalidChallenge [6.2] for the given complaint. The check at line 5 passes because all the components sent by the receiver to the arbitrator are signed by the sender and they don't satisfy the equality: $\xi = H(h_0) \oplus H(h_1)$, therefore the arbitrator accuses the sender.

□

Claim 11.3.2. *Our OT protocol maintains the "Defamation-Free" definition (c.f. item 3 in definition A.3.1).*

Proof. We'll prove that the honest party can't be falsely accused by the arbitrator by showing that all complaints (considering the sender and the complaint type) sent to the arbitrator can't cause a false accusation of the honest party.

Due to claim 11.3.1, the arbitrator will always accuse the adversary for complaints presented by the honest party and therefore defamation-freeness is guaranteed in this case.
Hence, if the corrupt party wants to frame the honest party it would falsely claim one of the following:

1. **Invalid challenge (c.f. protocol HandleInvalidChallenge [6.2])** – considering a corrupt receiver, the sender is accused only in line 6. The arbitrator reaches this line iff the corrupt receiver provides the components ξ, h_0, h_1 signed by the sender and $\xi \neq H(h_0) \oplus H(h_1)$.

   However, the honest sender sends to the receiver signed values of ξ, h_0, h_1 only once (c.f. line 10 in protocol Π^{PVOT}_{sender} [5.1]), where these values satisfy the equation $\xi = H(h_0) \oplus H(h_1)$. The adversary can't generate different messages that are signed by the sender with more than negligible probability which means the sender is accused with negligible probability.

2. **Invalid challenge response (c.f. protocol HandleInvalidChallengeResponse [6.6])** – considering a corrupt sender, the receiver is accused only in line 5 or in case of no rebut (i.e. **??** in the judgment routine). Therefore, we'll consider the case where the preceding checks in line 2 and line 5 failed, otherwise the sender is accused.

   Because the check at line 2 fails it's safe to assume that $\text{VERIFYDECOMMITMENT}(c, \pi_\rho, \rho) \neq \perp$ because the honest receiver sent a signed valid decommitment and the sender can't forge a valid decommitment signed by the receiver with more than negligible probability. Moreover, by definition $\rho = H(h_0)$, where h_0 is the value received by the receiver at line 10 in protocol $\Pi^{PVOT}_{receiver}$ [5.3]. Also, because the check at line 5 failed as well, we can deduce that $\rho \neq H(h)$ – we explained above why the provided decommitment is valid, hence the check fails iff $\rho \neq H(h)$.

   In **??**, the arbitrator notifies the receiver that he's being accused. The receiver executes line 2 in response and rebuts by sending h_0 (where h_0 was received in line 10 in protocol $\Pi^{PVOT}_{receiver}$ [5.3]). Note that the receiver rebuts w.p. 1 and therefore the arbitrator never reaches **??** in the judgment routine.

   Then, the arbitrator continues and performs the check at line 2: as we specified above $\rho = H(h_0)$ which means $H(h_0) \neq H(h)$ and therefore $h \neq h_0$. Considering an honest receiver it's safe to assume the value h_0 provided as a rebut is signed by the sender, therefore the check passes except for a negligible probability which means the sender is accused except for a negligible probability.

3. **Inconsistent padding (c.f. protocol HandleInconsistentPadding [6.3])** – considering a corrupt receiver, the sender is accused only in line 3, line 6, line 8 or line 10.

   The arbitrator reaches line 3 iff the honest sender's rebuttal contains a message containing A that's not signed by the receiver. The sender acts honestly and therefore provides the message received in line 5, hence the arbitrator never reaches line 3.

   Considering the next case, the arbitrator reaches line 6 iff the sender's rebut contains a value b s.t. $B \neq b \cdot G$. The sender acts honestly and therefore the value B generated by him (that was sent to the receiver in line 3 and also was provided to the arbitrator) is valid – i.e. $B = b \cdot G$ for $b \xleftarrow{\$} \mathbb{Z}_q$, and b is also included in the sender's rebut. Therefore, $B = b \cdot G$ and hence the check at line 5 passes with negligible probability – meaning that the sender is accused at line 6 with negligible probability.

   Considering the next case, the arbitrator reaches line 8 iff $h_0^* \neq H(H(b \cdot A))$. The message h_0^* provided by the receiver is signed by the sender, therefore it corresponds to the value of h_0 as sent in line 10. Moreover, the sender is honest and therefore the value of h_0 was calculated correctly, i.e. $h_0 = H(H(b \cdot A))$. Hence, $h_0^* = H(H(b \cdot A))$ and therefore the check at line 7 will pass with negligible probability which means the sender will be accused in line 8 with negligible probability.

The final case (i.e. line 8) is similar to the previous case – the honest sender calculates h_1 correctly and therefore $h_1^* = h_1 = H(H(b \cdot (A - B)))$. Hence, the check at line 9 will pass with negligible probability which means the sender will be accused in line 10 with negligible probability.

□

12. PVOT (Offline) Protocol: Overview

In this section we describe our publicly-verifiable OT protocol in the preprocessing model. The protocol is based on *black-box* use of random OT. In a random-OT protocol, the parties do not have inputs; instead, the sender receives two random outputs, while the receiver is given a random choice bit and the corresponding output. We don't require the underlying random OT to support provable outputs because the preprocessing model allows us to perform an early, unverifiable, abort when corrupt behavior is detected in the preprocessing stage.

We use the random OT to construct a verifiable $(\frac{1}{2}, 0)$-weak OT protocol. (In $(p, 0)$-weak OT, the sender is given the receiver's choice bit with probability p.)

From $(\frac{1}{2}, 0)$-weak PVOT we build a public verifiable OT using a standard reduction from weak to full OT, presented in Crépeau and Kilian [7] (with a small modification to support input *strings*, rather than bits). We show that this reduction preserves output verifiability, giving us a full PVOT protocol. For a full, formal specification of the protocol, see chapter 13.

The security analysis appears in chapter 14.

13. Full Description of the PVOT (Offline) Protocol

In this chapter we give a full, formal description of our protocol to realize $\mathcal{F}_{PVOT}$.

13.1. Preliminaries

Let H be a collision-resistant hash function from 2κ to κ bits. (κ is the security parameter). [I think a one-way should be enough, actually]

We'll also use a strong extractor H'. Think of this as family of hash functions from 2κ to κ bits, with the property that if X has min-entropy more than κ, then $K, H'_K(X)$ is (very close to) uniformly distributed, when K is a uniform key.

13.2. Ideal functionality $\mathcal{F}_{(p,q)-PVWOT}$

Upon corruption, the ideal functionality tosses random coin $coin_p$ ($coin_q$) w.p. p (q) for determining whether to send c, α^c (α^0, α^1) to the corrupt sender (receiver). In case of a corrupt instance (indicated by the tossed coin), a message *corrupt* is sent to the adversary and when the honest party will send his input, it will be sent to to the adversary as well.The formal description of the functionality appears in fig. 13.2.1

13.3. PVOT (offline) main routines

13.3.1. PVWOT main routines

The sender's main routine is described in protocol $\Pi^{sender}_{(\frac{1}{2},0)-PVWOT}$ [6.1].

The receiver's main routine is described in protocol $\Pi^{receiver}_{(\frac{1}{2},0)-PVWOT}$ [6.2].

The arbitrator's main routine is described in protocol Π^{PVWOT}_{arb} [6.3]. It uses protocol HandleOTComplaint [5.6] for handling the parties' complaints (if such are published).

13.3.2. PVOT (offline) main routines

The sender's main routine is described in protocol $\Pi^{(\frac{1}{2}^{\kappa},0)-PVWOT}_{\text{sender}}$ [6.1].

The receiver's main routine is described in protocol $\Pi^{(\frac{1}{2}^{\kappa},0)-PVWOT}_{\text{receiver}}$ [6.2].

The arbitrator's main routine is described in protocol $\Pi^{(\frac{1}{2}^{\kappa},0)-PVWOT}_{\text{arb}}$ [6.3].

Figure 13.2.1.: Functionality $\mathcal{F}_{(p,q)-PVWOT}$

Upon corruption: If the adversary corrupts the sender (receiver), w.p. p (q) send the adversary a message *corrupt*.
Preprocessing: Wait to receive "DonePreprocessing" from both parties. If received message "abort" from any party, send "abort" to both parties.
Online
Inputs: Wait to receive message $(\textbf{transfer}, \alpha^0, \alpha^1)$ from the sender and $(\textbf{choose}, c)$ from the receiver.

- If the sender is corrupt and the ideal functionality is corrupt, send c to the sender.
- If the receiver is corrupt and the ideal functionality is corrupt, send (α^0, α^1) to the receiver.

Output: Send α^c to the receiver.
Prove output: Output message $(\textbf{prove-output}, c, \alpha^c)$

Protocol $\Pi^{sender}_{(\frac{1}{2},0)-PVWOT}$ **[6.1]** Publicly-verifiable OT pseudo-code

function SENDERROUTINE(α^0, α^1)
  // Offline phase
  Receive (a_0, a_1) from $\mathcal{F}^{malicious}_{\text{random-OT}}$
  Send $\big((H(a_0), H(a_1)), \sigma(H(a_0), H(a_1))\big)$ to receiver
  Set $k_0, k_1 \xleftarrow{\$} \{0,1\}^\ell$ and send $\big((k_0, k_1), \sigma(k_0, k_1)\big)$ to receiver

  // Online phase
  Receive $(b', \sigma(b'))$ from the receiver
  Send $\big((b', \sigma(b')), \sigma(b', \sigma(b'))\big)$ to the receiver
  Compute z_0, z_1 s.t.

$$z_0 = \alpha^b \oplus H'_{k_0}(a_0)$$
$$z_1 = \alpha^{1-b} \oplus H'_{k_1}(a_1)$$

  and send $\big((z_0, z_1), \sigma(z_0, z_1)\big)$ to the receiver
end function

Protocol $\Pi^{receiver}_{(\frac{1}{2},0)-PVWOT}$ **[6.2]** Publicly-verifiable OT pseudo-code

1: **function** ReceiverRoutine(c)
2: // Offline phase
3: Receive $\big((h'_0, h'_1), \sigma(h')\big)$ from the sender
4: Receive $\big((k'_0, k'_1), \sigma(k')\big)$ from the sender
5: Receive (r, a_r) from $\mathcal{F}^{malicious}_{\text{random-OT}}$
6: **if** $h'_r \neq H(a_r)$ **then**
7: **return** $\perp$
8: **else if** $\sigma(k)$ doesn't correspond to the sender's signature for input (k_0, k_1) **then**
9: **return** $\perp$
10: **else if** $\sigma(h)$ doesn't correspond to the sender's signature for input (h_0, h_1) **then**
11: **return** $\perp$
12: **end if**

13: // Online phase
14: Set $b \leftarrow c \oplus r$ and send $(b, \sigma(b))$ to sender
15: Receive $(b'', \sigma(b''^*))$from the sender
16: **if** $b'' \neq (b, \sigma(b))$ or $\sigma(b''^*)$ doesn't correspond to the sender's signature for input b'' **then**
17: **Broadcast** message $\Big(\textbf{invalid-double-signature}, (b'', \sigma(b''^*))\Big)$
18: **return** $\perp$
19: **end if**
20: Receive $((z'_0, z'_1), \sigma(z'))$ from the sender
21: **return** $H'_{k'_r}(a_r) \oplus {z'}_r$
22: **end function**

23: **function** ProveOutput
24: Denote $(b'', \sigma(b''^*)), r, a_r, \big(h'_0, h'_1, \sigma(h')\big), \big(k'_0, k'_1, \sigma(k')\big), \big(z'_0, z'_1, \sigma(z'^*)\big)$ the messages received in lines 3 to 5 and 20, respectively.
25: **Broadcast** message $\Big(\textbf{prove-output}, ((b'', \sigma(b''^*)), r, a_r, \big(h'_0, h'_1, \sigma(h')\big), \big(k'_0, k'_1, \sigma(k')\big), \big(z'_0, z'_1, \sigma(z'^*)\big)$
26: **return** $\perp$
27: **end function**

Protocol $\Pi_{\mathbf{arb}}^{PVWOT}$ [6.3] The arbitration routine of the WOT protocol

function Π_{ARB}^{PVWOT}
  Wait for a complaint $\Big(reason, (message)\Big)$.
  if received a complaint **then**
    Denote P_i the sender of the complaint
    $res \leftarrow$ HandlePVWOTComplaint$(P_i, reason, message)$
    if $res \neq \perp$ **then**
      return res
    end if
  end if
end function

Protocol HandlePVWOTComplaint [6.4] Publicly-verifiable WOT pseudo-code

function HandlePVWOTComplaint$(id, reason, message)$
  if $reason =$ **prove-output** and $id = id_{receiver}$ **then**
    parse $message$ as a bit b''', group elements $a^*, (h_0'', h_1''), (k_0'', k_1''), (z''_0, z''_1)$, and a signature for each parsed argument.
    $res \leftarrow$ PVWOTHandleProveOutput$((b''', \sigma(b'''^*)), a^*, \big(h_0'', h_1'', \sigma(h'')\big), \big(k_0'', k_1'', \sigma(k'')\big), \big(z''_0, z$
    // c.f. protocol PVWOTHandleProveOutput [6.6]
  else if $reason =$ **invalid-double-signature** and $id = id_{receiver}$ **then**
    parse $message$ as a bit b''', and a signature $\sigma(b'''^*)$.
    $res \leftarrow$ PVWOTHandleInvalidDoubleSignature$(b''', \sigma(b'''^*))$ // c.f. protocol PVWOTHandleInvalidDoubleSignature [6.5]
  end if
  return res
end function

Protocol PVWOTHandleInvalidDoubleSignature [6.5] Publicly-verifiable OT pseudo-code

function PVWOTHandleInvalidDoubleSignature($(b''', \sigma(b'''^*))$)
  if $\sigma(b'''^*)$ doesn't correspond to the sender's signature for input b''' **then**
    return $\Big($**cheat-detected**$, id_{receiver}\Big)$
  end if
  Denote $b''' = (b'''_1, \sigma(b'''^*_1))$
  if $\sigma(b'''^*_1)$ doesn't correspond to the receiver's signature for input b'''_1 **then**
    return $\Big($**cheat-detected**$, id_{sender}\Big)$
  else
    return $\Big($**cheat-detected**$, id_{receiver}\Big)$
  end if
  Denote $i \in \{0,1\}$ the first index s.t. $h''_i = H(a^*)$
  Return (**prove-output**$, i \oplus b'''_1, H'_{k''_i}(a^*) \oplus z''_i)$
end function

Protocol PVWOTHandleProveOutput [6.6] Publicly-verifiable OT pseudo-code

function PVWOTHandleProveOutput$((b''', \sigma(b'''^*)), a^*, \big(h''_0, h''_1, \sigma(h'')\big), \big(k''_0, k''_1, \sigma(k'')\big), \big(z''_0, z''_1, \sigma$
  if $\sigma(h'')$ doesn't correspond to the sender's signature for input (h''_0, h''_1) **then**
    Return $id_{receiver}$
  else if $\sigma(k'')$ doesn't correspond to the sender's signature for input (k''_0, k''_1) **then**
    Return $id_{receiver}$
  else if $\sigma(z'')$ doesn't correspond to the sender's signature for input (z''_0, z''_1) **then**
    Return $id_{receiver}$
  else if $\sigma(b'''^*)$ doesn't correspond to the sender's signature for input b''' **then**
    Return $id_{receiver}$
  else if for all $i \in \{0,1\}$: $h''_i \neq H(a^*)$ **then**
    Return $id_{receiver}$
  end if
  Denote $b''' = (b'''_1, \sigma(b'''^*_1))$
  if $\sigma(b'''^*_1)$ doesn't correspond to the receiver's signature for input b'''_1 **then**
    Return $id_{receiver}$
  end if
  Denote $i \in \{0,1\}$ the first index s.t. $h''_i = H(a^*)$
  Return (**prove-output**$, i \oplus b'''_1, H'_{k''_i}(a^*) \oplus z''_i)$
end function

Protocol $\Pi^{(\frac{1}{2}^{\kappa},0)-PVWOT}_{\textbf{sender}}$ **[6.1]** Sender's routine in PVOT (offline) protocol

1: **function** $\Pi^{(\frac{1}{2}^{\kappa},0)-PVWOT}_{\text{SENDER}}(\alpha^0, \alpha^1)$
2: If exists $i \in [1,\kappa]$ s.t. $\mathcal{F}^i_{(\frac{1}{2},0)-PVWOT}$ returned an "abort" message, abort the execution.
3: For $i \in [1, \kappa-1]$, Sample $a_{0,i} \xleftarrow{\$} \{0,1\}^\kappa$ and then set:

$$a_{0,\kappa} = \alpha^0 \oplus \big(\bigoplus_{i=1}^{n-1} a_{0,i}\big)$$
$$a_{1,i} = \alpha^0 \oplus \alpha^1 \oplus a_{0,i} \qquad (\forall i \in [1,\kappa])$$

4: For $i \in [1,\kappa]$: send $(\textbf{transfer}, a_{0,i}, a_{1,i})$ to $\mathcal{F}^i_{(\frac{1}{2},0)-PVWOT}$
5: **end function**

Protocol $\Pi^{(\frac{1}{2}^{\kappa},0)-PVWOT}_{\textbf{receiver}}$ **[6.2]** Receiver's routine in PVOT (offline) protocol

1: **function** $\Pi^{(\frac{1}{2}^{\kappa},0)-PVWOT}_{\text{RECEIVER}}(c)$
2: If exists $i \in [1,\kappa]$ s.t. $\mathcal{F}^i_{(\frac{1}{2},0)-PVWOT}$ broadcasted an "abort" message, abort the execution.
3: For $i \in [1, \kappa-1]$, Sample $c_i \xleftarrow{\$} \{0,1\}$ and then set:

$$c_\kappa = c \oplus \big(\bigoplus_{i=1}^{\kappa-1} c_i\big)$$

4: For $i \in [1,\kappa]$: send $(\textbf{choose}, c_i)$ to $\mathcal{F}^i_{(\frac{1}{2},0)-PVWOT}$
5: **end function**
6: **function** ProveOutput
7: For $i \in [1,\kappa]$: send **prove-output** to $\mathcal{F}^i_{(\frac{1}{2},0)-PVWOT}$
8: **end function**

Protocol $\Pi^{(\frac{1}{2}^{\kappa},0)-PVWOT}_{\textbf{arb}}$ **[6.3]** Arbitrator's routine in PVOT (offline) protocol — its single purpose is for proving the receiver's output

1: **function** PreprocessOTArbiration
2: Wait to receive **prove-output** messages from all $\mathcal{F}^i_{(\frac{1}{2},0)-PVWOT}$ (for $i \in [1,n]$). If received **abort-preprocessing** from $\mathcal{F}^i_{(\frac{1}{2},0)-PVWOT}$ (for any $i \in [1,n]$), then **return abort-preprocessing**.
3: For all $i \in [1,\kappa]$, denote $(\textbf{prove-output}, c_i, x_i)$ the output message sent from $\mathcal{F}^i_{(\frac{1}{2},0)-PVWOT}$
4: **return** message $(\textbf{prove-output}, \bigoplus_{i=1}^{\kappa} c_i, \bigoplus_{i=1}^{\kappa} x_i)$
5: **end function**

14. PVOT (Offline) protocol security analysis

14.1. Publicly Verifiable (p, q)-Weak OT (PVWOT) Protocol

14.1.1. Simulatability of corrupt sender

The following describes the ideal-world simulator for simulating a corrupt sender's view:

1. Get response res from $\mathcal{F}_{PVWOT}$, indicating whether $\mathcal{F}_{PVWOT}$ is a corrupt instance or not.
2. Set $a_0, a_1 \xleftarrow{\$} \{0,1\}^{2\kappa}$, and simulate sending (a_0, a_1) to the sender from $\mathcal{F}^{malicious}_{\text{random-OT}}$
3. Wait for the sender to send k_0, k_1, and h_0, h_1:
   - If $h_0 \neq H(a_0)$ and $h_1 \neq H(a_1)$, send "abort" to $\mathcal{F}_{PVWOT}$
   - If exists a single $i \in \{0,1\}$ s.t. $h_i \neq H(a_i)$, observe the value of res:

   Case 1: res = honest, simulate an abort of the honest party.

   Case 2: res = corrupt, set $r \leftarrow 1 - i$. Simulate message "PreprocessingDone" from the honest party. Then wait to receive the honest receiver's input c from $\mathcal{F}_{PVWOT}$, and then simulate an honest execution of the receiver using input c and randomness r.
4. Simulate message "PreprocessingDone" from the honest party.
5. Send $b \xleftarrow{\$} \{0,1\}$ to the sender (in this case $h_0 = H(a_0)$ and $h_1 = H(a_1)$)
6. Wait for the sender to send z_0, z_1, and extract α^0, α^1:

$$\alpha^b = z_0 \oplus H'_{k_0}(a_0)$$
$$\alpha^{1-b} = z_1 \oplus H'_{k_1}(a_1)$$

7. Send $(\textbf{transfer}, \alpha^0, \alpha^1)$ to $\mathcal{F}_{PVOT}$
8. If received $(\textbf{prove-output}, c, \alpha^c)$ from $\mathcal{F}_{PVOT}$:
   a) Set $r \xleftarrow{\$} c \oplus b$
   b) Simulate a broadcast message $(\textbf{prove-output}, a_r, h_r, k_r, z_r)$ sent from the receiver.

Claim 14.1.1. *The corrupt-sender's view in the real-world model and the ideal-world model are indistinguishable*

Proof. The environment's view consists of a tuple of messages, denoted by the following RVs:

$$(A^0, A^1), B, OUT, VIEW^{arb}, OUT^{arb}$$

where (A^0, A^1) correspond to the randomness chosen by $\mathcal{F}^{malicious}_{\text{random-OT}}$, B corresponds to the receiver's bit b, OUT corresponds to the honest party's output and $VIEW_{arb}$ corresponds to the messages sent during arbitration.
We'll use a hybrid argument in order to prove this claim. The different hybrids are defined as follows:

1. **Hybrid #0 (i.e. real world)**
2. **Hybrid #1** — the same as previous hybrid, except that we choose r differently. The simulator receives the honest party's input c.
   The simulator for this world use the real-world protocol with the following changes:
   - We configure $\mathcal{F}^{malicious}_{\text{random-OT}}$ to output r according to item 3
3. **Hybrid #2** — the same as previous hybrid, except that we don't use the honest party's input c.
   The simulator for this world use the real-world protocol with the following changes:
   - Set $b \xleftarrow{\$} \{0,1\}$, and when proving output set $r \leftarrow b \oplus c$
4. **Hybrid #3 (i.e. ideal world)** — the same as previous hybrid, except that we change the honest party's output.
   The simulator for this world use the real-world protocol with the following changes:
   - The honest party receives the output from the ideal functionality

For hybrid $\#i$, denote the following RVs:

$$(A^0_i, A^1_i), B_i, OUT_i, VIEW^{arb}, OUT^{arb}_i$$

We'll prove indistinguishability between adjacent hybrids:

1. **Hybrid #0 – Hybrid #1**: The only change difference between these hybrids is the value of r generated by $\mathcal{F}^{malicious}_{random-OT}$.

   Hence, in order to prove indistinguishability between the hybrids we need to prove that the value of r is identically distributed in both hybrids. For $i \in \{0,1\}$ denote the RV R_i corresponding to the value of r in hybrid $\#i$.

   In hybrid #0 r is uniformly-generated by $\mathcal{F}^{malicious}_{random-OT}$, therefore:

$$\Pr(R_0 = r) = \frac{1}{2}$$

In hybrid #1, the value of r depends on the correctness of h_0, h_1 that were sent by the sender. Denote X_{diff} the event that exactly one of h_0, h_1 is invalid. Let $p_1 = \Pr(R_1 = r)$, hence:

$$\begin{aligned} p_1 &= \Pr(R_1 = b \wedge \overline{X_{diff}}) + \Pr(R_1 = r \wedge X_{diff}) \\ &= \Pr(\overline{X_{diff}}) \cdot \underbrace{\Pr(R_1 = r \mid \overline{X_{diff}})}_{\frac{1}{2}\ (*)} + \Pr(X_{diff}) \cdot \Pr(R_1 = r \mid X_{diff}) \\ &= \frac{1}{2} \cdot \Pr(\overline{X_{diff}}) + \Pr(X_{diff}) \cdot \underbrace{\Pr(R_1 = r \mid X_{diff})}_{p^{\text{not-same}}} \end{aligned}$$

We will show that $p^{\text{not-same}} = \frac{1}{2}$ — assuming X_{diff}, i.e. only one $h_i \neq H(a_i)$ ($i \in \{0,1\}$), $R_1 = r$ iff $h_{1-b} \neq H(a_{1-b})$ and $\mathcal{F}_{PVWOT}$ is corrupt, or $h_b \neq H(a_b)$ and $\mathcal{F}_{PVWOT}$ is honest. Let $p_1^{\text{not same}} = \Pr(R_1 = r \mid X_{diff})$, and X_{honest} an event indicating that $\mathcal{F}_{PVWOT}$ is honest. hence:

$$\begin{aligned} p_1^{\text{not same}} &= \Pr(X_{honest} \wedge h_b \text{ is invalid} \mid X_{diff}) \\ &\quad + \Pr(\overline{X_{honest}} \wedge h_{1-b} \text{ is invalid} \mid X_{diff}) \\ &= \underbrace{\Pr(X_{honest} \mid X_{diff})}_{\frac{1}{2}} \cdot \Pr(h_b \text{ is invalid} \mid X_{diff}) \\ &\quad + \underbrace{\Pr(\overline{X_{honest}} \mid X_{diff})}_{\frac{1}{2}} \cdot \Pr(h_{1-b} \text{ is invalid} \mid X_{diff}) \\ &= \frac{1}{2} \cdot \underbrace{\Big(\Pr(h_b \text{ is invalid} \mid X_{diff}) + \Pr(h_{1-b} \text{ is invalid} \mid X_{diff}) \Big)}_{1 \text{ (complementary events)}} \\ &= \frac{1}{2} \end{aligned}$$

Revisiting p_1, we get that:

$$\begin{aligned} p_1 &= \frac{1}{2} \cdot \Pr(\overline{X_{diff}}) + \Pr(X_{diff}) \cdot \underbrace{\Pr(R_1 = r \mid X_{diff})}_{p_1^{not-same}} \\ &= \frac{1}{2} \cdot \Big(\underbrace{\Pr(\overline{X_{diff}}) + \Pr(X_{diff})}_{1 \text{ (complementary events)}} \Big) = \frac{1}{2} \end{aligned}$$

Hence, $\Pr(R_0 = r) = \Pr(R_1 = r)$. Therefore, both views are indistinguishable.

(*) If both are valid/invalid (i.e. $\overline{X_{diff}}$) — the simulator sends a random $r \xleftarrow{\$} \{0,1\}$.

2. **Hybrid #1 – Hybrid #2**: For $i \in \{1,2\}$ denote $VIEW_i^{pref} = (A_i^0, A_i^1)$. We only values changed in hybrid #2 are B_2 and $VIEW_2^{arb}$.

These hybrids differ after the simulator reach item 5 — notice that, in this case, the value of r is revealed only after receiving a **prove-output** command. Therefore, we can set the value of r retroactively *after* receiving a **prove-output** command, and set $b \xleftarrow{\$} \{0,1\}$ instead.

3. **Hybrid #2 – Hybrid #3**: For $i \in \{2,3\}$ denote $VIEW_i^{pref} = \big((A_i^0, A_i^2), B_i\big)$. We only value changed in hybrid #3 is OUT_3, therefore:

$$\Pr[VIEW_2^{pref} = view^{pref}] = \Pr[VIEW_3^{pref} = view^{pref}]$$

We need to prove that

$$\Pr[OUT_2 = out \mid VIEW_2^{pref} = view^{pref}] = \Pr[OUT_3 = out \mid VIEW_3^{pref} = view^{pref}]$$

Note that the simulator reached item 6 (in both hybrids) iff $h_0 = H(a_0)$ and $h_2 = H(a_2)$. Therefore, the output in the output in hybrid #2 is $z_r \oplus h_r$, which is the same as in hybrid #3 because the simulator sends to $\mathcal{F}_{PVWOT}$ the inputs $(z_0 \oplus h_2, z_2 \oplus h_2)$ — and therefore the honest party receives $z_r \oplus h_r$.

Hence,

$$\Pr[OUT_2 = out \mid VIEW_2^{pref} = view^{pref}] = \Pr[OUT_3 = out \mid VIEW_3^{pref} = view^{pref}]$$

□

14.1.2. Simulatability of corrupt receiver

The following describes the ideal-world simulator for simulating a corrupt receiver's view:

1. Simulate an honest sender in offline phase
   a) Set $a_0, a_1 \xleftarrow{\$} \{0,1\}^{2\kappa}$, $r \xleftarrow{\$} \{0,1\}$ and simulate sending (r, a_r) to the receiver from $\mathcal{F}^{malicious}_{\text{random-OT}}$
2. Receive b' from the receiver, and set $c \leftarrow b' \oplus r$
3. Send $(\textbf{choice}, c)$ to $\mathcal{F}_{PVOT}$ and receive α^c as a response.
4. Set z_0, z_1 as follows:

$$z_r = \alpha^r \oplus H'_{k_r}(a_r)$$
$$z_{1-r} \xleftarrow{\$} \{0,1\}^{\kappa}$$

5. Send z_0, z_1 to the receiver

Claim 14.1.2. *The corrupt-receiver's view in the real-world model and the ideal-world model are indistinguishable*

Proof. The environment's view consists of a tuple of messages, denoted by the following RVs:

$$(X_0, X_1), (R, A^R), (H^0, H^1), (K^0, K^1), (Z^0, Z^1), OUT, OUT^{arb}$$

where (X_0, X_1) correspond to the honest sender's inputs, (R, A^R) correspond to the randomness chosen by $\mathcal{F}^{malicious}_{\text{random-OT}}$, $(H^0, H^1), (K^0, K^1), (Z^0, Z^1)$ correspond to the sender's messages received in **??**, OUT corresponds to the honest party's output and OUT^{arb} corresponds to the arbitrator's output.

Denote the following hybrids:

1. Hybrid #0 (i.e. real world)
2. Hybrid #1 — similar to previous hybrid, except that we compute z_{1-r} differently. The simulator receive the input α^{1-r}, as in previous hybrid.
   Calculate z_{1-r} as follows:
   $$z_{1-r} = \alpha^{1-r} \oplus U_\kappa$$
3. Hybrid #2 (i.e. ideal world)— similar to previous hybrid, except that we compute z_{1-r} differently. The simulator doesn't receive input α^{1-r}.
   Calculate z_{1-r} as follows:
   $$z_{1-r} = U_\kappa$$

For hybrid $\#i$, denote the following RVs:

$$VIEW_i = \left((X_0, X_1), (R_i, A_i^R), (H_i^0, H_i^1), (K_i^0, K_i^1), (Z_i^0, Z_i^1), OUT_i, OUT_i^{arb}\right)$$

We'll prove indistinguishability between adjacent hybrids:

1. **Hybrid #0 - Hybrid #1** — For $i \in \{0, 1\}$ we'll denote

   $$\begin{aligned} VIEW_i^{ext} &= (X_{1-R_i} \oplus Z_i^{1-R_i}, VIEW_i) \\ &= (X_{1-R_i} \oplus Z_i^{1-R_i}, (X_0, X_1), (R_i, A_i^R), (H_i^0, H_i^1), (K_i^0, K_i^1), (Z_i^0, Z_i^1), OUT_i, OUT_i^{arb}) \end{aligned}$$

   We'll prove that $VIEW_0^{ext} \approx VIEW_1^{ext}$, and then deduce that $VIEW_0 \approx VIEW_1$.

   In hybrid #0, $Z_0^{1-R_0} = X_{1-R_0} \oplus H'_{K_0^{1-R_0}}(A_0^{1-R_0})$ and hence

   $$VIEW_0^{ext} = \left(H'_{K_0^{1-R_0}}(A_0^{1-R_0}), (X_0, X_1), (R_0, A_0^R), (H_0^0, H_0^1), (K_0^0, K_0^1), (Z_0^0, Z_0^1), OUT_0, OUT_0^{arb}\right)$$

   while in hybrid #1, $Z_0^{1-R_0} = X_{1-R_0} \oplus U_\kappa$ and hence

   $$VIEW_1^{ext} = \left(U_\kappa, (X_0, X_1), (R_1, A_1^R), (H_1^0, H_1^1), (K_1^0, K_1^1), (Z_1^0, Z_1^1), OUT_1, OUT_1^{arb}\right)$$

   Note that A^{1-R_i} has κ bits of entropy, Note that $H_i^{1-R_i} = H(A^{1-R_i}) \in \{0,1\}^\kappa$, and $A^{1-R_i} \in \{0,1\}^{2\kappa}$. Therefore A^{1-R_i} has at least $2\kappa - \kappa = \kappa$ bits of entropy.

Also, note that $K_i^{1-R_i} \xleftarrow{\$} U_\ell$, and that H' is a strong extractor. Hence, by definition of H'

$$(K_i^{1-R_i}, H'_{K_i^{1-R_i}}(A_i^{1-R_i})) \approx (K_i^{1-R_i}, U_\kappa)$$

and therefore

$$\begin{aligned}\big(H'_{K_0^{1-R_0}}(A_0^{1-R_0}), (X_0, X_1), (R_0, A_0^R), (H_0^0, H_0^1), (K_0^0, K_0^1), (Z_0^0, Z_0^1), OUT_0, OUT_0^{arb}\big) \\ \approx \big(U_\kappa, (X_0, X_1), (R_1, A_1^R), (H_1^0, H_1^1), (K_1^0, K_1^1), (Z_1^0, Z_1^1), OUT_1, OUT_1^{arb}\big)\end{aligned}$$

2. **Hybrid #1 - Hybrid #2** — In hybrid #1, as calculated in previous bullet,

$$\Pr[Z_1^{1-r} = z \mid VIEW_1^{pref} = view^{pref}] = neg(\kappa)$$

In hybrid #2, the value of Z_2^{1-r} is uniformly generated, hence

$$\Pr[Z_2^{1-r} = z \mid VIEW_2^{pref} = view^{pref}] = neg(\kappa)$$

Therefore $\Pr[Z_1^{1-r} = z \mid VIEW_1^{pref} = view^{pref}] = \Pr[Z_2^{1-r} = z \mid VIEW_2^{pref} = view^{pref}]$

□

14.2. Reducing 1-2 PVOT protocol to $(\frac{1}{2}, 0)$-PVWOT protocol

14.2.1. Corrupt sender simulatability

The following describes the ideal-world simulator for simulating a corrupt sender's view:

1. Receive res from $\mathcal{F}_{(\frac{1}{2}^\kappa, 0)-PVWOT}$, indicating if the functionality is corrupt or honest.
   - If $res = corrupt$, for all $i \in [1, \kappa]$, simulate message *corrupt* from $\mathcal{F}^i_{(\frac{1}{2}, 0)-PVWOT}$ to the sender
   - If $res = honest$,
     - perform a rejection sampling of a subset $I_{corrupt} \subset [1, \kappa]$ until $I_{corrupt} \neq [1, \kappa]$: for all $i \in [1, \kappa]$, $i \in I_{corrupt}$ w.p. $\frac{1}{2}$.
     - For all $i \in I_{corrupt}$, simulate message *corrupt* from $\mathcal{F}^i_{(\frac{1}{2}, 0)-PVWOT}$ to the sender, and for all $i \notin I_{corrupt}$, simulate message *honest* from $\mathcal{F}^i_{(\frac{1}{2}, 0)-PVWOT}$ to the sender.
2. If $res = corrupt$, wait to receive the honest receiver's input c from $\mathcal{F}^i_{(\frac{1}{2}^\kappa, 0)-PVWOT}$:
   - For all $i \in [1, \kappa]$, set $c_i \xleftarrow{\$} \{0, 1\}$ s.t. $\bigoplus_i c_i = c$

- for $i \in [1,\kappa]$ wait for the sender to send input message $(\textbf{transfer}, a_0^i, a_1^i)$ to $\mathcal{F}^i_{(\frac{1}{2},0)-PVWOT}$
  - Simulate a message c_i from $\mathcal{F}^i_{(\frac{1}{2},0)-PVWOT}$ to the sender
- Send $(\textbf{transfer}, z_0, z_1)$ to $\mathcal{F}_{(\frac{1}{2}^\kappa,0)-PVWOT}$, where $z_r = \bigoplus_{i\in[1,n]} a^i_{c_i}$ and $z_{1-r} \xleftarrow{\$} U_\kappa$

3. If $res = honest$:
   - Denote $i^* \notin I_{corrupt}$, for all $i \neq i^*$ set $c_i \xleftarrow{\$} \{0,1\}$
   - for $i \in [1,\kappa]$ wait for the sender to send input message $(\textbf{transfer}, a_0^i, a_1^i)$ to $\mathcal{F}^i_{(\frac{1}{2},0)-PVWOT}$
     - If $i \in I_{corrupt}$, simulate a message $c_i \xleftarrow{\$} \{0,1\}$ from $\mathcal{F}^i_{(\frac{1}{2},0)-PVWOT}$ to the sender
   - Send $(\textbf{transfer}, a_0^{i^*} \oplus \bigoplus_{i\neq i^*} a^i_{c_i}, a_1^{i^*} \oplus \bigoplus_{i\neq i^*} a^i_{c_i})$ to $\mathcal{F}_{(\frac{1}{2}^\kappa,0)-PVWOT}$

Claim 14.2.1. *The corrupt-receiver's view in the real-world model and the ideal-world model are indistinguishable*

Proof sketch. The environment's view consists of a tuple of messages, denoted by the following RVs:

$$\{COIN_i\}_{i=1}^\kappa, \{C_i\}_{i=1}^\kappa, OUT$$

where $COIN_i$ (for all $i \in [1,n]$) corresponds to the coin tossed by $\mathcal{F}^i_{(\frac{1}{2},0)-PVWOT}$, indicating wether the ideal functionality is corrupt or not, C_i (for all $i \in [1,\kappa]$) corresponds to the honest party's input that was received from $\mathcal{F}^i_{(\frac{1}{2},0)-PVWOT}$, and OUT corresponds to the honest receiver's output.

The ideal world simulator changes the values of $\{COIN_i\}_{i=1}^\kappa, \{C_i\}_{i=1}^\kappa$.

First, we'll prove that

$$\{COIN_i^{REAL}\}_{i\in[1,\kappa]} \approx \{COIN_i^{IDEAL}\}_{i\in[1,\kappa]}$$

Denote $X_{corrupt}$ the event that $\mathcal{F}_{(\frac{1}{2}^\kappa,0)-PVWOT}$ is corrupt. In the real world, each coin is uniformly distributed, hence for all $i \in [1,\kappa]$:

$$\Pr[COIN_i^{REAL} = corrupt] = \frac{1}{2}$$

In the ideal world,

$$\begin{aligned}\Pr[COIN_i^{IDEAL} = corrupt] &= \Pr[COIN_i^{IDEAL} = corrupt \wedge X_{corrupt}] \\ &\quad + \Pr[COIN_i^{IDEAL} = corrupt \wedge \overline{X_{corrupt}}] \\ &= \underbrace{\Pr[X_{corrupt}]}_{\frac{1}{2}^{\kappa}} \cdot \underbrace{\Pr[COIN_i^{IDEAL} = corrupt \mid X_{corrupt}]}_{(*)1} \\ &\quad + \underbrace{\Pr[\overline{X_{corrupt}}]}_{1-\frac{1}{2}^{\kappa}} \cdot \underbrace{\Pr[COIN_i^{IDEAL} = corrupt \mid \overline{X_{corrupt}}]}_{\Pr[i \in I_{corrupt} \mid \overline{X_{corrupt}}]} \\ &= \frac{1}{2}^{\kappa} + (1 - \frac{1}{2}^{\kappa}) \cdot \underbrace{\Pr[i \in I_{corrupt} \mid \overline{X_{corrupt}}]}_{(**)\frac{1}{2}} \\ &= \frac{1}{2}^{\kappa} + (1 - \frac{1}{2}^{\kappa}) \cdot (\frac{1}{2} - \frac{1}{2}^{\kappa}) = \frac{1}{2} + neg(\kappa)\end{aligned}$$

Therefore,

$$\{COIN_i^{REAL}\}_{i\in[1,\kappa]} \approx \{COIN_i^{IDEAL}\}_{i\in[1,\kappa]}$$

(*) Considering a corrupt instance, $I_{corrupt} = [1, n]$ and hence $\Pr[COIN_i^{IDEAL} = corrupt \wedge X_{corrupt}] = 1$ for all $i \in [1, \kappa]$.

(**) Considering an honest instance, $I_{corrupt}$ is generated s.t. $i \in I_{corrupt}$ w.p. $\frac{1}{2}$ for each $i \in [1, \kappa]$. Hence $\Pr[COIN_i^{IDEAL} = corrupt \wedge \overline{X_{corrupt}}] = \frac{1}{2}$ for all $i \in [1, \kappa]$.

The last argument remaining is that

$$\{C_i^{REAL}\}_{i\in[1,\kappa]} \mid \{COIN_i^{REAL}\}_{i\in[1,\kappa]} \approx \{C_i^{IDEAL}\}_{i\in[1,\kappa]} \mid \{COIN_i^{IDEAL}\}_{i\in[1,\kappa]}$$

In the real world, $\{C_i^{REAL}\}_{i\in[1,\kappa]}$ are uniformly distributed, s.t. $\bigoplus_{i\in[1,\kappa]} M_i^{real} = \alpha^c$.

A similar argument holds for $\{C_i^{ideal}\}_{i\in[1,\kappa]}$, *in the case of a corrupt functionality.*

In the case of an *honest functionality*, the sender receives only a subset of $\{C_i^{ideal}\}_{i\in S}$ for a subset $S \subset [1, \kappa]$ (and $C_i^{ideal} = \perp$ for all $i \notin S$) and therefore

$$\{C_i^{REAL}\}_{i\in[1,\kappa]} \mid \{COIN_i^{REAL}\}_{i\in[1,\kappa]} \approx \{C_i^{IDEAL}\}_{i\in[1,\kappa]} \mid \{COIN_i^{IDEAL}\}_{i\in[1,\kappa]}$$

in that case as well. □

14.2.2. Corrupt receiver simulatability

The following describes the ideal-world simulator for simulating a corrupt receiver's view:

1. Let i^* be the index of the last instance of $\mathcal{F}_{(\frac{1}{2},0)-PVWOT}$ which the receiver sent his input message to.
2. For $j \in [1, \kappa]$, denote $(\textbf{choose}, c_j)$ the message sent from the receiver to $\mathcal{F}^j_{(\frac{1}{2},0)-PVWOT}$.
3. For $j \neq i^*$:

- Simulate a message ("output", a_j) from $\mathcal{F}^j_{(\frac{1}{2},0)-PVWOT}$ to the receiver, where $a_j \xleftarrow{\$} \{0,1\}^\kappa$

4. Denote $c = \bigoplus_{i=1}^{\kappa} c_i$. Send c to $\mathcal{F}_{(\frac{1}{2}^\kappa,0)-PVWOT}$ and receive α^c as a response.

5. Simulate a message ("output", $\alpha^c \oplus \bigoplus_{j \neq i^*} a_j$) from $\mathcal{F}^{i^*}_{(\frac{1}{2},0)-PVWOT}$ to the receiver

Claim 14.2.2. *The corrupt-receiver's view in the real-world model and the ideal-world model are indistinguishable*

Proof sketch. The environment's view consists of a tuple of messages, denoted by the following RVs:

$$\{M_i\}_{i=1}^{\kappa}, OUT$$

where M_i (for all $i \in [1,\kappa]$) corresponds to the message received from $\mathcal{F}^i_{(\frac{1}{2},0)-PVWOT}$, and OUT corresponds to the honest sender's output.

The ideal world simulator changes the values of the messages $\{M_i^{ideal}\}_{i\in[1,\kappa]}$ only. Hence, in order to prove indistinguishability between the real and ideal views, we need to prove that

$$\{M_i^{real}\}_{i\in[1,\kappa]} \approx \{M_i^{ideal}\}_{i\in[1,\kappa]}$$

In the real world, $\{M_i^{real}\}_{i\in[1,\kappa]}$ are uniformly distributed, s.t. $\bigoplus_{i\in[1,\kappa]} M_i^{real} = \alpha^c$.

A similar argument holds for $\{M_i^{ideal}\}_{i\in[1,\kappa]}$, hence

$$\{M_i^{real}\}_{i\in[1,\kappa]} \approx \{M_i^{ideal}\}_{i\in[1,\kappa]}$$

□

15. Handling Non-Responsive Adversaries

In this section we describe how to build, based on a given two-party protocol π that realizes functionality $\mathcal{F}$, a protocol Π_π that realizes $\mathcal{F}$ and has the same security properties as π, in addition to the ability to handle *non-responsive* adversaries.

Non-responsive behavior refers to a case where the protocol requires a party to transmit a message, but this party refuses to send *any* message — whether it's a valid one or not.

We assume, w.l.o.g., π requires the parties to alternate in sending messages (otherwise, we can always concatenate successive messages into one long message, or, alternatively, add empty messages from the other party to ensure alternation).

15.1. Overview

Our protocol for ensuring responsiveness requires each protocol message of π to be prepended with an instance id *sid*, a sender id *pid*, a sequence number t and to be signed by the sender. The sequence numbers are per-instance (e.g., party P_0 sends message 1, then P_1 sends message 2, then P_0 sends message 3).

If the protocol instructs a party to terminate, it will send a final (signed) *termination message* consisting of the next sequence number and the special symbol $\perp$. W.l.o.g, assume party P_0 sends the first message of π.

Complaint

If a party P is non-responsive, the other party will broadcast a complaint containing the tuple (m, σ, m', σ'), where m is the last message received from party P and m' is the succeeding message in the protocol (sent by the complaining party).

Rebuttal

If a party P is accused of being non-responsive with a complaint (m, σ, m', σ'), such that m has the prefix (sid, P, i) and m' has the prefix $(sid, P', i+1)$, P will broadcast a rebuttal message m'', with sequence number $i+2$ (as specified by the protocol).

Arbitration

On receiving a complaint of the form (m, σ, m', σ'), the arbitrator will perform the following actions:

1. If (m, σ) was previously received by the arbitrator then ignore. Otherwise,

2. Verify that σ is a valid signature on m by party P and σ' is a valid signature on m' by party P'.

3. Parse the prefix of m to extract the tuple (sid, P, i, v) and the prefix of m' to extract the tuple (sid', P', i').

4. If $v = \perp$, $sid \neq sid'$, $P = P'$ or $i' \neq i + 1$ then ignore the complaint (it is invalid). Otherwise,

5. Wait to receive a rebuttal message (m'', σ'') (or for a timeout to elapse)

6. If (m'', σ'') was received, σ'' is a valid signature on m'' by party P and the prefix of m'' contains $(sid, P, i + 2)$, then ignore the complaint. Otherwise,

7. When timeout has elapsed, output (**cheat-detected**, P).

15.2. Security Sketch

The reason that this protocol isn't entirely trivial is that it must prevent the adversary from framing an honest party. To see how this can happen, consider a "toy" protocol in which one party can demand the i^{th} message from the other, for any i. Suppose the honest party has only received the first message from the adversary, and the adversary now demands message 10. The honest party cannot produce this message—it might require previous information that the adversary has not provided.

To prevent this type of attack, we require the complaining party to provide a proof that the accused party *can* construct the i^{th} message. This proof consists of a signed message showing that it has sent message $i - 2$ (thus, it must have received message $i - 3$), as well as the following protocol message from the complainer (message $i - 1$).

Bibliography

[1] Gilad Asharov and Claudio Orlandi. Calling out cheaters: Covert security with public verifiability. In Xiaoyun Wang and Kazue Sako, editors, *Advances in Cryptology - ASIACRYPT 2012 - 18th International Conference on the Theory and Application of Cryptology and Information Security, Beijing, China, December 2-6, 2012. Proceedings*, volume 7658 of *Lecture Notes in Computer Science*, pages 681–698. Springer, 2012. doi: 10.1007/978-3-642-34961-4_41. URL `https://doi.org/10.1007/978-3-642-34961-4_41`.

[2] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. More efficient oblivious transfer and extensions for faster secure computation. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 535–548. ACM, 2013. doi: 10.1145/2508859.2516738.

[3] Yonatan Aumann and Yehuda Lindell. Security against covert adversaries: Efficient protocols for realistic adversaries. *J. Cryptology*, 23(2):281–343, 2010. doi: 10.1007/s00145-009-9040-7.

[4] Ran Canetti, Ben Riva, and Guy N. Rothblum. Practical delegation of computation using multiple servers. In Yan Chen, George Danezis, and Vitaly Shmatikov, editors, *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*, pages 445–454. ACM, 2011. doi: 10.1145/2046707.2046759.

[5] Nishanth Chandran, Vipul Goyal, Rafail Ostrovsky, and Amit Sahai. Covert multi-party computation. *48th Annual IEEE Symposium on Foundations of Computer Science (FOCS07)*, 2007. doi: 10.1109/focs.2007.61. `https://web.cs.ucla.edu/~rafail/PUBLIC/83.pdf`.

[6] Richard Cleve. Limits on the security of coin flips when half the processors are faulty (extended abstract). In Juris Hartmanis, editor, *Proceedings of the 18th Annual ACM Symposium on Theory of Computing, May 28-30, 1986, Berkeley, California, USA*, pages 364–369. ACM, 1986. doi: 10.1145/12130.12168.

[7] Claude Crépeau and Joe Kilian. Achieving oblivious transfer using weakened security assumptions (extended abstract). In *29th Annual Symposium on Foundations of Computer Science, White Plains, New York, USA, 24-26 October 1988*, pages 42–52. IEEE Computer Society, 1988. doi: 10.1109/SFCS.1988.21920.

[8] Ivan Damgård, Martin Geisler, and Jesper Buus Nielsen. From passive to covert security at low cost. In Daniele Micciancio, editor, *Theory of Cryptography, 7th Theory of Cryptography Conference, TCC 2010, Zurich, Switzerland, February 9-11, 2010. Proceedings*, volume 5978 of *Lecture Notes in Computer Science*, pages 128–145. Springer, 2010. doi: 10.1007/978-3-642-11799-2_9. URL `https://doi.org/10.1007/978-3-642-11799-2_9`.

[9] Jack Doerner, Yashvanth Kondi, Eysa Lee, and abhi shelat. Secure two-party threshold ECDSA from ECDSA assumptions. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, pages 980–997. IEEE Computer Society, 2018. doi: 10.1109/SP.2018.00036.

[10] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In Alfred V. Aho, editor, *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA*, pages 218–229. ACM, 1987. doi: 10.1145/28395.28420.

[11] Vipul Goyal, Payman Mohassel, and Adam D. Smith. Efficient two party and multi party computation against covert adversaries. In Nigel P. Smart, editor, *Advances in Cryptology - EUROCRYPT 2008, 27th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Istanbul, Turkey, April 13-17, 2008. Proceedings*, volume 4965 of *Lecture Notes in Computer Science*, pages 289–306. Springer, 2008. doi: 10.1007/978-3-540-78967-3_17. URL `https://doi.org/10.1007/978-3-540-78967-3_17`.

[12] Cheng Hong, Jonathan Katz, Vladimir Kolesnikov, Wen-jie Lu, and Xiao Wang. Covert security with public verifiability: Faster, leaner, and simpler. In Yuval Ishai and Vincent Rijmen, editors, *Advances in Cryptology - EUROCRYPT 2019 - 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19-23, 2019, Proceedings, Part III*, volume 11478 of *Lecture Notes in Computer Science*, pages 97–121. Springer, 2019. doi: 10.1007/978-3-030-17659-4_4. URL `https://doi.org/10.1007/978-3-030-17659-4_4`.

[13] Yan Huang, Jonathan Katz, and David Evans. Quid-pro-quo-tocols: Strengthening semi-honest protocols with dual execution. In *IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA*, pages 272–284. IEEE Computer Society, 2012. doi: 10.1109/SP.2012.43.

[14] Vladimir Kolesnikov and Alex J. Malozemoff. Public verifiability in the covert model (almost) for free. *IACR Cryptol. ePrint Arch.*, 2015:1067, 2015. URL `http://eprint.iacr.org/2015/1067`.

[15] Yehuda Lindell. Fast cut-and-choose-based protocols for malicious and covert adversaries. *J. Cryptology*, 29(2):456–490, 2016. doi: 10.1007/s00145-015-9198-0.

[16] Payman Mohassel and Matthew Franklin. Efficiency tradeoffs for malicious two-party computation. In *Proceedings of the 9th International Conference on Theory and Practice of Public-Key Cryptography*, PKC06, page 458473, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3540338519. doi: 10.1007/11745853_30. URL `https://doi.org/10.1007/11745853_30`.

[17] Payman Mohassel and Ben Riva. Garbled circuits checking garbled circuits: More efficient and secure two-party computation. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part II*, volume 8043 of *Lecture Notes in Computer Science*, pages 36–53. Springer, 2013. doi: 10.1007/978-3-642-40084-1_3. URL `https://doi.org/10.1007/978-3-642-40084-1_3`.

[18] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th Annual Symposium on Foundations of Computer Science, Toronto, Canada, 27-29 October 1986*, pages 162–167. IEEE Computer Society, 1986. doi: 10.1109/SFCS.1986.25.

A. Secure Multiparty Computation: Existing Security Definitions

A.1. Malicious Security

The formal definition is copied verbatim from Section 2.2 in Aumann and Lindell [3].

Multiparty computation. A multiparty protocol problem is cast by specifying a random process that maps sets of m-inputs to sets of m-outputs (one for each party). We will assume that the number of parties m is fixed, but as noted in [12], this can easily be generalized to the case that m is a parameter. We refer to such a process as a functionality and denote it $f : (\{0,1\}^*)^m \to (\{0,1\}^*)^m$ where $f = (f_1, \ldots, f_m)$. That is, for every vector of inputs $\bar{x} = (x_1, \ldots, x_m)$, the output-vector is a random variable $\bar{y} = (f_1(\bar{x}), \ldots, f_m(\bar{x}))$ ranging over vectors of strings. The i^{th} party P_i, with input x_i, wishes to obtain $f_i(\bar{x})$. We sometimes denote such a functionality by $(\bar{x}) \mapsto (f_1(\bar{x}), \ldots, f_m(\bar{x}))$ Thus, for example, the oblivious transfer functionality is denoted by $((x_0, x_1), \sigma) \mapsto (\lambda, x_\sigma)$, where (x_0, x_1) is the first party's input, σ is the second party's input, and λ denotes the empty string (meaning that the first party has no output). We assume the existence of special symbols abort and corrupted that are not in the range of f (these have special meaning, as will be seen later).

Execution in the ideal model. As we have mentioned, some malicious behavior cannot be prevented (for example, early aborting). This behavior is therefore incorporated into the ideal model. Let the set of parties be $P_1, \ldots, P_m$ and let $I \subseteq [m]$ denote the indices of the corrupted parties, controlled by an adversary $\mathcal{A}$. An ideal execution proceeds as follows:

- **Inputs:** Each party obtains an input; the i^{th} party's input is denoted x_i. The adversary $\mathcal{A}$ receives an auxiliary input denoted 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 abort (by replacing the input x_i with a special abort $_i$ message), 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 its auxiliary input z. Denote the vector of inputs sent to the trusted party by $\bar{w}$ (note that $\bar{w}$ does not necessarily equal $\bar{x}$). If the trusted party receives an input of the form abort $i \in I$, it sends abort $_i$ to all parties and the ideal execution terminates. (If it receives abort $_i$ for more than one i, then it takes any arbitrary one, say the smallest i, and ignores all others.) Otherwise, the execution proceeds to the next step.

- **Trusted party sends outputs to adversary:** The trusted party computes $(f_1(\bar{w}), \ldots, f_m(\bar{w}))$ and sends $f_i(\bar{w})$ to party P_i, for all $i \in I$ (i.e., to all corrupted parties).
- **Adversary instructs trusted party to continue or halt:** $\mathcal{A}$ sends either continue or abort $_i$ to the trusted party (for some $i \in I$). If it sends continue, the trusted party sends $f_j(\bar{w})$ to party P_j, for all $j \notin I$ (i.e., to all honest parties). Otherwise, if it sends abort $_i$, the trusted party sends abort $_i$ to all parties P_j for $j \notin I$
- **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 $\{f_i(\bar{w})\}_{i\in I}$ obtained from the trusted party

Let $f : (\{0,1\}^*)^m \to (\{0,1\}^*)^m$ be an m-party functionality, where $f = (f_1, \ldots, f_m)$, let $\mathcal{A}$ be a non-uniform probabilistic polynomial-time machine, and let $I \subseteq [m]$ be the set of corrupted parties. Then, the ideal execution of f on inputs $\bar{x}$, auxiliary input z to $\mathcal{A}$ and security parameter n denoted $IDEAL_{f,\mathcal{A}(z),I}(\bar{x}, n)$, is defined as the output vector of the honest parties and the adversary $\mathcal{A}$ from the above ideal execution.

Execution in the real model. Let f be as above and let π be an m -party protocol for computing f. Furthermore, let $\mathcal{A}$ be a non-uniform probabilistic polynomial-time machine and let I be the set of corrupted parties. Then, the real execution of π on inputs $\bar{x}$, auxiliary input z to $\mathcal{A}$ and security parameter n, denoted $\mathrm{REAL}_{\pi,\mathcal{A}(z),I}(\bar{x}, n)$, is defined as the output vector of the honest parties and the adversary $\mathcal{A}$ from the real execution of π.

We will consider executions where all inputs are of the same length, and will therefore say that a vector $\bar{x} = (x_1, \ldots, x_m)$ is balanced if for every i and j it holds that $|x_i| = |x_j|$.

Definition A.1.1 (secure multiparty computation)**.** Let f and π be as above. Protocol π is said to securely compute f with abort in the presence of malicious adversaries 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 $I \subseteq [m]$

$$\left\{\mathrm{IDEAL}_{f,\mathcal{S}(z),I}(\bar{x}, n)\right\}_{\bar{x},z\in(\{0,1\}^*)^{m+1};n\in\mathbb{N}} \stackrel{\mathrm{c}}{\equiv} \left\{\mathrm{REAL}_{\pi,\mathcal{A}(z),I}(\bar{x}, n)\right\}_{\bar{x},z\in(\{0,1\}^*)^{m+1};n\in\mathbb{N}}$$

where $\bar{x}$ is a balanced vector.

A.2. Covert Security with Explicit Cheat Formulation

The formal definition is copied verbatim from Section 3.3 in Aumann and Lindell [3].

In the ideal model for the explicit cheat formulation for covert adversaries, the adversary can send the following additional special instructions:

- Special input $corrupted_i$: If the ideal-model adversary sends $corrupted_i$ instead of an input, the trusted party sends $corrupted_i$ to all honest parties 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 $cheat_i$: If the ideal-model adversary sends $cheat_i$ instead of an input, the trusted party hands it all of the honest parties inputs. Then, the trusted party tosses coins and with probability ϵ 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 $corrupted_i$ to all honest parties. If it was not detected, the trusted party gives the ideal-model adversary the ability to set the outputs of the honest parties to whatever values it wishes. Thus, a $cheat_i$ input is used to model a protocol execution in which the real-model adversary decides to cheat. Such cheating is always successful in the ideal model in that the adversary learns the honest parties inputs. However, as required, this cheating is also always detected with probability at least ϵ. Note also that if the cheat attempt is not detected then the adversary is given full cheat capability including the ability to determine the honest parties outputs.

Let $\epsilon : \mathbb{N} \to [0, 1]$ be a function. Then, the ideal execution with ϵ proceeds as follows:

Inputs: Each party obtains an input; the i^{th} partys 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 $i \in \mathcal{I}$ and the auxiliary input z. Denote the vector of inputs sent to the trusted party by $\bar{w}$.

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

Attempted cheat option: If a corrupted party sends $w_i = cheat_i$ to the trusted party as its input, then the trusted party sends to the adversary all of the honest parties

inputs $\{x_j\}_{j\notin\mathcal{I}}$ (as above, if multiple $cheat_i$ messages are sent, the trusted party ignores all but one). In addition,

1. With probability ϵ, the trusted party sends $corrupted_i$ to the adversary and all of the honest parties.
2. With probability $1-\epsilon$, the trusted party sends undetected to the adversary. Following this, the adversary sends the trusted party output values $\{y_j\}_{j\notin\mathcal{I}}$ of its choice for the honest parties. Then, for every $j\notin\mathcal{I}$, the trusted party sends y_j to P_j.

The ideal execution then ends at this point.
If no w_i equals $abort_i$, $corrupted_i$ or $cheat_i$, the ideal execution continues below.

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

Trusted party answers honest parties: After receiving its outputs, the adversary sends either $abort_i$ for some $i\in\mathcal{I}$, or continue to the trusted party. If the trusted party receives continue then it sends $f_j(w)$ to all honest parties P_j ($j\notin\mathcal{I}$). Otherwise, if it receives $abort_i$ for some $i\in\mathcal{I}$, it sends $abort_i$ 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 $\{x_i\}_{i\in\mathcal{I}}$, 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 $\left\{IDEAL^{\epsilon}_{f,\mathcal{S}(z),i^*}(\bar{x},n)\right\}_{\bar{x},z\in(\{0,1\}^*)^3}$.

Definition A.2.1 (Covert security with ϵ-deterrence (explicit cheat formulation))**.** Let f and π as in definition A.1.1, and let $\epsilon:\mathbb{N}\to[0,1]$ be a function.
Protocol π is said to securely compute f in the presence of covert adversaries with ϵ-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 $i^*\in\{0,1\}$:

$$\left\{IDEAL^{\epsilon}_{f,\mathcal{S}(z),i^*}(\bar{x},n)\right\}_{\bar{x},z\in(\{0,1\}^*)^3;n\in\mathbb{N}}\equiv\left\{REAL_{\pi,\mathcal{A}(z),i^*}(\bar{x},n)\right\}_{\bar{x},z\in(\{0,1\}^*)^3;n\in\mathbb{N}}$$

where $\bar{x}$ is a balanced vector

A.3. Covert Security with Public Verifiability

The formal definition is copied verbatim from Section 2.3 in Asharov and Orlandi [1].
Let f be a two-party functionality. We consider the triplet $(\pi, Blame, Judgement)$. The

algorithm *Blame* gets as input the view of the honest party (in case of cheat detection) and outputs a certificate *Cert*. The verification algorithm, *Judgement*, takes as input a certificate *Cert* and outputs the identity id (for instance, the verification key) of the party to blame or $\perp$ in the case of an invalid certificate.

The protocol: Let π be a two party protocol. If an honest party detects a cheating in π then the honest party is instructed to compute *Cert = Blame(view)* and send it to the adversary.
Let $REAL_{\pi,A(z),i^*}(x_0, x_1; 1^n)$ denote the output of the honest party and the adversary on a real execution of the protocol π where P_0, P_1 are invoked with inputs x_0, x_1, the adversary is invoked with an auxiliary input z and corrupts party P_{i^*} for some $i \in \{0, 1\}$.

The ideal world. The ideal world is exactly as Definition 1. Let $IDEAL_{\pi,A(z),i^*}(x_0, x_1)$ denote the output of the honest party, together with the output of the simulator, on an ideal execution with the functionality f, where P_0, P_1 are invoked with inputs x_0, x_1, respectively, the simulator $\mathcal{S}$ is invoked with an auxiliary input z and the corrupted party is P_{i^*} , for some $i \in \{0, 1\}$.

Notations. Let $EXEC_{\pi,A(z)}(x_0, x_1; r_0, r_1; 1^n)$ denote the messages and the outputs of the parties in an execution of the protocol π with adversary $\mathcal{A}$ on auxiliary input z, where the inputs of P_0, P_1 are x_0, x_1, respectively, and the random tapes are (r_0, r_1). Let $EXEC_{\pi,A(z)}(x_0, x_1; 1^n)$ denote the probability distribution of $EXEC_{\pi,A(z)}(x_0, x_1; r_0, r_1; 1^n)$ where (r_0, r_1) are chosen uniformly at random.
Let $OUTPUT\Big(EXEC_{\pi,A(z)}(x_0, x_1; 1^n)\Big)$ denote the output of the honest party in the execution described above. We are now ready to define the security properties.

Definition A.3.1 (Covert security with ϵ-deterrent and public verifiability)**.** Let f, π, *Blame*, *Judgement* be as above. We say that $(\pi, Blame, Judgement)$ securely computes f in the presence of a covert adversary with ϵ-deterrent and public verifiability if the following conditions hold:

1. **(Simulatability with ϵ-deterrent)**
   The protocol π (where the honest party broadcasts $Cert = Blame(view)$ if it detects cheating) is secure against a covert adversary according to the *explicit cheat formulation with ϵ-deterrent* (see definition A.2.1).

2. **(Accountability)**
   For every PPT adversary $\mathcal{A}$ corrupting party P_{i^*} for $i^* \in \{0, 1\}$, there exists a negligible function $\mu(.)$ such that for all sufficiently large $x_0, x_1, z \in (\{0, 1\}^*)^3$ the following holds:
   If $OUTPUT\Big(EXEC_{\pi,A(z),i^*}(x_0, x_1; 1^n)\Big) = corrupted_{i^*}$ then:

   $$\Pr[Judgement(cert) = id_{i^*}] > 1 - \mu(n)$$

   where *Cert* is the output certificate of the honest party in the execution.

3. **(Defamation-Free)**
   For every PPT adversary $\mathcal{A}$ controlling $i^* \in \{0,1\}$ and interacting with the honest party, there exists a negligible function $\mu(.)$ such that for all sufficiently large $x_0, x_1, z \in (\{0,1\}^*)^3$:

$$\Pr[Cert^* \leftarrow \mathcal{A}; Judgement(Cert^*) = id_{1-i^*}] < \mu(n)$$

תקציר

חישוב בטוח מרובה משתתפים – Secure Multi-Party Computation – מאפשר לכל קבוצת משתתפים לבצע כל חישוב שהוא, מבלי לדרוש מאף אחד לגלות את הקלט הפרטי שלו לשאר המשתתפים.

במודל הבטיחות Covert Security, רמת הבטיחות מוגדרת ע"י פרמטר הרתעה המייצג את ההסתברות לגלות ניסיון רמאות במהלך ריצת הפרוטוקול. הגדרה זו באה למדל מקרים שבהם התנהגות זדונית של משתתף עלולה לגרור ענישה, במידה וייתפס – למשל תשלום קנס, איבוד אמינות וכו'.

בעוד שהבסיס לקבלת המודל של Covert Security הוא המחיר שמשתתף זדוני יצטרך לשלם עבור ניסיון רמאות, משתתפים תמימים צריכים להיות מסוגלים לספק *הוכחה פומבית* לכך שהתבצע ניסיון רמאות. דרישה זו באה לידי ביטוי במודל הבטיחות Covert Security with *Public Verifiability* (CPV Security).

כיום, כאשר גובר השימוש בפרוטוקולי blockchain, המודל של CPV Security הופך למאוד רלוונטי, מכיוון שניתן להשתמש ב-blockchain על מנת לאכוף תשלומים – בפרט, לאכוף ענישה במקרה של רמאות במהלך ריצת הפרוטוקול. שימוש כזה מבסס את המוטיבציה לייעל את *עלות הבדיקה* של הוכחת רמאות, מכיוון שביצוע חישובים על-גבי ה-blockchain עולה כסף כתלות בסיבוכיות החישוב.

הפרוטוקולים הקיימים כיום לא יעילים מבחינה זו – נדרש ביצוע של פעולות כבדות חישובית או לחלופין סימולציית ריצה של הפרוטוקול כולו.

בעבודה זו, (1) אנחנו מגדירים את המודל של Covert Security with *Interactive* Public Verifiability, הכללה רעיונית של CPV Security המאפשרת התממשקות לפרוטוקולי blockchain לצורך אכיפת קנסות, (2) אנחנו מציגים פרוטוקול יעיל לחישוב בטוח לשני שחקנים (Two-Party Secure Computation) שמקיים את ההגדרה לעיל, כאשר עלות הבדיקה של הוכחת רמאות כוללת רק בדיקת חתימות, חישובי Hash ופעולות בינאריות.

בנוסף, על מנת לאפשר ביצוע חישוב תגובתי (reactive computation) בעזרת הפרוטוקול שלנו אנחנו מציגים שיטה כללית שבעזרתה ניתן להפוך כל פרוטוקול שמקיים את הגדרת הבטיחות שלנו לפרוטוקול שמבטיח, בנוסף, את נכונות התוצאה של החישוב – כלומר כל ניסיון רמאות שהוא לא יגרום לשינוי ערך הפלט המתקבל. בנוסף, שיטה זו משמרת את יעילות הבדיקה של הפרוטוקול המקורי.

עבודה זו בוצעה בהדרכתו של דר' טל מורן מבי"ס אפי ארזי למדעי המחשב, המרכז הבינתחומי, הרצליה.

המרכז הבינתחומי בהרצליה
בית-ספר אפי ארזי למדעי המחשב
התכנית לתואר שני (M.Sc.) – מסלול מחקרי

חישוב יעיל מרובה משתתפים במודל Covert עם יכולת הוכחה פומבית

מאת
ערן גולדשטיין

עבודת תזה המוגשת כחלק מהדרישות לשם קבלת תואר מוסמך M.Sc.
במסלול המחקרי בבית ספר אפי ארזי למדעי המחשב, המרכז הבינתחומי הרצליה

אוגוסט 2020