# Foundations of Secure E-Commerce: The Order Layer

Amir Herzberg, Igal Yoffe
Bar-Ilan University, Ramat-Gan, 52900, Israel
{herzbea,ioffei}@cs.biu.ac.il

October 18, 2006

**Abstract**

We present specifications and provable protocol, for secure ordering and provision of digital goods and services. Notably, our protocol includes fully automated resolution of disputes between providers and customers. Disputes may involve the timely receipt of orders and goods, due to communication failures and malicious faults, as well as disputes of fitness of goods and order. The protocol and specifications are modular, with precise yet general-purpose interfaces. This allows usage as an underlying service to different e-commerce scenarios and applications, in particular secure online banking and brokerage. The protocol is practical, efficient, reliable and secure, under realistic failure and delay conditions. Our design and specifications are a part of a layered architecture for secure e-commerce applications [18].

*Keywords: Certified delivery, cryptographic protocol, e-banking, fair exchange, layered specifications, non-repudiation, secure e-commerce, secure orders.*

# Table of Contents (for reviewer ease)



# List of Tables

# List of Figures



# List of Definitions

# 1 Introduction

Dispute resolution are key to modern commerce. Consider the most basic trade, a client submitting an order for some goods or service. Different disputes may arise between the parties, e.g. regarding the quality of the goods or the time of delivery. Disputes may result from intentional attempts to cheat by either client or server (provider, bank, etc.), or from unintentional delivery problems.

In simple trades, there is exchange of goods, or of goods for payment. In such cases, parties can use a *trusted third party*, ensuring fair exchange, and thereby preventing disputes. However, often the server is *obliged* to provide the service or goods, upon receipt of appropriate order. In such cases, the server should *compensate* the client if it fails to provide the goods or services. In these (common) cases, *dispute resolution* is the only viable option. Modern commerce employs multiple mechanisms to ensure efficient and fair *dispute resolution*, ranging from signed orders and receipts to arbiters, notaries and courts. The efficiency and security of dispute resolution are critical.

Clearly, disputes are critical also for digital transactions and electronic commerce. There are many works on preventing and resolving disputes for electronic orders and goods; see Section 6 (related works). In particular, provably-secure fair-exchange protocols ensure that either both parties receive appropriate content (e.g. goods, payment, or contract), or neither party receive content.

However, fair-exchange cannot *force* the server to provide the goods or service, as required by many applications, e.g. e-banking and e-brokerage. Such applications require dispute resolution mechanisms, based on *evidences*, as for non-electronic transactions. Indeed, the provision of such evidences is one of the main goals of digital signatures. There are many designs and standards for producing (digitally signed) evidences for electronic transactions; see Section 6.

However, existing works on production of evidences for electronic orders, do not include rigorous specifications and proofs of security. Furthermore, existing works do not present an automated dispute resolution process. Indeed, current e-commerce systems depend on manual resolution - or simply on customers accepting the records of the service providers (e.g., broker, bank, clearing house). This is problematic, especially since communication systems are subject to failures, and computer systems are subject to attacks.

In this work, we present precise specifications and a provably-secure protocol for secure orders, including a fully automated process for dispute resolution, between a provider of digital goods and services, and its customers. Our design is efficient, practical and modular, with clear interfaces to applications and lower layers.

Our protocol is quite simple; however, the definition of appropriate, flexible, extensible yet well-defined *specifications* is non-trivial. The specifications, which we consider as the main challenge, allow resolving of disputes involving the timely receipt of orders and goods, due to communication failures and malicious faults, as well as disputes on the fitness of the goods to the order. This includes malicious buyers, which claim that an order was never placed, or that goods were not received (in time), as well as malicious sellers, which provide inappropriate or late good or services.

The design we present is flexible, and supports many types of e-commerce orders or transactions, allowing its use as underlying layer for secure commerce protocols. In our layered architecture design [18] each network principal employs secure e-commerce application layers, including *payment* layers, the *order* layer (in this paper) and an *attestation* layer, as a bottom layer; see Figure 1. Each layer provides *evidences* for the upper layers. For instance, the attestation layer, used by our order-layer protocol, issues to the sender evidences of message delivery (EOD) or of failure to submit message (EOFS) (see Table 2); these become part of the evidences, e.g. of goods delivery (EOGD), produced by the order layer.

Our specifications and design support arbitrary *trade validation function* for orders, provided as a 'black box' function. The trade validation function is defined as part of an *agreement* between the client and server. This allows dispute resolution for complex orders and goods, supporting many, diverse e-commerce applications. For example, the 'order' may specify a security, price and order type (e.g. 'buy 100 shares at 10$'); the 'goods' may be a signed receipt, specifying the results of the execution and current status of the account (with 100 more shares, and 1000$ less available funds).

**Contribution of this work.** Our main contribution is the specifications of the order layer, as a fully-automated, well-defined service, to e-commerce protocols and applications. We also present an efficient, practical and yet provably-secure order layer protocol. Our model and analysis are the first application of the adversarial layered specification framework [17]. A final contribution is our validation constructions, where every e-commerce layer defines its validation functions for automated dispute resolution, which is efficient and fair to all parties. An extended abstract version of this work appeared in [19].

**Organization.** The rest of the paper is organized as follows. In the next section we describe, informally, the lower layers (model); specifically, attestation layer, communication layer and digital signatures. In Section 3 we informally describe the order layer functionality and specifications and discuss the handling of typical fraud scenarios. Next, in the same section we show the order layer protocol implementation. We present the formal specifications in Section 4, and the analysis and poof of security of the protocol, in Section 5. In Section 6 we survey related work, and the last section concludes.

**Notation.** Throughout this document, we use **dot notation:** $\alpha.\beta$, to denote element $\beta$ of a record or tuple $\alpha$.

# 2 Lower layers: Attestation, Communication and Signatures

The order layer, see Figure 1 would employ lower layer services, such as attestation layer, for certified, attested, delivery; communication layer for non-attested communication, and signature scheme. We describe the services in the following sections.

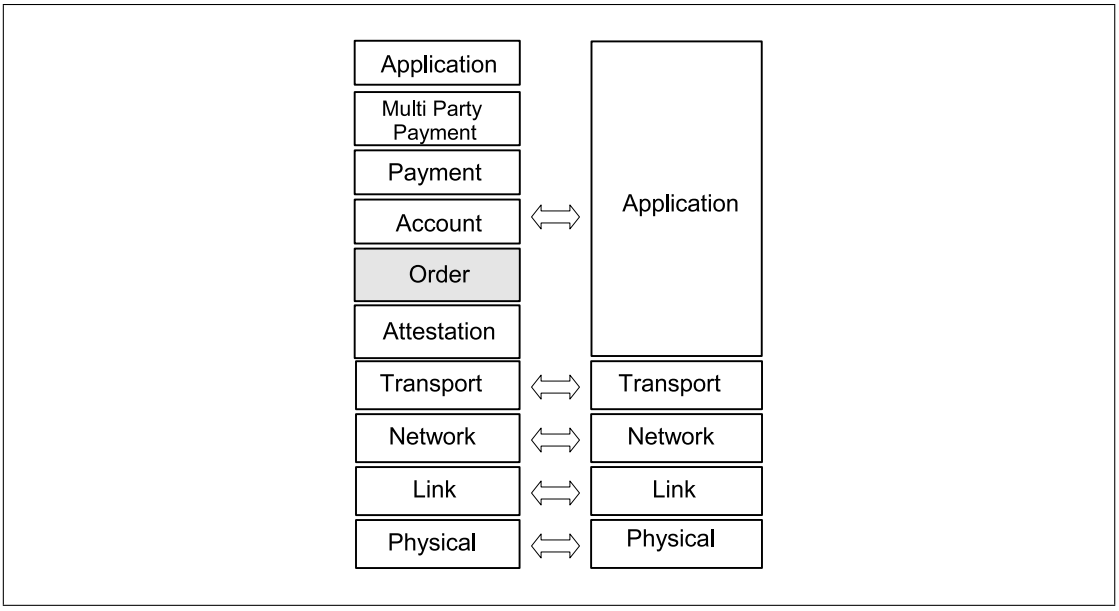


Figure 1: Secure e-commerce layers vs. Internet layers.

## 2.1 Attestation layer

The *Attestation* layer is the lowest secure e-commerce layer, see Figure 1. Attestation layer is based on top of a communication (transport) layer, such as, for example, TCP [28], TLS/SSL [7, 5, 12], and provides additional certification services. Attestation layer has three parties: client, server, and notary. The notary is a trusted third party (TTP) and acts as time-stamping and certification (attestation) provider.

**Attestation Agreement.** An attestation channel requires the parties to agree on an attestation agreement, specified in Table 1. The agreement, which is received from upper layer (in particular, we would generate such agreement from upper layer agreement, see lines 15-20 of Figure 4) specifies identities (by address and public key), for the sender, recipient and TTP. An

| **Agreement Field** | **Description** |
|---|---|
| *C=(C.addr, C.vk$_{att}$), S=(S.addr, S.vk$_{att}$), N=(N.addr, N.vk$_{att}$)* | The identities of the principals participating in the agreement; client (C), server (S) and notary (N), respectively. Principal's identity is an *(addr,vk$_{att}$)* tuple, of principal's address and public validation key. |

Table 1: Attestation agreement.

| **Evidence Field** | **Description** |
|---|---|
| *type* | Evidences of origin, delivery and failed submission, *EOO, EOD, EOFS*, respectively. |
| *ctime* | Evidence creation time. |
| *msg* | The message sent. |
| $\sigma$ | Signature over evidence fields. |

Table 2: Attestation evidence structure.

*attestation evidence* structure, as in Table 2, is a time-stamped and signed statement, regarding

the delivery of messages. The attestation evidence $e$ is containing evidence type, $e.type$, the message $e.msg$ that the evidence refers to, an evidence creation time $e.ctime$, and signature $e.\sigma$. There are three types of evidence provided by the attestation layer:

**Evidence of Delivery (EOD),** is an evidence for the message sender, that the intended message recipient received the message (during given time interval). EOD is signed by the recipient (and used by the sender).

**Evidence of Origin (EOO),** is an evidence that the message originated from the claimed sender (during given time interval). The EOO evidence is signed by the sender (and used by the recipient).

**Evidence of Failure and Submission (EOFS),** allows the sender to prove sending the message in question (during given time interval), even if the message was not received due to communication failures, or if the recipient failed to acknowledge receiving it. The EOFS is signed by the notary (and used by the sender).

**Validation.** The validation *Validate(AttAgr,e)* efficient predicate returns whether the evidence $e$ (Table 2) is valid, under the attestation agreement *AttAgr*. The validation functionality is not related to any particular instance of attestation module, and could be invoked by any third party, which had obtained the attested communication agreement and the evidence in question.

**Attestation interface.** The interface between payment and attestation layers is described in Table 3, and consists of initialization interface and an interface to send and receive message along with their respective evidences.

| Method | Direction | Description |
|---|---|---|
| *Init($1^k$,r,addr)* | in | Initializes layer, with security parameter, $1^k$, randomness $r$ |
| *InitResult(vk,$\Delta_{att}$)* | out | and principal's address *addr*. Returns generated validation key *vk* of the initializer and layer's delay bound $\Delta_{att}$. |
| *OpenChannel(AttAgr,$\rho$)* | in | Establishes an attested channel for the role $\rho \in \{$'S', 'C', 'N'$\}$, client, server and notary, respectively. |
| *OpenChannelResult(success)* | out | Notifies the principal on attestation channel establishment. |
| *CloseChannel()* | in | Closes an attested communication channel. |
| *Send(AttAgr,m)* | in | Sends a message $m$ on an open channel. |
| *SendResult(AttAgr,e)* | out | Returns an attestation evidence, Table 2, for previously sent message on attestation channel *AttAgr*, or value of *CommFail*. |
| *Receive(AttAgr,e)* | out | Delivery of evidence of origin $e$, Table 2, which also includes the message, *e.msg*, over an attestation channel identified by *AttAgr* agreement. Similarly to *SendResult* also used for channel failure *CommFail* notification. |

Table 3: Attestation layer interface.

**Initialization.** Attestation initialization is two phased, where in *'Init'*, the attestation layer generates secret and validation key pair, keeps the secret key and in *'InitResult'* returns the validation key. In *'OpenChannel'* a certified delivery channel is established with the party specified by the agreement supplied as an input. The established channel is uni-directional, with one party acting as a client, which is able to send messages, and thus obtain EOD or EOFS evidences, and the other party acting as a server, which receives EOOs. Communicating both ways would require dual-channel establishment, with each party being a client on one channel, and server on the other.

### 2.1.1 Attestation layer: Informal specifications

Intuitively, it should be hard to fake attestation evidences. More specifically, it should be hard to fake attestation evidences if parties are valid, i.e., proper initialization of attestation layer took place, and an attestation channel was correctly and successfully opened. Another requirement, we would make from an attestation layer, is that we would demand it provides to upper layer only evidences which had passed the attestation validation function.

Now let us highlight some of the specifications of the attestation layer; see the full specifications in Appendix A.7, or in [16].

**Correctness specifications.** Typically, our attestation layer correctness specifications are 'adversarial win' predicates on attestation interfaces. For example, the "fake-EOD" predicate $\mathcal{S}_{ATT}^{F-EOD}$ (Appendix A.7, Definition 43) specifies that if an EOD evidence $e$ was generated, and $e$ passes *Att.Validate(AttAgr,e)*, for an attestation agreement *AttAgr*, then for a honest server, which had opened an attestation channel with *AttAgr*, the predicate is true if no *'Att.Receive'* took place with the message specified by the EOD, in the time specified by the evidence. Similarly we define additional predicates for faking EOO and EOFS evidences.

The specifications are defined over execution $X$ of a protocol machine (see Appendix A.7). For such execution,

$\mathcal{S}_{\boldsymbol{ATT}}^{\boldsymbol{I\text{-}Recv}}(X)$,$\mathcal{S}_{\boldsymbol{ATT}}^{\boldsymbol{I\text{-}Send}}(X)$ **(Appendix A.7,Definition 40–41)** all evidences delivered by attestation layer of non-adversarial attestation parties, pass attestation validation function.

$\mathcal{S}_{\boldsymbol{ATT}}^{\boldsymbol{F-EOO}}(X)$,$\mathcal{S}_{\boldsymbol{ATT}}^{\boldsymbol{F-EOD}}(X)$,$\mathcal{S}_{\boldsymbol{ATT}}^{\boldsymbol{F-EOFS}}(X)$ **(Appendix A.7,Definition 42–44)** fake attestation evidences (adversarial win).

**Liveness specifications.** The liveness conditions are as follows: If both client and notary are valid parties, and communication link between the client and notary is sustained, then client would obtain, within bounded time, an evidence for a sent message. Furthermore, additional condition specifies, that if the server is also a valid party, and communication channel is sustained, at least with the notary, for both parties, then client would receive an EOD for sent message.

The following specifications are over lower layers interfaces events, so the specification would not be bound to any specific attestation layer protocol implementation,

$\mathcal{S}_{\boldsymbol{ATT}}^{\boldsymbol{INIT}}(X)$ **(Appendix A.7,Definition 35)** attestation parties share uniform delay bounds, and for properly initialized parties an attestation channel could be established.

$\mathcal{S}_{\boldsymbol{ATT}}^{\boldsymbol{LINK}}(X)$ **(Appendix A.7,Definition 39)** if no attestation failure was indicated then there are evidence for sent messages, and if pairwise links are sustained (at least to notary) between non-adversarial attestation parties, sent messages are delivered with respective evidences.

## 2.2 Communication layer

We can use any basic communication mechanism for direct communication between the notary and the client/server. We only need several methods: *'Comm.Init'* which receives protocol machine's address *addr* and returns the communication delay bound $\Delta_{comm}$; *'Comm.Send'* and *'Comm.Receive'*, both with ($\rho$,m) arguments, for sending or respectively receiving a message $m$ to or from a party $\rho$. We expect the *'Comm.Send'* to have a result value of `true` or $CommFail$.

Summarizing we very shortly review communication layer specifications, an adversarial environment is to uphold for $X$, protocol execution,

$\mathcal{S}_{\boldsymbol{COMM}}^{\boldsymbol{LINK}}(X)$ **(Appendix A.5, Definition 26)** specifies that if there we no communication failure indication between two principals then sent messages are delivered.

$\mathcal{S}^{INIT}_{COMM}(X)$ **(Appendix A.5, Definition 27)** specifies that communicating parties initialization is bounded in time, and all parties share communication timeout delay.

### 2.3 Signature Scheme

We adapt common digital signature scheme definition [14], where ($\mathcal{DS}.\pi$,$\mathcal{DS}.\mathsf{Verify}$) pair, is a protocol $\mathcal{DS}.\pi$, which includes two interfaces $\mathcal{DS}.\mathsf{Gen}, \mathcal{DS}.\mathsf{Sign}$, key and signature generation, respectively (and their respective output interfaces), and $\mathcal{DS}.\mathsf{Verify}$ is poly-time verification algorithm (we would sometimes omit the $\mathcal{DS}$ prefix for briefness, when usage is clear from context). While full definition and specifications are provided in Appendix A.6, we would only mention that $\mathcal{S}^{Sound}_{\mathcal{DS}}(X)$ predicate on execution $X \in \mathbb{X}$ means that generated signatures pass verification and $\mathcal{S}_{\mathcal{DS}}.AW(X)$ defines adversarial win against the signature scheme in an execution.

## 3 Order Layer: Overview and Protocol

### 3.1 Informal description and overview

There are various reasons to create a layered model for e-commerce, varying from modularity and contemporary software architectural reasons to ease of implementation and formal verification. We begin by describing, in brief, the functionality of an order layer, built on top of communication and attestation layers. Later sections would include more detailed scenarios, implementation and analysis.

An order layer transaction involves three principals, an order client, server, and a notary which is a trusted party. The latter may not be necessary in an optimistic scenario, where parties are honest, and no communication failures occur. Furthermore, we assume, for simplicity, the same notary is the one to provide attestation services for sent messages.

To participate in an order layer transaction, the client and server must agree on identities, public validation keys and roles. In addition the parties should agree on an efficient trade validation function, *ValidateTrade*, for mapping orders and the corresponding goods to several return values, indicating whether the order is valid, or whether the goods match the order.

An order layer transaction begins with an order client issuing an order message to the server, as shown in Figure 2.

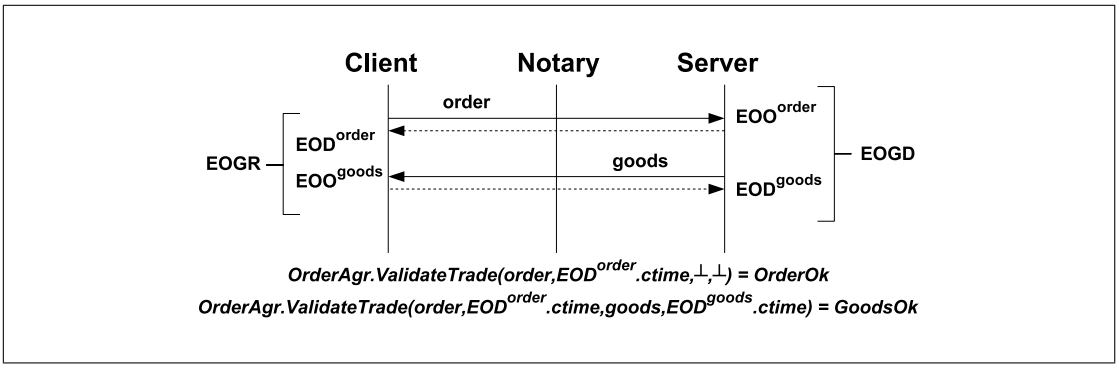


Figure 2: Order flow without faults. Client places a valid order, receives valid goods in return, and obtains *evidence of goods and receipt* (EOGR). The EOGR is formed from *evidence of delivery* for the order, as a proof of order placement and *evidence of origin* for the goods, as a proof of goods source. In a similar way the server obtains *evidence of goods delivery* (EOGD) for the vended goods.

The client obtains an evidence of delivery for the placed order and waits for the goods to arrive. When goods arrive, their validity is checked by the *ValidateTrade* function, and the order transaction terminates, with the client obtaining an *evidence of goods and receipt* (EOGR), which constitutes of the order agreement, order placement evidence ($EOD^{\texttt{order}}$ in the figure), and evidence of goods origin by the server ($EOO^{\texttt{goods}}$). The validity of the former goods is assured in the context of the *OrderAgr* agreement signed between the client and the server.

We present simple and efficient evidence arbitration in Section 3.3, Figure 12. For example, when aforementioned EOGR evidence would reach an arbiter, the arbiter would invoke the evidence validation function. The evidence validation function would check the validity of the order and goods by the same *ValidateTrade* function from the order agreement, and validate the EOD and EOO evidences, by deriving from the order agreement, the lower-layer (attestation) agreement between the client and the server, and invoking lower layer validation function for the aforementioned delivery and origin evidences, attaining full and automatic resolution of EOGR validity.

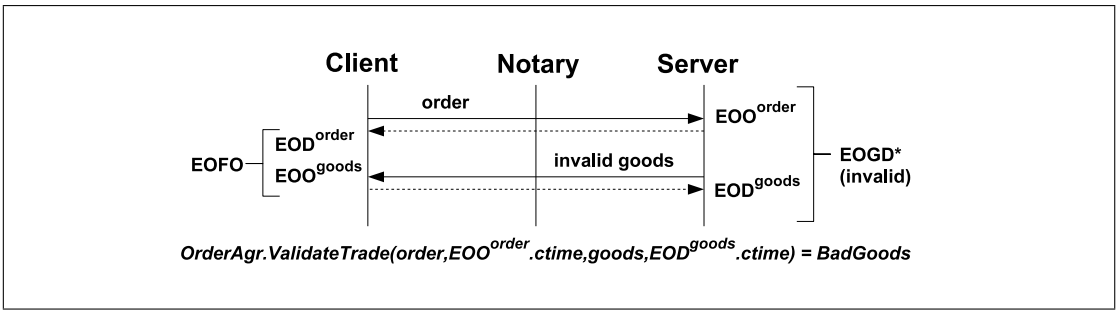


Figure 3: Delivery of invalid goods. When invalid goods are received from the server by the client, the client issues an *evidence of failed order* (EOFO) to upper e-commerce layer. While the server could also issue an *evidence of goods delivery* (EOGD) to upper layer, it would be considered invalid (marked with asterisk), as arbiter's *ValidateTrade* check, according to the order agreement between the client and the server, would fail for such goods, and thus the entire EOGD would fail order layer *Validate* check.

We show additional scenario in Figure 3, where a faulty or malicious server issues unsuitable goods. At client side, validation of such goods, along with the corresponding order would fail, and client would obtain an *evidence of failed order* (EOFO), which would be made of the evidence of placed order delivery and evidence of origin for the (unsuitable) goods. Later, such evidence would be considered valid by an arbiter, for valid order EOD and valid goods EOO, as the arbiter would invoke the same *ValidateTrade* function, defined by the order agreement between the parties. Similarly, while the faulty server could have issued an *evidence of goods delivery* (EOGD) to the upper layer, for the invalid goods, such evidence would be deemed invalid by an arbiter, again by way of invocation of the *ValidateTrade* function from the order layer agreement signed between the principals.

Consider the following, typical, e-commerce disputes, and how they are handled in our scheme.

**Disputes on Delivery or Quality.** Client claims an order was placed, server denies. This claim could be easily resolved as the claiming principal would have an evidence for his order, either an evidence of failed order (EOFO) or evidence of goods and receipt (EOGR), that also includes the order evidence. Additional typical claim is that invalid order or goods were received from the other principal. This claim is easily verified, as verification of former evidences includes invocation of *ValidateTrade* function to match order to goods.

**Disputes on order placement.** Client claims order was not placed. Server should be able to refute this, by presenting EOGD or EOFGD evidences. In both cases the evidences includes the order, and an evidence of origin (EOO) for that order, which means the order is approved and non-repudiable.

We formally define order layer specifications in Section 4.2. Informally, we expect an order layer protocol to uphold a range of conditions, on the order layer interfaces, such as the following non-comprehensive list:

**Valid evidences.** Valid protocol parties should always issue valid order layer evidences to upper layers, i.e., evidences that pass *Order.Validate* (see Section 4.2, Definition 6).

**Fair evidences.** If client had obtained valid EOGR, goods evidence for an order, and server and notary are valid parties, then the server had surely obtained evidence of goods delivery (or evidence failed goods delivery - as our protocol would support recovery). On the opposite, if the server presents valid evidence of goods delivery, EOGD, then the client had obtained EOGR, evidence of goods and receipt (see Section 4.2, Definition 12, 13).

**Attainable evidences.** Valid and honest server would always receive evidence for issued goods, if there is reliable communication between the server and the notary (see Section 4.2, Definition 16). Even if there is a communication failure between server and notary, the server would still issue an evidence of client's order (EOCO) to upper layer (see Definition 15). As a motivation for the later scenario, consider a honest stock broker which had performed stock exchange operation, however which is unable to send acknowledgment to the client, due to a communication failure. The broker, however, is required to support its taken action based on client's order, in front of an arbiter, with an evidence of client order.

**Negligible probability of successful attack.** We expect the probability of the protocol run being incorrect to be negligible in the security parameter. Basically, it should be hard to forge the order layer evidences (Section 4.2, Definitions 7–11). For example, the predicate describing fake EOGR (Definition 7) considers an execution where a valid EOGR evidence is output, however, there was no vend of goods specified by that EOGR at a valid (and honest) server.

**Non-Notarized Communication Failures.** Recovery from non-notarized communication failures is possible for honest parties. Consider the case where a client (or similarly a server) issuing an order request, receives in return a communication failure (instead of an EOD). The client could not possibly know whether the channel had failed, before the request had been delivered (and server had obtained an EOO), or afterwards, and the failure had prevented the client from receiving an EOD (or EOFS). For recovery from the former, honest parties could include with the next order or goods response the EOO evidences received from other parties. Thus, the state of an order transaction, that had experience communication failure, between honest parties, would be unknown only until the next order transaction successfully completes.

## 3.2 Agreements, Evidences and Interface

The order layer encapsulates operations ("orders") related to funds, e-brokerage, or digital goods and services. The layer provides the service for placing an order for goods or services by a client, and validating that the server returned order result adhere to an order agreement between the principals.

### 3.2.1 Order Agreement and *ValidateTrade* function

We define an order agreement between trading parties as specified in Table 4. An order agreement is used to generate attestation agreements between order client and server. The order agreement also specifies a trade validation function, *ValidateTrade(order,$t_o$,goods,$t_g$)*, which receives an *order* and order creation time $t_o \in \mathbb{R}$, and respectively, *goods* and goods vend time $t_g \in \mathbb{R}$. The *ValidateTrade* function provides versatility of trade by allowing the client and the server to agree on appropriate goods and services similarly to traditional trade agreements. The function should have *BadOrder, BadGoods, OrderOk, GoodsOk* return values. The *BadOrder* return value is issued for an order which is invalid under the agreement, regardless of the value of goods. The second, *BadGoods* return value, is issued for goods which do not match the order, the *OrderOk* returned for valid order, without goods; and the *GoodsOk* status is returned when the corresponding goods match the order, in the context of the order agreement.

Since good provision is not always immediate, the order agreement additionally includes $\Omega_{goods}$, which is a bound on goods delivery time, taking into account the communication delays and time it make take the server to generate the goods.

In addition to efficiency requirement from the *ValidateTrade* function, we somewhat simplify the requirements on the relation of goods to orders and for ease of analysis have need for *ValidateTrade* function to be monotonic in its two time coordinates, up to the goods delivery bound specified by the order agreement, i.e.,

1. *ValidateTrade(order,* $t_o$*,* $\perp$*,* $\perp$*)* = *OrderOk* $\implies$
   *ValidateTrade(order,* $t'_o$*,* $\perp$*,* $\perp$*)* = *OrderOk*, $\forall t_o, t'_o \in \mathbb{R}$, s.t., $t_o \leq t'_o$
   (valid orders continue to be always valid),
2. *ValidateTrade(order,* $t_o$*, goods,* $t_g$*)* = *GoodsOk* $\implies$
   *ValidateTrade(order,* $t_o$*, goods,* $t'_g$*)* = *GoodsOk*, $\forall t_o, t_g, t'_g \in \mathbb{R}$, s.t.,
   $(t_g \leq t'_g \leq t_o + \Omega_{goods})$
   (once valid goods continue to be valid, unless goods claimed time is past agreement bound),
3. *ValidateTrade(order,* $t_o$*,* $\perp$*,* $\perp$*)* = *OrderOk* $\wedge$ $((t_g < t_o) \vee (t_g > t_o + \Omega_{goods})) \implies$
   *ValidateTrade(order,* $t_o$*, goods,* $t_g$*)* = *BadGoods*, $\forall t_o, t_g \in \mathbb{R}$
   (if goods claimed time is before order time, or past agreement bound, goods are considered invalid),

| **Agreement Field** | **Description** |
|---|---|
| *ValidateTrade(order,*$t_{order}$*, goods,*$t_{goods}$*)* returns *status*; | Trade validation function. Validates that issued *goods* at time $t_{goods}$ match an *order* created at $t_{order}$ time. The return value is *status* $\in$ { *OrderOk, BadOrder, GoodsOk, BadGoods* } |
| $\Omega_{goods}$ | Bound on goods issue time as considered by previous *ValidateTrade*. After $\Omega_{goods}$, no goods considered valid. |
| *C.addr, C.*$vk_{att}$*, C.*$vk_{order}$*, S.addr, S.*$vk_{att}$*, S.*$vk_{order}$*, N.addr, N.*$vk_{att}$*, N.*$vk_{order}$ | Order layer participating principals. A client (C), server (S) and a notary (N), each as a $(addr, vk_{att}, vk_{order})$ tuple of address, and pair of validation keys, respectively. |

Table 4: Order agreement.

### 3.2.2 Order interface

The interface between the application and order layer, Table 5, defines the initialization, ordering goods or services, and validation of order results. In the first, *Init* phase, each order layer machine establishes its own identity, as returned by attestation layer. Using this information a principal may establish order agreements with other network principals.

| Method | Direction | Description |
|---|---|---|
| *Init*($1^k$,$\rho$,$r$,*addr*) | in | Initializes the order layer, with security parameter $1^k$, address *addr*, role $\rho \in \{$'*S*', '*C*', '*N*'$\}$ and randomness $r$. |
| *InitResult*(*vk*,$\Delta_{order}$) | out | Returns initializer's validation key(s) *vk*, and a $\Delta_{order}$ bound on evidence return time. |
| *OpenChannel*(*OrderAgr*,$\rho$) | in | Opens an order channel with the principals specified by *OrderAgr* agreement using role $\rho$. |
| *OpenChannelResult*(*status*) | out | Notifies the application layer of the order channel establishment success. |
| *CloseChannel*() | in | Closes an order channel. |
| *OrderResult*(*OrderAgr*,*e*) | out | Returns *CommFail* or order evidence *e* result, for an order on *OrderAgr* agreement open channel. |
| ***Client*** | | |
| *Order*(*OrderAgr*,*order*) | in | Instructs the order layer to issue an order, described by *order*, over an order channel established over an *OrderAgr* agreement. |
| ***Server*** | | |
| *VendRequest*(*OrderAgr*,*order*) | out | Instructs the application layer to issue goods, described by *order*, and implicitly by the order agreement, in the order context. |
| *VendRequestResult*(*OrderAgr*,*goods*) | in | Returns goods vended by upper layer. |

Table 5: Order layer interface.

When an order channel is established, order transactions are invoked with *Order* event, supplying client specified order information which could define funds transfer options, e-brokerage conduct, digital goods request, and possibly other relevant information (e.g., original merchant offer for digital goods). We then expect an *OrderResult* event within finite time, as governed by $\Delta_{order}$ returned from order layer initialization interface, and goods delivery bound, specified in the order agreement.

On the server side, we assume an application (or upper) level functionality to issue goods or services, using *VendRequest* interface. The goods and services are issued in the context of the order agreement specified for the open order channel, and are verifiable by order agreement's *ValidateTrade*.

| Order Evidence Field | Description |
|---|---|
| *type* | Evidences of placed order, failed order, goods delivery, failed goods delivery, or client order *EOGR, EOFO, EOGD, EOFGD, EOCO*, respectively. |
| *ctime* | Evidence generation time. |
| *order* | The order specified by the evidence. |
| *goods* | The corresponding goods. |
| $\sigma$ | Order layer proof, for the above evidence. |

Table 6: Order layer evidences structure.

### 3.2.3 Order Layer Notary

For simplicity, we assume that the parties agree on a single notary, which is trusted to resolve disputes both regarding the contents of orders and goods (order layer disputes), as well as disputes regarding the timely delivery of orders and goods (attestation layer disputes). This implies that the order layer protocol does not provide evidences for failure of the order-layer notary itself to perform its function, e.g. to provide timely evidences. This allows our protocol to use the order layer notary also as the attestation layer notary, and to use attestation layer

only between client and server, and ordinary communication between notary and the client and the server. We believe it is straightforward, although messy, to extend the specification and protocol, to allow separate notary for the attestation layer, providing evidences of failures of the order layer notary.

#### 3.2.4 Order Evidences

The structure of order layer evidences is shown in Table 6. The evidences include various trade related evidences, as specified below, and evidence's $e$ proof $e.\sigma$ consists of lower, attestation layer evidences.

**Evidence of Goods and Receipt (EOGR),** is client's (buyer) proof that the corresponding order had reached the server (seller), and that goods had been obtained for the order.

**Evidence of Goods Delivery (EOGD),** is server's (seller) proof that the goods issued for client's order had reached the client.

**Evidence of Failed Order (EOFO),** is client's proof that the order process had failed. It could either be the case that the order message itself was not acknowledged by the server, or if it was acknowledged but the server did not issue goods.

**Evidence of Failed Goods Delivery (EOFGD),** is server's proof that the goods delivery process had failed, since the goods message was not acknowledged by the client.

**Evidence of Client Order (EOCO),** when server goods delivery process results in a communication failure, and goods delivery status is unknown, the server's order layer returns EOCO as a proof, that an order was indeed placed. It is typical desire of a service provider to prove (to upper layer, and consequently to an arbiter) that an action was taken due to client order. The EOCO evidence is issued in the case an order was placed but the corresponding service or goods could not be delivered due to server's own communication failure (thus even EOFS is not obtainable from attestation layer on server side), where the server should at least present a proof of client's order.

### 3.3 The OL-protocol

We present the order layer OL-protocol in Figures 4–7, containing the implementation for the initialization (common to all participants), server, notary and client, respectively. The OL-protocol employs signature scheme $\mathcal{DS}$, defined, with specifications, in Appendix A.6. In initialization, Figure 4, the attested channels are opened between each pair of roles specified in the order agreement (Table 4), namely, client, server and notary. Next, in Figures 5-7, the protocol implementation describes how each party acts upon receiving a messages, over regular and attested channels.

The protocol interaction is of a request-response form. As shown in in Figure 2, in a faultless execution, the client sends a valid order and receives valid goods in return; both client and server obtain order layer evidences for the process, EOGR (evidence of goods and receipt) and EOGD (evidence of goods delivery), respectively. For a dishonest server issuing invalid goods, as shown in Figure 3, the client would obtain an EOFO (evidence of failed order), and while the dishonest server could try and provide an EOGD to an upper layer, it would be disqualified by the order layer arbitration function. On the other hand, if valid goods delivery fails, as in Figure 8, the server would obtain EOFGD (evidence of failed goods delivery), and if order delivery fails the client would obtain EOFO, as shown in Figure 9.

Additionally, the protocol is optimistic, from order layer outlook. The client is placing the order at the server and expects to receive the goods. If goods are not received (possibly due

```
on Order.Init(1^k,ρ,r,addr) :
    Δ_comm = Comm.Init(addr)
    (vk_att,Δ_att) = Att.Init(1^k,r,addr)
    Δ_order = 2 · Δ_att + 4 · Δ_comm
    if ρ = 'N'
        vk_order = DS.Gen(1^k,r)
    else
        vk_order = ⊥
    Order.InitResult((vk_att,vk_order),Δ_order)
on Order.OpenChannel(OrderAgr', ρ') :
    OrderAgr = OrderAgr'
    success = (ρ = ρ') ∧ (OrderAgr.ρ = (addr,vk_att,vk_order)) // check identity same as in Init
    if ρ ∈ {'C','N'}
        //prepare two attestation agreements, one for each direction
        AttAgr^{C,S} = OrderAgr.((C.addr,C.vk_att),(S.addr,S.vk_att),(N.addr,N.vk_att))
        AttAgr^{S,C} = OrderAgr.((S.addr,S.vk_att),(C.addr,C.vk_att),(N.addr,N.vk_att))
    if ρ = 'S'
        //prepare same attestation agreements, reversing roles
        AttAgr^{C,S} = OrderAgr.((S.addr,S.vk_att),(C.addr,C.vk_att),(N.addr,N.vk_att))
        AttAgr^{S,C} = OrderAgr.((C.addr,C.vk_att),(S.addr,S.vk_att),(N.addr,N.vk_att))
    if ρ ∈ {'C','S'}
        success ∧= Att.OpenChannel(AttAgr^{C,S},'C') ∧
            ∧ Att.OpenChannel(AttAgr^{S,C},'S')
    else // ρ = 'N'
        success ∧= Att.OpenChannel(AttAgr^{C,S},'N') ∧
            ∧ Att.OpenChannel(AttAgr^{S,C},'N')
    Order.OpenChannelResult(success)
```

Figure 4: Initialization in OL-protocol for order layer parties, $\rho \in \{$'S', 'C', 'N'$\}$, saves digital signature $\mathcal{DS}$ instantiation, $addr, \Delta_{att}, \Delta_{comm}$, $OrderAgr$ and derived attestation agreements as principal's state.

```
on Att.Receive(AttAgr^{C,S},oe={EOO,ctime,order,σ}) :
    if ValidateTrade(oe.msg,oe.ctime,⊥,⊥) = OrderOk    // good order
        goods = VendRequest(OrderAgr,order)    // application provides goods
        ge = Send(AttAgr^{S,C},goods)    // get evidence for goods delivery
        if (ge.type = EOD)
            OrderResult(OrderAgr,{EOGD,ge.ctime,oe.msg,ge.msg,{oe,ge}})
        if (ge.type = EOFS)
            OrderResult(OrderAgr,{EOFGD,ge.ctime,oe.msg,ge.msg,{oe,ge}})
        if (ge=CommFail)
            OrderResult(OrderAgr,{EOCO,oe.ctime,oe.msg,⊥,{oe,⊥}})
on Comm.Receive(N,(AttAgr^{C,S},oe={EOD,ctime,order,σ})) :
    if Att.Validate(AttAgr^{C,S},oe) ∧ ValidateTrade(oe.msg,oe.ctime,⊥,⊥) = OrderOk
        Comm.Send(N,(AttAgr^{S,C},ge))
```

Figure 5: Order layer OL-protocol for the Server.

```
1:  on Comm.Receive(C,(AttAgr^{C,S},oe={EOD,ctime,order,σ})) :
2:      if Att.Validate(AttAgr^{C,S},oe) ∧ ValidateTrade(oe.msg,oe.ctime,⊥,⊥) = OrderOk
3:          set timer for 2 · Δ_comm
4:          Comm.Send(S,(AttAgr^{C,S},oe))
5:  on Comm.Receive(S,(AttAgr^{S,C},ge={EOFS,ctime,goods,σ})) :
6:      if timer set ∧ Att.Validate(AttAgr^{S,C},ge)
7:          cancel timer
8:          Comm.Send(C,(AttAgr^{S,C},ge))
9:  on Comm.Receive(S,(AttAgr^{S,C},ge={EOD,ctime,goods,σ})) :
10:     if timer set ∧ Att.Validate(AttAgr^{S,C},ge)
11:         if ValidateTrade(oe.msg,oe.ctime,ge.msg,ge.ctime) = GoodsOk
12:             cancel timer    // client is cheating
13: on timer :
14:     Comm.Send(C,(OrderAgr,DS.Sign({EOFO,currtime(),order,⊥})))
```

Figure 6: Order layer OL-protocol for the Notary.

```
1:  on Order(OrderAgr,order) :
2:      if ValidateTrade(order,currtime(),⊥,⊥) = OrderOk
3:          oe = Send(AttAgr^{C,S},order)
4:          if (oe = EOFS) OrderResult(OrderAgr,{EOFO,oe.ctime,oe.msg,⊥,{oe,⊥}})
5:          if (oe = EOD) set timer_server for Δ_att + OrderAgr.Ω_goods
6:          if (oe=CommFail) OrderResult(OrderAgr,CommFail)
7:  on timer_server :
8:      set timer_notary for 4 · Δ_comm    // goods were not received on time
9:      Comm.Send(N,(AttAgr^{C,S},oe))
10: on Comm.Receive(N,(OrderAgr,e={EOFO,ctime,order,⊥,σ})) :
11:     if Order.Validate(OrderAgr,e,'C')    // implicit check of notary's signature
12:         OrderResult(OrderAgr,e)
13: on Receive(AttAgr^{S,C},ge={EOO,ctime,goods,σ}) :
14: on Comm.Receive(N,(AttAgr^{S,C},ge={EOFS,ctime,goods,σ})) :
15:     if Att.Validate(AttAgr^{S,C},ge)    // may not be true only for Comm.Receive (line 14)
16:         cancel timer_server
17:         if ValidateTrade(oe.msg,oe.ctime,ge.msg,ge.ctime) = GoodsOk
18:             OrderResult(OrderAgr,{EOGR,ge.ctime,oe.msg,ge.msg,{oe,ge}})
19:         else OrderResult(OrderAgr,{EOFO,ge.ctime,oe.msg,ge.msg,{oe,ge}})
20: on timer_notary :
21:     OrderResult(OrderAgr,CommFail)
```

Figure 7: Order layer OL-protocol for the Client. The protocol terminates for the *order* and cancels timers after first *OrderResult*.

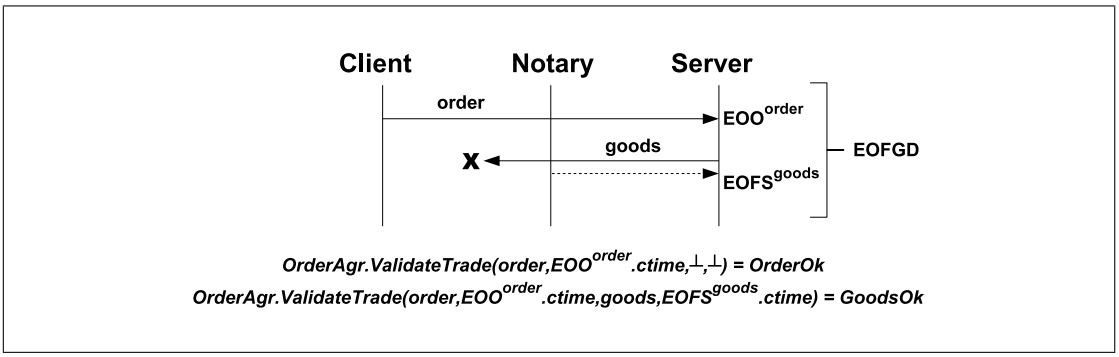


Figure 8: Failed goods delivery flow. A valid order is received by the server, however, goods delivery fails with an *evidence of failure and submission* and thus *evidence of failed goods delivery* is issued by server's order layer to upper e-commerce layer.

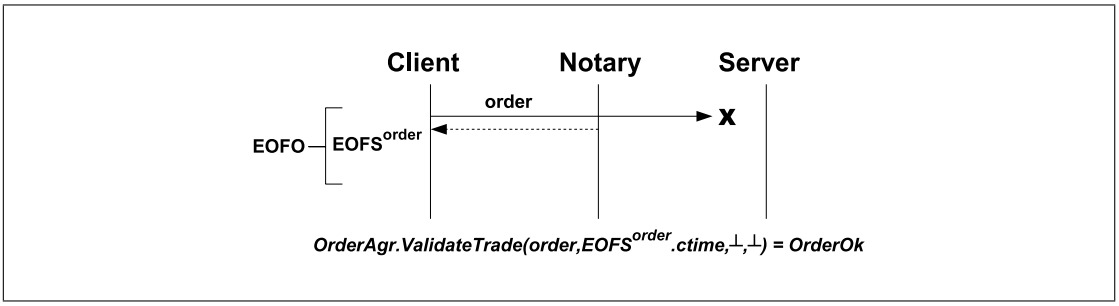


Figure 9: Failed order flow. The client obtains an *evidence of failure and submission* for the order message, from the attestation layer, and wraps the evidence as order layer's *evidence of failed order*. Whether the EOFO would be considered valid by arbitrating party depends on whether the order described by the failed message is valid, as checked by *ValidateTrade* function from order agreement between parties.

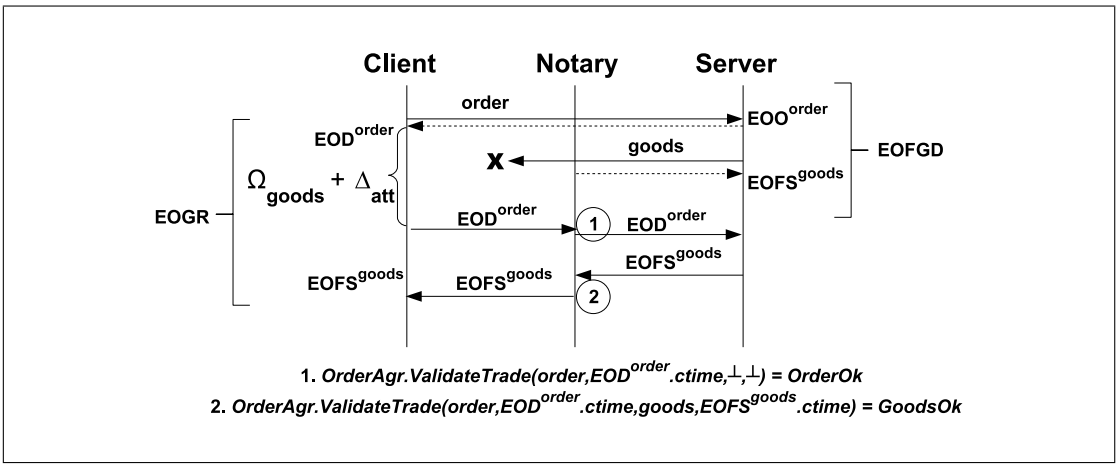


Figure 10: Recovery from transient client-notary (and server-client) communication failure. The server did issue goods for client order, and was issued an *evidence of failure and submission*. Overall, the server have obtained order layer *evidence of failed goods delivery*. When the server is later contacted by the notary, it forwards the evidence of failure and submission for the goods, which includes the goods message, to the notary. The notary, in turn, forwards it to the client. If goods are valid client issues *evidence of goods and receipt* (and evidence of failed order for invalid goods).

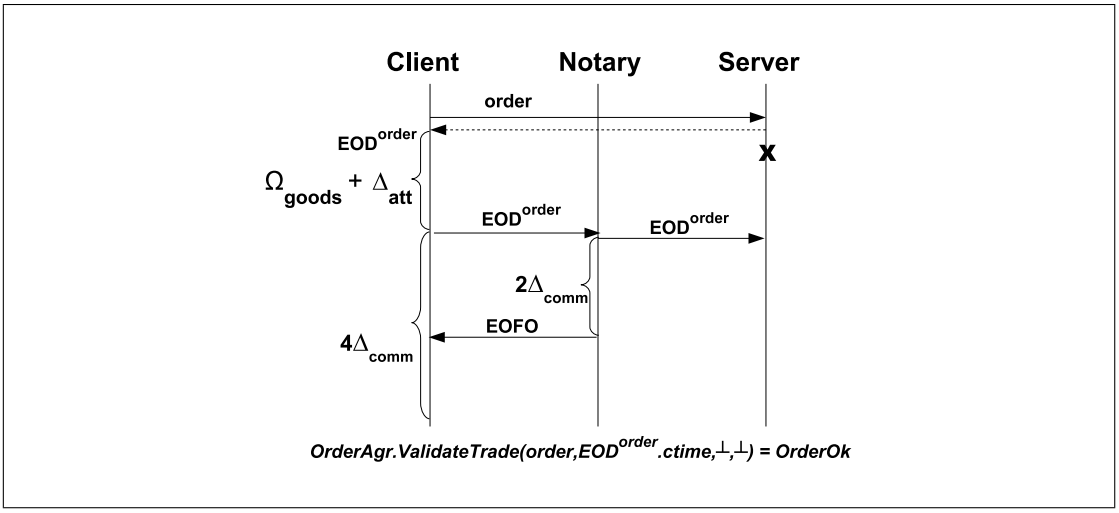


Figure 11: Failed order flow for a failed server. As in previous scenario, Figure 11, goods are not obtained by the client. An order is placed by the client, which retains an *evidence of delivery* for the order, however, no goods are received in return. After waiting $\Delta_{att}$ to allow server goods message to reach the client over the attested channel between the two, the evidence of delivery for order placement is forwarded to the notary over regular communication channel with delay $\Delta_{comm}$. If the order is valid, and yet goods are not sent by server to the notary, the notary is the one to issue an *evidence of failed order* to the client.

to client's own communication failure), order failure recovery process is initiated, as shown in Figures 10 and 11. In Figure 10, the recovery process involves sending an order EOD (evidence of delivery) to the notary, after the goods were not received by the client. The order content and order creation time (taken from the EOD attestation evidence) would be validated, and if valid the order EOD would be forwarded to the server, which should return the EOFS goods evidence to the notary. After checking for goods fitness, the notary would forward the evidence back to the client. In the case client is cheating the server would provide the notary an EOD for (valid) goods, hence, notary would conclude the goods has been already delivered and would abort recovery process. Additional case, described in Figure 11, is when no valid goods are provided by the server, up to a pre-defined communication delay, and the notary would issue self-signed EOFO (evidence of failed order) to the client.

We now shortly describe the protocol implementation as shown in Figures 4–7. For conciseness the protocol is for a single, unique, *order*; extension for multiple orders is trivial.

**Initialization.** The initialization code, shown in Figure 4 is split two fold, initializing an OL-protocol machine (*'Order.Init'* event handler) and opening an order channel (*'Order.OpenChannel'* event handler). When protocol machine is initialized it initializes the lower layers, communication and attestation layer, receiving the machine address, communication and attestation delay bounds and attestation validation key, respectively. Next, if the machine is in the notary role, it generates additional pair of keys for signing and validating EOFO evidences. When opening an order channel, the protocol machine validates that the identity used in the order agreement is the correct one, generates two attestation agreements (one for each direction between order-client and order-server, as messages are sent two ways), and returns order channel open success indication as dependent on successful opening of the two attestation channels. After opening the two attestation channels, the OL-protocol machine is ready to participate in the order process.

**Server.** The server implementation, in Figure 5, handles two events: a received order event

(lines 1–10) and recovery for failed order (lines 11–13). The order processing is trivial: upon receiving an order EOO evidence from attestation layer, goods are requested from upper layer, and are sent to the client. The evidence for the sent goods (if available) together with the evidence for the order composes the order layer evidence the server provides to upper layer in *'OrderResult'*. For the recovery process, lines 11–13, the server is contacted by the notary over a regular, non-attested channel and presents it with a (valid) EOD of an order, meaning that the client claims it did not receive the issued goods - as could be if the server had obtained an EOFS for the goods message. When such request is received the server responds to the notary with the goods evidence for the order.

**Notary.** The order layer notary implementation is shown in Figure 6; notice that in addition, the notary machine also runs an attestation-layer notary. Notary's order layer functionality is to support the recovery process for failed orders, and to issue notary-signed EOFO if recovery fails and server is unable to provide the goods. Upon receiving an order EOD on line 1, the notary checks for evidence and order validity, and forward the message to the server, waiting for a response (lines 5, 9) or response timeout (line 13). The response could be goods EOFS, line 5, which is checked for validity and forwarded back to the client (thus completing recovery process), or goods EOD, line 9, the meaning of which is that the client is cheating (as server shows it did deliver goods). If the client is cheating the recovery process is aborted. On the other hand, if no evidence was received from the server, line 13, the notary signs an EOFO, effectively placing the blame for communication failure on the server.

**Client.** Implementation of OL-protocol for client side is shown in Figure 7. The order process begins on line 1, and progresses with obtaining an attestation evidence for a sent order. If the order message did reach the server (EOD was obtained) a recovery timer is set (recovery begins on line 7) and the client begins to wait for server's response (line 13), otherwise the order process terminates with EOFO if communication failure was notarized (EOFS was obtained for the sent order message), or with a plain communication failure. If server response is obtained on line 13, a corresponding *'OrderResult'*, which depends on the received goods fitness, is issued to upper layer. Otherwise, on timer wakeup, recovery process is initiated (as it could be the case that client was experiencing communication failures which prevented him from receiving the goods) and the order EOD which proves the server did receive the order is sent over a regular communication channel to the notary; additional timer is set for receiving notary's response (lines 7 and 9). If no notary response would be received by the time of second timer wakeup, communication failure would be declared (line 20), otherwise, it is expected that either the notary would provide an EOFO (line 10) as an evidence of server failure or would forward evidence of the goods sent by the server (line 13).

**Validation.** The validation functionality, *Validate(OrderAgr,e,ρ)*, Figure 12, is common to all parties. That is, an automatic dispute resolution system, or an arbiter, upon dispute, would instantiate the order layer, and supply the relevant order agreement along with the the protocol-specific order evidence $e$, and the checking role $\rho \in \{\text{'}C\text{'}, \text{'}S\text{'}\}$, which is used to decide the direction lower attestation agreement should be derived from the order agreement (a single order channel is built on two attestation channels, since delivery of messages is both ways). The order layer evidences are typically composed of pairs of relevant attestation evidences[1].

[1]The order layer validation algorithm specified in Figure 12 efficiently decides whether order layer evidences are valid, for the identities supplied in the order agreement. The identities are typically specified as communication addresses and public signature verification keys of the parties. However, its up to the upper layer or the arbiter to map the identities to real physical or legal entities, e.g., by means of X.509 certificates [1].

# 4 Specifications

For the protocol analysis we adopt a layered analysis, where having concrete and well-defined specifications for both the layer below the analyzed layer and the analyzed layer itself, we relate the two specifications and show that given an adversary which 'breaks' the analyzed layer specifications, we could use this adversary to either 'break' the lower-layer specifications, or 'break' signature scheme specifications in the analyzed layer. In Figure 13 we show the specifications we are to consider. The order layer specification, $\mathcal{S}_{ORDER}$ would be defined as set of requirement on the general interface of the layer, which particular implementation we have provided in previous Section 3.3, the lower layer specifications $\mathcal{S}_{ATT}$ and $\mathcal{S}_{COMM}$, respectively for attestation and communication layers, are also to be defined on layers interfaces, not be tied to any concrete implementation of the aforementioned layers.

The rest of this section is devoted for formalizing specification for order layer, first by defining some notation and general terms, then presenting the requirements, i.e., initialization, correctness and liveness specifications of the order layer.

## 4.1 Model, Notation and General terms

We provide the execution model formalism in Appendix A; the following paragraphs shortly summarize the general concepts, so the reader could immediately proceed to Section 4.2.

Single protocol executions result from the interaction of protocol machines with an adversary representing the rest of the world. A single protocol execution is defined by a sequence of *rounds*, where each round consists of one invocation of the adversary $\mathcal{A}$, and then one invocation of the protocol $\pi$. We model the adversary $\mathcal{A}$ as a function from the sequence of all outputs of the protocol so far, to the next input to the protocol. Then we consider multiple instances of the protocol, each running with its own state, to represent multiple processors (from set $\mathbb{P}$ of protocol machine processors identities) running the protocol. At each round $i$, the adversary $\mathcal{A}$ invokes a specific *instance* (processor) $p_i$ of the protocol. For shorthand, an event of protocol $\pi$ is a tuple $\xi = <p, \iota, v_\iota>$, where $p \in \mathbb{P}$ is processor identity, $\iota \in \pi.I_{IN} \cup \pi.I_{OUT}$ is protocol interface and $v_\iota \in \{0,1\}^*$, is the value of the interface.

We use $\mathbb{X}$ to denote the set of single protocol executions (of any $\pi, \mathcal{A}$ and randomness $R$). We use the shorthand notation $t(\xi)$ to refer to the time of event $\xi$ in an execution and use additional notation of $\xi \in X[t_0, t_1]$, to denote that event's occurrence time, $t(\xi)$, was in the interval $[t_0, t_1] \subset \mathbb{R}$, or just $\xi \in X[t_0]$ to denote specific, $t(\xi) = t_0$, event time.

## 4.2 Order layer specifications

### 4.2.1 Initialization specifications

We begin with defining bounded and proper initialization (similarly to attestation layer specifications (Appendix A.7) along with a *valid open* predicate, to specify parties that have correctly opened an order channel. However, we first assume, for simplicity, that a global $\Delta_{order}$ delay is shared by all initialized machines, as in the next predicate,

**Definition 1 (Uniform initialization $\mathcal{S}_{ORDER}^{INIT-U}$ predicate)** Predicate $\mathcal{S}_{ORDER}^{INIT-U}(X)$ is `true` for execution $X \in \mathbb{X}$, if $\exists \Delta_{order} \in \mathbb{R}$ s.t., for every event
$<p, \text{'Order.InitResult'}, ((vk_0, vk_1), \Delta)> \in X : \Delta = \Delta_{order}$, where $p \in \mathbb{P}$, $vk_0, vk_1 \in \{0,1\}^*$.

We also define a liveness predicate assuring initialization is bounded, for simplicity, with the previous $\Delta_{order}$ delay bound,

```
Order.Validate(OrderAgr,e,ρ):
 1:    if (DS.Verify(e,OrderAgr.N.vk_order) ∧ e.type=EOFO) return true
 2:    (oe,ge) = e.σ    // order and goods attestation evidences.
 3:    if ρ = 'C'
 4:       AttAgr^{C,S} = OrderAgr.((C.addr,C.vk_att),(S.addr,S.vk_att),(N.addr,N.vk_att))
 5:       AttAgr^{S,C} = OrderAgr.((S.addr,S.vk_att),(C.addr,C.vk_att),(N.addr,N.vk_att))
 6:    if ρ = 'S'
 7:       AttAgr^{C,S} = OrderAgr.((S.addr,S.vk_att),(C.addr,C.vk_att),(N.addr,N.vk_att))
 8:       AttAgr^{S,C} = OrderAgr.((C.addr,C.vk_att),(S.addr,S.vk_att),(N.addr,N.vk_att))
 9:    if ge≠ ⊥ // check evidence time validity
10:       if e.ctime ≠ ge.ctime return false
11:    else if e.ctime ≠ oe.ctime return false
-------------------------------------------------------------------------
       Client evidences:
-------------------------------------------------------------------------
12:    case e.type=EOFO:
13:       if oe.type = EOFS
14:          return Att.Validate(AttAgr^{C,S},oe) ∧ e.order = oe.msg ∧
15:             ∧ ValidateTrade(e.order,oe.ctime,⊥,⊥)=OrderOk
16:       if oe.type = EOD
17:          return Att.Validate(AttAgr^{C,S},oe) ∧ Att.Validate(AttAgr^{S,C},ge) ∧
18:             ∧ ValidateTrade(e.order,oe.ctime,e.goods,ge.ctime)=BadGoods ∧
19:             ∧ e.order = oe.msg ∧ ge.type = EOO|EOFS ∧ e.goods = ge.msg
-------------------------------------------------------------------------
20:    case e.type=EOGR:
21:       return Att.Validate(AttAgr^{C,S},oe) ∧ Att.Validate(AttAgr^{S,C},ge) ∧
22:          ∧ ValidateTrade(e.order,oe.ctime,e.goods,ge.ctime)=GoodsOk ∧
23:          ∧ e.order = oe.msg ∧ e.goods = ge.msg ∧
24:          ∧ ge.type = EOO|EOFS ∧ oe.type = EOD
-------------------------------------------------------------------------
       Server evidences:
-------------------------------------------------------------------------
25:    case e.type=EOFGD:
26:       return Att.Validate(AttAgr^{C,S},oe) ∧ Att.Validate(AttAgr^{S,C},ge) ∧
27:          ∧ ValidateTrade(e.order,oe.ctime,e.goods,ge.ctime)=GoodsOk ∧
28:          ∧ ge.type = EOFS ∧ oe.msg = e.order ∧ ge.msg = e.goods ∧
29:          ∧ oe.type = EOO
-------------------------------------------------------------------------
30:    case e.type=EOGD:
31:       return Att.Validate(AttAgr^{C,S},oe) ∧ Att.Validate(AttAgr^{S,C},ge) ∧
32:          ∧ ValidateTrade(e.order,oe.ctime,e.goods,ge.ctime)=GoodsOk ∧
33:          ∧ ge.type = EOD ∧ oe.msg = e.order ∧ ge.msg = e.goods ∧
34:          ∧ oe.type = EOO
-------------------------------------------------------------------------
35:    case e.type=EOCO:
36:       return Att.Validate(AttAgr^{C,S},oe) ∧ oe.type = EOO ∧ oe.msg = e.order ∧
37:          ∧ ValidateTrade(e.order,oe.ctime,⊥,⊥)=OrderOk
-------------------------------------------------------------------------
38:    return false
```

Figure 12: Implementation of order layer *Validate* efficient algorithm for order evidence validation. In the algorithm, for brevity and simplicity, we assume that the notary N, is also the notary of the order agreement attestation channels.

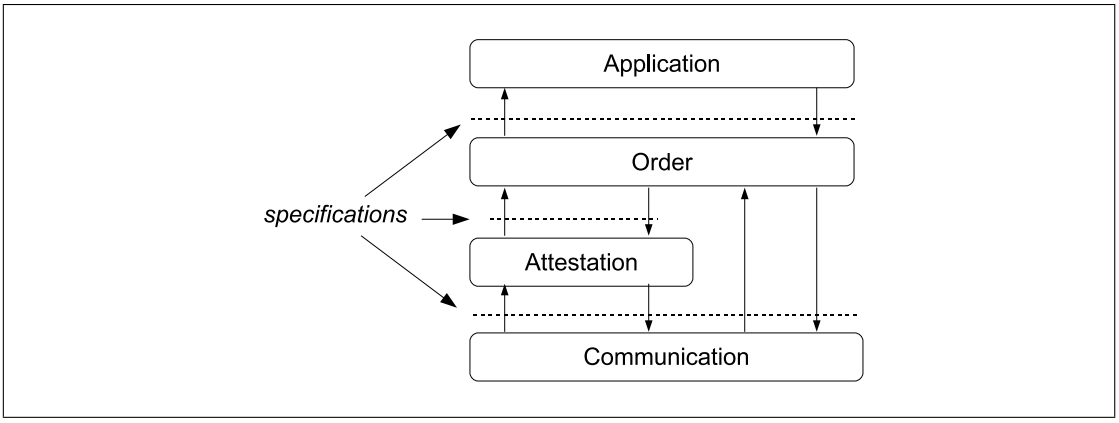


Figure 13: Layered adversarial specifications model for the protocol.

**Definition 2 (Bounded initialization $\mathcal{S}^{INIT-B}_{ORDER}$ predicate)** Predicate $\mathcal{S}^{INIT-B}_{ORDER}(X)$ is `true` for execution $X \in \mathbb{X}$, if for every event

$$\xi =< p, \text{'Order.Init'}, (1^k, r, addr) >\in X,$$

there is an event

$$< p, \text{'Order.InitResult'}, ((vk_0, vk_1), \Delta) >\in X[t(\xi), t(\xi) + \Delta_{order}],$$

where $p \in \mathbb{P}$ and $vk_0, vk_1, addr, r \in \{0,1\}^*$ and $\Delta \in \mathbb{R}$. In that case we also say that in execution $X$, protocol machine $p$ is *properly initialized for order* with address $addr$ and validation keys $vk_0, vk_1$.

Next, denote by $\mathcal{AGR}^{ORDER}$ the domain of possible order agreements. We define that protocol machine processor has opened a valid order channel if it was properly initialized, and had returned successful channel open indication when was supplied a valid order agreement. In addition, for simplicity, valid client machines restrict the (adversarial) upper layer to issue valid orders and valid server machines restrict the upper layer to issue goods which fit the orders.

**Definition 3 (Valid order role *Order.ValidOpen* predicate)** Predicate $Order.ValidOpen(X, OrderAgr, p, \rho)$ is `true` for execution $X \in \mathbb{X}$, order agreement $OrderAgr \in \mathcal{AGR}^{ORDER}$, protocol machine processor $p \in \mathbb{P}$, and role $\rho \in \{\text{'S'}, \text{'C'}, \text{'N'}\}$, if $p$ is *properly initialized for order with address addr, validation keys* $vk_0, vk_1$, and for every event

$$\xi =< p, \text{'Order.OpenChannel'}, (OrderAgr, \rho) >\in X,$$

s.t., $OrderAgr.\rho = (addr, vk_0, vk_1)$, there was an event

$$< p, \text{'Order.OpenChannelResult'}, \texttt{true} >\in X[t(\xi), t(\xi) + \Delta_{order}],$$

and in addition,
$\forall \zeta =<p, \text{'Order'}, (OrderAgr, order)>\in X :$
$\quad OrderAgr.ValidateTrade(order, t(\zeta), \perp, \perp) = OrderOk,$
and
$\forall \zeta =<p, \text{'VendRequest'}, (OrderAgr, order)>\in X$, there is a subsequent event
$\varsigma =<p, \text{'VendRequestResult'}, (OrderAgr, goods)>\in X[t(\zeta), t(\zeta) + OrderAgr.\Omega_{goods}]$, s.t.,
$\quad OrderAgr.ValidateTrade(order, t(\zeta), \perp, \perp) = OrderOk \implies$
$\quad\quad OrderAgr.ValidateTrade(order, t(\zeta), goods, t(\varsigma)) = GoodsOk.$

As in attestation, we define additional liveness predicate, to bound the time to open a channel.

**Definition 4 (Bounded open channel $\mathcal{S}^{BOC}_{ORDER}(X)$ predicate)** Predicate $\mathcal{S}^{BOC}_{ORDER}(X)$ is `true` for execution $X \in \mathbb{X}$, if for every event

$$\xi =< p, \text{'Order.OpenChannel'}, (OrderAgr, \rho) >\in X,$$

there was an event

$$< p, \text{'Order.OpenChannelResult'}, res >\in X[t(\xi), t(\xi) + \Delta_{order}],$$

where $p \in \mathbb{P}, OrderAgr \in \mathcal{AGR}^{ATT}, res \in \{0,1\}^*, \rho \in \{\text{'S'}, \text{'C'}, \text{'N'}\}$.

The following predicate combines all initialization predicates.

**Definition 5 (Initialization $\mathcal{S}^{INIT}_{ORDER}$ predicate)** For execution $X \in \mathbb{X}$,

$$\mathcal{S}^{INIT}_{ORDER}(X) \equiv \mathcal{S}^{INIT-B}_{ORDER}(X) \wedge \mathcal{S}^{INIT-U}_{ORDER}(X) \wedge \mathcal{S}^{BOC}_{ORDER}(X)$$

#### 4.2.2 Correctness specifications

We begin by defining a predicate to identify delivery of invalid evidences to upper layer,

**Definition 6 (Invalid order result $\mathcal{S}^{I\text{-}OR}_{ORDER}$ predicate)** Predicate $\mathcal{S}^{I\text{-}OR}_{ORDER}(X)$ is `true` for execution $X \in \mathbb{X}$, if there exist processor $p \in \mathbb{P}$, order agreement $OrderAgr \in \mathcal{AGR}^{ORDER}$, role $\rho \in \{\text{'S'}, \text{'C'}\}$ and event $<p, \text{'OrderResult'}, (OrderAgr, e)>\in X$, s.t.,

$$Order.ValidOpen(X, OrderAgr, p, \rho) = \texttt{true} \wedge Order.Validate(OrderAgr, e) = \texttt{false}$$

**Remark 1** We assume, for simplicity, in the rest of this section, that goods are always to be issued immediately, i.e., $\forall OrderAgr \in \mathcal{AGR}^{ORDER} : OrderAgr.\Omega_{goods} = 0$. The extension of the rest of the predicates in this section with non-zero goods issue time, is trivial, e.g., see footnote for Definition 8.

We now define adversarial win predicates on the order layer interfaces. In the following predicate, we define that an adversarial client had succeeded in forging evidence of goods and receipt if it provides valid evidence of such, for a honest server, which had never issued the goods in the evidence.

**Definition 7 (Forging evidence of goods and receipt $\mathcal{S}^{F-EOGR}_{ORDER}$ predicate)** Predicate $\mathcal{S}^{F-EOGR}_{ORDER}(X)$ is `true` for execution $X \in \mathbb{X}$, if an adversary $\mathcal{A}$ outputs an order agreement and evidence, $(OrderAgr, e)$, s.t.,

1. $Order.Validate(OrderAgr, e)$ is `true`, and
2. $e.type = EOGR$, and
3. $\exists n \in \mathbb{P}, Order.ValidOpen(X, OrderAgr, n, \text{'N'}) = \texttt{true}$, and
4. $\exists s \in \mathbb{P},\ Order.ValidOpen(X, OrderAgr, s, \text{'S'}) = \texttt{true}$, and
5. $<s, \text{'VendRequestResult'}, (OrderAgr, e.goods)>\notin X[e.ctime]$.

Next, we define as adversarial win, if a valid evidence of goods delivery could be output, but no order described by the evidence took place, or no goods were actually delivered, to a honest client.

**Definition 8 (Forging evidence of goods delivery $\mathcal{S}^{F-EOGD}_{ORDER}$ predicate)** Predicate $\mathcal{S}^{F-EOGD}_{ORDER}(X)$ is `true` for execution $X \in \mathbb{X}$, if an adversary $\mathcal{A}$ outputs an order agreement and evidence, $(OrderAgr, e)$, s.t.,

1. $Order.Validate(OrderAgr, e)$ is `true`, and
2. $e.type = EOGD$, and
3. $\exists n \in \mathbb{P}, Order.ValidOpen(X, OrderAgr, n, \text{'N'}) =$ `true`, and
4. $\exists c \in \mathbb{P},\ Order.ValidOpen(X, OrderAgr, c, \text{'C'}) =$ `true`, and
   (a) $<c, \text{'Order'}, (OrderAgr, e.order)> \notin X[e.ctime - \Delta_{order}, e.ctime]$ [2], or
   (b) $<c, \text{'OrderResult'}, (OrderAgr, \{EOGR, ctime, order, e.goods, \sigma\})> \notin X[e.ctime - \Delta_{order}, e.ctime]$.

For forgery of an evidence of failed order, we define a predicate to capture a valid EOFO evidence, where, however, the goods issued by the application to server's order layer are always valid, and there was sustained communication between the notary and the server.

**Definition 9 (Forging evidence of failed order $\mathcal{S}^{F-EOFO}_{ORDER}$ predicate)** Predicate $\mathcal{S}^{F-EOFO}_{ORDER}(X)$ is `true` for execution $X \in \mathbb{X}$, if an adversary $\mathcal{A}$ outputs an order agreement and evidence, $(OrderAgr, e)$, s.t.,

1. $Order.Validate(OrderAgr, e)$ is `true`, and
2. $e.type = EOFO$, and
3. $\exists s \in \mathbb{P},\ Order.ValidOpen(X, OrderAgr, s, \text{'S'}) =$ `true`, and
4. $\exists n \in \mathbb{P},\ Order.ValidOpen(X, OrderAgr, n, \text{'N'}) =$ `true`, and
5. $\mathcal{S}^{LinkOk}_{ATT}(X, AttAgr^{\{C,S\}}, e.ctime - \Delta_{order}, e.ctime, \Delta_{att}, \text{'S'}) =$ `true`, and
6. $\mathcal{S}^{LinkOk}_{ATT}(X, AttAgr^{\{S,C\}}, e.ctime - \Delta_{order}, e.ctime, \Delta_{att}, \text{'C'}) =$ `true`,

where $AttAgr^{\{C,S\}} = OrderAgr.((C.addr, C.vk_{att}), (S.addr, S.vk_{att}), (N.addr, N.vk_{att}))$ and $AttAgr^{\{S,C\}} = OrderAgr.((S.addr, S.vk_{att}), (C.addr, C.vk_{att}), (N.addr, N.vk_{att}))$.

For forgery of an evidence of failed goods delivery, we define a predicate to capture a valid EOFGD evidence, where, however, the goods issued by the application to server's order layer are always valid, and there was sustained communication between the notary and the client.

**Definition 10 (Forging evidence of failed goods delivery $\mathcal{S}^{F-EOFGD}_{ORDER}$ predicate)** Predicate $\mathcal{S}^{F-EOFGD}_{ORDER}(X)$ is `true` for execution $X \in \mathbb{X}$, if an adversary $\mathcal{A}$ outputs an order agreement and evidence, $(OrderAgr, e)$, s.t.,

1. $Order.Validate(OrderAgr, e)$ is `true`, and
2. $e.type = EOFGD$, and
3. $\exists c \in \mathbb{P},\ Order.ValidOpen(X, OrderAgr, c, \text{'C'}) =$ `true`, and
4. $\exists n \in \mathbb{P},\ Order.ValidOpen(X, OrderAgr, n, \text{'N'}) =$ `true`, and
5. $\mathcal{S}^{LinkOk}_{ATT}(X, AttAgr^{\{S,C\}}, e.ctime - \Delta_{order}, e.ctime, \Delta_{att}, \text{'S'}) =$ `true`,

[2] See Remark 1; if we are to consider $OrderAgr.\Omega_{goods} \neq 0$, condition should be rewritten as $<c, \text{'Order'}, (OrderAgr, e.order)> \notin X[e.ctime - \Delta_{order} - OrderAgr.\Omega_{goods}, e.ctime]$

where $AttAgr^{\{S,C\}} = OrderAgr.((S.addr, S.vk_{att}), (C.addr, C.vk_{att}), (N.addr, N.vk_{att}))$.

For forgery of evidence of client order, we require a valid EOCO evidence, where, however, no client order have been placed.

**Definition 11 (Forging evidence of client order $\mathcal{S}^{F-EOCO}_{ORDER}$ predicate)** Predicate $\mathcal{S}^{F-EOCO}_{ORDER}(X)$ is `true` for execution $X \in \mathbb{X}$, if an adversary $\mathcal{A}$ outputs an order agreement and evidence, $(OrderAgr, e)$, s.t.,

1. $Order.Validate(OrderAgr, e)$ is `true`, and
2. $e.type = EOCO$, and
3. $\exists c \in \mathbb{P}$, $Order.ValidOpen(X, OrderAgr, c, \text{'C'}) =$ `true`, and
4. $\exists n \in \mathbb{P}$, $Order.ValidOpen(X, OrderAgr, n, \text{'N'}) =$ `true`, and
5. $<c, \text{'Order'}, (OrderAgr, e.order)> \notin X[e.ctime]$.

We define additional predicates, to capture fairness of evidence exchange for both sides, i.e., if the client have got an evidence for the order, then the server has also obtained one, and vice versa. We begin with server, and state that if the client presents an evidence of goods and receipt, the server must have obtained an evidence of goods delivery or failed goods delivery (see Figure 10 to recall that our protocol supports recovery in the EOFGD case, and the client may still obtain EOGD with notary's assistance).

**Definition 12 (Server Evidence for Client Evidence $\mathcal{S}^{N\text{-}SEfCE}_{ORDER}$ predicate)** Predicate $\mathcal{S}^{N\text{-}SEfCE}_{ORDER}(X)$ is `true` for execution $X \in \mathbb{X}$, if an adversary $\mathcal{A}$ outputs an order agreement and evidence, $(OrderAgr, e_c)$, s.t.,

1. $Order.Validate(OrderAgr, e_c)$ is `true`, and
2. $e_c.type = EOGR$, and
3. $\exists s \in \mathbb{P}$, where $Order.ValidOpen(X, OrderAgr, s, \text{'S'}) =$ `true`, and
4. $\exists n \in \mathbb{P}$, $Order.ValidOpen(X, OrderAgr, n, \text{'N'}) =$ `true`, and
5. $<s, \text{'OrderResult'}, (OrderAgr, e_s)> \notin X[e_c.ctime, e_c.ctime + \Delta_{order}]$, s.t., $e_s.order = e_c.order$ and $e_s.goods = e_c.goods$, and $e_s.type = EOGD \vee EOFGD$,
6. $\mathcal{S}^{LinkOk}_{ATT}(X, AttAgr^{\{S,C\}}, e.ctime - \Delta_{order}, e.ctime, \Delta_{att}, \text{'C'}) =$ `true`,

where $AttAgr^{\{S,C\}} = OrderAgr.((S.addr, S.vk_{att}), (C.addr, C.vk_{att}), (N.addr, N.vk_{att}))$.

**Definition 13 (Client evidence for server evidence $\mathcal{S}^{N\text{-}CEfSE}_{ORDER}$ predicate)** Predicate $\mathcal{S}^{N\text{-}CEfSE}_{ORDER}(X)$ is `true` for execution $X \in \mathbb{X}$, if an adversary $\mathcal{A}$ outputs an order agreement and evidence, $(OrderAgr, e_s)$, s.t.,

1. $Order.Validate(OrderAgr, e_s)$ is `true`, and
2. $e_s.type = EOGD$, and
3. $\exists c \in \mathbb{P}$, where $Order.ValidOpen(X, OrderAgr, c, \text{'C'}) =$ `true`, and
4. $\exists n \in \mathbb{P}$, $Order.ValidOpen(X, OrderAgr, n, \text{'N'}) =$ `true`, and
5. $<c, \text{'OrderResult'}, (OrderAgr, e_c)> \notin X[e_s.ctime - \Delta_{order}, e_s.ctime]$, s.t., $e_c.order = e_s.order$ and $e_c.goods = e_s.goods$, and $e_c.type = EOGR$.

We now combine the above predicates to define adversarial win specification for attestation layer.

**Definition 14 (Adversarial win $\mathcal{S}_{ORDER}.AW$ predicate)** Let $\mathcal{S}_{ORDER}.AW(X)$ be order layer adversarial win specification for execution $X \in \mathbb{X}$,

$$\begin{aligned}\mathcal{S}_{ORDER}.AW(X) \equiv \mathcal{S}_{ATT}^{LINK}(X) \wedge \mathcal{S}_{ATT}^{INIT}(X) \wedge \neg \mathcal{S}_{ATT}.AW(X) \wedge (\mathcal{S}_{ORDER}^{I\text{-}OR}(X) \vee \\ \vee \mathcal{S}_{ORDER}^{F-EOGR}(X) \vee \mathcal{S}_{ORDER}^{F-EOGD}(X) \vee \mathcal{S}_{ORDER}^{F-EOFO}(X) \vee \\ \vee \mathcal{S}_{ORDER}^{F-EOFGD}(X) \vee \mathcal{S}_{ORDER}^{F-EOCO}(X) \vee \mathcal{S}_{ORDER}^{N\text{-}SEfCE}(X) \vee \mathcal{S}_{ORDER}^{N\text{-}CEfSE}(X))\end{aligned}$$

### 4.2.3 Liveness specifications

The first liveness predicate is to assert that a honest server, selling goods, will always provide an evidence for upper layer.

**Definition 15 (Server gets evidence $\mathcal{S}_{ORDER}^{SE_1}$ predicate)** Predicate $\mathcal{S}_{ORDER}^{SE_1}(X)$ is `true` for execution $X \in \mathbb{X}$, if for every protocol machine processor $s \in \mathbb{P}$ s.t., for every event $\zeta = <s, \text{'VendRequestResult'}, (OrderAgr, goods)> \in X$, where $OrderAgr \in \mathcal{AGR}^{ORDER}$, that was preceded by a matching $\xi = <s, \text{'VendRequest'}, (OrderAgr, order)> \in X$ event, s.t.,

1. $\exists n \in \mathbb{P}, Order.ValidOpen(X, OrderAgr, n, \text{'N'}) = \texttt{true}$, and
2. $Order.ValidOpen(X, OrderAgr, s, \text{'S'}) = \texttt{true}$, and
3. $OrderAgr.ValidateTrade(order, t(\xi), \perp, \perp) = OrderOk$,

there was an $<s, \text{'OrderResult'}, (OrderAgr, e)> \in X[t(\zeta), t(\zeta) + \Delta_{order}]$ event, s.t.,

1. $e.goods = goods$, and
2. $e.type = EOGD \vee EOFGD \vee EOCO$, and
3. $Order.Validate(OrderAgr, e) = \texttt{true}$.

The second liveness requirement we define is about sustained link to notary for honest parties. The predicate is to assert that when client gets an evidence of goods and receipt, the honest server acquires either evidence of goods delivery or evidence of failed goods delivery. Notice that the predicate definition is quite similar to previous Definition 15.

**Definition 16 (Server gets evidence $\mathcal{L}_{ORDER}^{SE_2}$ predicate)** Predicate $\mathcal{L}_{ORDER}^{SE_2}(X)$ is `true` for execution $X \in \mathbb{X}$, if for every protocol machine processor $s \in \mathbb{P}$, s.t., for every event $\zeta = <s, \text{'VendRequestResult'}, (OrderAgr, goods)> \in X$, where $OrderAgr \in \mathcal{AGR}^{ORDER}$, that was preceded by a matching $\xi = <s, \text{'VendRequest'}, (OrderAgr, order)> \in X$ event, there was an $<s, \text{'OrderResult'}, (OrderAgr, e)> \in X[t(\zeta), t(\zeta) + \Delta_{order}]$ event, s.t.,

1. $e.goods = goods$, and
2. $e.type = EOGD \vee EOFGD$, and

whenever,

1. $OrderAgr.ValidateTrade(order, t(\xi), \perp, \perp) = OrderOk$, and
2. $Order.ValidOpen(X, OrderAgr, s, \text{'S'}) = \texttt{true}$, and
3. $\exists n \in \mathbb{P}, Order.ValidOpen(X, OrderAgr, n, \text{'N'}) = \texttt{true}$, and

4. $\mathcal{S}^{LinkOk}_{ATT}(X, AttAgr^{\{S,C\}}, t(\xi), t(\xi)+\Delta_{order}, \Delta_{att}, \text{'C'}) = \texttt{true}$,
   where $AttAgr^{\{S,C\}} = OrderAgr.((S.addr, S.vk_{att}), (C.addr, C.vk_{att}), (N.addr, N.vk_{att}))$.

In next predicate we assert, that a honest client, making an order, and having sustained communication with a honest notary, will always provide evidence for the order, to the upper layer.

**Definition 17 (Client gets evidence $\mathcal{L}^{CE}_{ORDER}$ predicate)** Predicate $\mathcal{L}^{CE}_{ORDER}(X)$ is `true` for execution $X \in \mathbb{X}$, if for every $\xi = <c, \text{'Order'}, (OrderAgr, order)> \in X, c \in \mathbb{P}$ event, where $OrderAgr \in \mathcal{AGR}^{ORDER}$, the protocol ensures
$\zeta = <c, \text{'OrderResult'}, (OrderAgr, e)> \in X[t(\xi), t(\xi)+\Delta_{order}]$, s.t.,

1. $e.order = order$, and
2. $e.type = EOGR \vee EOFO$, and
3. $Order.Validate(OrderAgr, e) = \texttt{true}$,

whenever:

1. $\exists n \in \mathbb{P}, Order.ValidOpen(X, OrderAgr, n, \text{'N'}) = \texttt{true}$, and
2. $\exists c \in \mathbb{P}, Order.ValidOpen(X, OrderAgr, c, \text{'C'}) = \texttt{true}$, and
3. $\mathcal{S}^{LinkOk}_{ATT}(X, AttAgr^{\{C,S\}}, t(\xi), t(\xi)+\Delta_{order}, \Delta_{att}, \text{'C'}) = \texttt{true}$,
   where $AttAgr^{\{C,S\}} = OrderAgr.((C.addr, C.vk_{att}), (S.addr, S.vk_{att}), (N.addr, N.vk_{att}))$, and
4. $\mathcal{S}^{LinkOk}_{COMM}(X, c, n, t(\xi)+\Delta_{order}, t(\xi), t(\xi)+\Delta_{order}, \Delta_{comm}) = \texttt{true}$

**Remark 2** Since our implementation supports recovery over the regular communication channel, we have stated dependency on the communication channel link, in the above Definition 17. It is conceivable, however, that order layer notary would be a different notary from the attestation layer notary (the case that we do not discuss, for brevity), and each order layer participant would maintain pairwise attestation links with parties specified by the order layer agreement, e.g., a client would maintain bi-directional attestation link with the server, and additional bi-directional attestation link with the notary. In that case, the former predicate should be rewritten to reflect aforementioned attestation links, and would contain no dependency on the communication layer.

Our last requirement for sustained link and honest parties, is to assert that an order transaction successfully proceeds, goods are issued, and evidences are received for both parties,

**Definition 18 (Connected client and server complete transaction $\mathcal{L}^{CS}_{ORDER}$ predicate)** Predicate $\mathcal{L}^{CS}_{ORDER}(X)$ is `true` for execution $X \in \mathbb{X}$, if for every event
$\xi = <c, \text{'Order'}, (OrderAgr, order)> \in X$, where $c \in \mathbb{P}$, $OrderAgr \in \mathcal{AGR}^{ORDER}$, $AttAgr^{\{C,S\}} = OrderAgr.((C.addr, C.vk_{att}), (S.addr, S.vk_{att}), (N.addr, N.vk_{att}))$, and
$AttAgr^{\{S,C\}} = OrderAgr.((S.addr, S.vk_{att}), (C.addr, C.vk_{att}), (N.addr, N.vk_{att}))$, s.t.,

1. $Order.ValidOpen(X, OrderAgr, c, \text{'C'}) = \texttt{true}$, and
2. $\exists s \in \mathbb{P}, Order.ValidOpen(X, OrderAgr, s, \text{'S'}) = \texttt{true}$, and
3. $\exists n \in \mathbb{P}, Order.ValidOpen(X, OrderAgr, n, \text{'N'}) = \texttt{true}$, and
4. $\mathcal{S}^{LinkOk}_{ATT}(X, AttAgr^{\{C,S\}}, t(\xi), t(\xi)+\Delta_{order}, \Delta_{att}, \text{'C'}) = \texttt{true}$, and

5. $\mathcal{S}_{ATT}^{LinkOk}(X, AttAgr^{\{C,S\}}, t(\xi), t(\xi) + \Delta_{order}, \Delta_{att}, \text{'}S\text{'}) = \texttt{true}$, and
6. $\mathcal{S}_{ATT}^{LinkOk}(X, AttAgr^{\{S,C\}}, t(\xi), t(\xi) + \Delta_{order}, \Delta_{att}, \text{'}C\text{'}) = \texttt{true}$, and
7. $\mathcal{S}_{ATT}^{LinkOk}(X, AttAgr^{\{S,C\}}, t(\xi), t(\xi) + \Delta_{order}, \Delta_{att}, \text{'}S\text{'}) = \texttt{true}$,

there are subsequent,

1. $\zeta_c = <c, \text{'}OrderResult\text{'}, (OrderAgr, e_c)> \in X[t(\xi), t(\xi) + \Delta_{order}]$, and
2. $\zeta_s = <s, \text{'}OrderResult\text{'}, (OrderAgr, e_s)> \in X[t(\xi), t(\xi) + \Delta_{order}]$,

s.t.,

1. $e_c.type = EOGR$, and $e_c.order = order$, and
2. $e_s.type = EOGD$, and $e_s.order = order$, and $e_s.goods = e_c.goods$.

# 5 Analysis

In the first theorem we show that our protocol preserves bounded initialization and role validity predicates, if such preserved by lower layers.

**Theorem 1 (Initialization)** *Let $X \in \mathbb{X}$ be an execution with an OL-protocol machine, then,*

$$\mathcal{S}_{COMM}^{INIT}(X) \wedge \mathcal{S}_{ATT}^{INIT}(X) \Rightarrow \mathcal{S}_{ORDER}^{INIT}(X)$$

Proof 1 Straightforward from initialization code in Figure 4. The calculated $\Delta_{order}$ value deterministically depends on $\Delta_{comm}$ and $\Delta_{att}$ (lines 2–5), and success in opening a channel directly depends on opening lower layer channels (lines 12, 22–26). ■

Next, we are to show, for a layered configuration presented in Figure 13, that order layer specifications are sustained if specifications on lower layers are upheld. We consider order layer OL-protocol machine, as in Section 3.3, interacting with attestation and communication layers via the respective interfaces. We consider an execution as won by the adversary, if the lower layer specifications were upheld, however, the order layer specifications were not.

**Theorem 2 (Correctness)** *An OL-protocol machine securely-prevents $\mathcal{S}_{ORDER}.AW$ (Definition 14) against poly-time adversaries, if implemented with signature scheme $\mathcal{DS}$ upholding soundness and security specifications (Definitions 29–30) and on top of an attestation layer which securely-prevents $\mathcal{S}_{ATT}.AW$ (Definition 45), against poly-time adversaries.*

*Discussion.* In our protocol we rely on validity and security of the attestation layer evidences and security of the signature scheme. So each time an order layer evidence (which is typically a pair of attestation layer evidences) is shown to be incorrect, we are to show, that in that execution either some attestation evidence was forged, or that the signature scheme used to sign EOFO evidences is not secure, or as the last option, goods supplied by adversarial upper layer do not match the order.

Proof 2 From Lemmas 1–7. ■

**Lemma 1** *Let $X = X(OL, \mathcal{A}, R)$, be an execution where $\mathcal{S}_{ATT}.AW(X) = \texttt{false}$, $\mathcal{S}_{ATT}^{INIT}(X) = \texttt{true}$, then $\mathcal{S}_{ORDER}^{I\text{-}OR}(X) = \texttt{false}$.*

LEMMA PROOF 1 We will show a contradiction. Assume $\mathcal{S}_{ORDER}^{I\text{-}OR}(X) = \texttt{true}$ which means $\exists\, p \in \mathbb{P}$, where $<p, \text{'OrderResult'}, (OrderAgr, e)> \in X$, such that,
$Order.ValidOpen(X, OrderAgr, p, \rho) = \texttt{true}$ for some role $\rho \in \{\text{'S'}, \text{'C'}, \text{'N'}\}$, however
$Order.Validate(OrderAgr, e) = \texttt{false}$ (Figure 12). The following are the only evidence types which are output by the protocol and which could fail *Order.Validate*:

**EOGR** The only place where EOGR evidence is output is Figure 7 line 18. Beforehand *ValidateTrade* is checked on line 17, therefore with invalid EOGR it must be the case, by validation Figure 12, lines 20–24, that the attestation evidences are invalid, $\mathcal{S}_{ATT}^{I\text{-}Recv}(X) \vee \mathcal{S}_{ATT}^{I\text{-}Send}(X)$, which is a contradiction.

**EOCO** The only place where EOCO evidence is output is Figure 5 line 10. However, in Figure 5, *ValidateTrade* is checked on line 2, therefore with invalid EOCO it must be the case, by validation code, Figure 12, lines 35–37, that the corresponding EOO evidence (received in Figure 5, line 1) is invalid, therefore $\mathcal{S}_{ATT}^{I\text{-}Recv}(X)$, which is a contradiction.

**EOGD,EOFGD** Similarly, in Figure 7, *ValidateTrade* is checked on line 17, and in Figure 5, on line 2, therefore *goods* (line 3) are valid, by Definition 6. Hence, as previously, with invalid EOGD or EOFGD, it must be the case that the EOO/EOD/EOFS evidences are invalid, $\mathcal{S}_{ATT}^{I\text{-}Recv}(X) \vee \mathcal{S}_{ATT}^{I\text{-}Send}(X)$, which is a contradiction.

**EOFO** There are three ways an EOFO is output and checked by validation code, Figure 12, line 1, lines 12–15 and lines 16–19. In the first respective case, the EOFO is a notary signed EOFO, however, in Figure 7, line 11, this notary signed EOFO is explicitly validated before being passed to upper layer, therefore it must be the later EOFO case, where EOFO consists of lower layer evidences, where contradiction $\mathcal{S}_{ATT}^{I\text{-}Recv}(X) \vee \mathcal{S}_{ATT}^{I\text{-}Send}(X)$ follows similarly to previous considerations.

■

**Lemma 2** *Let $X = X(OL, \mathcal{A}, R)$, be an execution where $\mathcal{S}_{ATT}.AW(X) = \texttt{false}$, $\mathcal{S}_{ATT}^{INIT}(X) = \texttt{true}$, $\mathcal{S}_{ATT}^{LINK}(X) = \texttt{true}$, $\mathcal{S}_{\mathcal{DS}}.AW(X) = \texttt{false}$ and $\mathcal{S}_{\mathcal{DS}}^{Sound}(X) = \texttt{true}$, then $\mathcal{S}_{ORDER}^{F-EOFO}(X) = \texttt{false}$.*

LEMMA PROOF 2 We will show a contradiction. Let $(OrderAgr, e)$ be an order agreement and EOFO evidence output by $\mathcal{A}$. EOFO could by obtained in three cases - when order message fails with EOFS (Figure 7, line 4), when server does not provide goods (Figure 6, line 13), or when notary signed EOFO is obtained (Figure 7 line 11). By Definition 9 of $\mathcal{S}_{ORDER}^{F-EOFO}$ both server and notary are validly open and no attestation failure indications took place; by $\mathcal{S}_{ATT}^{LINK}(X) = \texttt{true}$, server had obtained an EOD for goods sent on line 4–6 of Figure 5, and client could not obtain EOFS on line 4 of Figure 7. Thus, for the first two cases it must be the case that $\mathcal{S}_{ATT}^{F-EOD}(X) \vee \mathcal{S}_{ATT}^{F-EOFS}(X) = \texttt{true}$ which is a contradiction. For the latter case of invalid goods delivered, since server is validly open - the goods obtained on line 3 pass *ValidateTrade* by Definition 3. Therefore it must be the case of $\mathcal{S}_{ATT}^{F-EOO}(X) = \texttt{true}$ (on line 15 of Figure 7) which is again a contradiction. For the (last) case of notary signed EOFO, $\mathcal{S}_{\mathcal{DS}}.AW(X) = \texttt{true}$, is a contradiction, since as previously discussed, server had obtained an EOD and notary is valid.
■

**Lemma 3** *Let $X = X(OL, \mathcal{A}, R)$, be an execution where $\mathcal{S}_{ATT}.AW(X) = \texttt{false}$, $\mathcal{S}_{ATT}^{INIT}(X) = \texttt{true}$, $\mathcal{S}_{ATT}^{LINK}(X) = \texttt{true}$, $\mathcal{S}_{\mathcal{DS}}.AW(X) = \texttt{false}$ and $\mathcal{S}_{\mathcal{DS}}^{Sound}(X) = \texttt{true}$, then $\mathcal{S}_{ORDER}^{F-EOGR}(X) = \texttt{false}$.*

LEMMA PROOF 3 We will show a contradiction. Let $(OrderAgr, e)$ be the tuple output by $\mathcal{A}$. Since there was no $<s, \text{'VendRequestResult'}, (OrderAgr, e.goods)> \in X[e.ctime]$, and $\exists s \in \mathbb{P}$,

s.t., $Order.ValidOpen(X, OrderAgr, s, \text{'S'}) = \texttt{true}$, the code on line 4, Figure 5, was never executed at time $e.ctime$. Since validity of $e$ itself is assured by Lemma 1, it must be the case of forged $e.\sigma$, $\mathcal{S}_{ATT}^{F-EOO}(X) \vee \mathcal{S}_{ATT}^{F-EOFS}(X)$, which is a contradiction. ■

For the following Lemmas, we omit the proofs which follow along almost identical considerations.

**Lemma 4** *Let $X = X(OL, \mathcal{A}, R)$, be an execution where $\mathcal{S}_{ATT}.AW(X) = \texttt{false}$, $\mathcal{S}_{ATT}^{INIT}(X) = \texttt{true}$, $\mathcal{S}_{ATT}^{LINK}(X) = \texttt{true}$, $\mathcal{S}_{\mathcal{DS}}.AW(X) = \texttt{false}$ and $\mathcal{S}_{\mathcal{DS}}^{Sound}(X) = \texttt{true}$, then $\mathcal{S}_{ORDER}^{F-EOGD}(X) = \texttt{false}$.*

**Lemma 5** *Let $X = X(OL, \mathcal{A}, R)$, be an execution where $\mathcal{S}_{ATT}.AW(X) = \texttt{false}$, $\mathcal{S}_{ATT}^{INIT}(X) = \texttt{true}$, $\mathcal{S}_{ATT}^{LINK}(X) = \texttt{true}$, $\mathcal{S}_{\mathcal{DS}}.AW(X) = \texttt{false}$ and $\mathcal{S}_{\mathcal{DS}}^{Sound}(X) = \texttt{true}$, then $\mathcal{S}_{ORDER}^{F-EOCO}(X) = \texttt{false}$.*

**Lemma 6** *Let $X = X(OL, \mathcal{A}, R)$, be an execution where $\mathcal{S}_{ATT}.AW(X) = \texttt{false}$, $\mathcal{S}_{ATT}^{INIT}(X) = \texttt{true}$, $\mathcal{S}_{ATT}^{LINK}(X) = \texttt{true}$, $\mathcal{S}_{\mathcal{DS}}.AW(X) = \texttt{false}$ and $\mathcal{S}_{\mathcal{DS}}^{Sound}(X) = \texttt{true}$, then $\mathcal{S}_{ORDER}^{N\text{-}SEfCE}(X) = \texttt{false}$.*

**Lemma 7** *Let $X = X(OL, \mathcal{A}, R)$, be an execution where $\mathcal{S}_{ATT}.AW(X) = \texttt{false}$, $\mathcal{S}_{ATT}^{INIT}(X) = \texttt{true}$, $\mathcal{S}_{ATT}^{LINK}(X) = \texttt{true}$, $\mathcal{S}_{\mathcal{DS}}.AW(X) = \texttt{false}$ and $\mathcal{S}_{\mathcal{DS}}^{Sound}(X) = \texttt{true}$, then $\mathcal{S}_{ORDER}^{N\text{-}CEfSE}(X) = \texttt{false}$.*

**Theorem 3 (Liveness 1)** *Let $X \in \mathbb{X}$ be an execution with an OL-protocol machine, where $\mathcal{S}_{ATT}.AW(X) = \texttt{false}$, $\mathcal{S}_{\mathcal{DS}}.AW(X) = \texttt{false}$ and $\mathcal{S}_{\mathcal{DS}}^{Sound}(X) = \texttt{true}$ then,*

$$\mathcal{S}_{COMM}^{INIT}(X) \wedge \mathcal{S}_{ATT}^{LINK}(X) \wedge \mathcal{S}_{ATT}^{INIT}(X) \Rightarrow \mathcal{S}_{ORDER}^{SE_1}(X) \wedge \mathcal{L}_{ORDER}^{SE_2}(X) \wedge \mathcal{L}_{ORDER}^{CS}(X)$$

*Discussion.* In a given execution, when a honest server gets an order, it is always able to provide evidence to upper layer, at least the EOCO type evidence, as EOCO just contains the order EOO. Moreover, while the server is always able to provide at least EOCO for a placed order, a sustained link to a honest notary is a promise that at least EOFS for sent goods would be obtained (in the case EOD was not). Therefore, the server would be able to assemble EOGD or EOFGD, from the order EOO and the goods evidence, as shown in the protocol (Figure 5). And in the case when all links are sustained, honest client and server would be able to exchange order and goods, and provide relevant evidences for upper layer, as attestation layer is to provide evidences for all messages sent.

Proof 3 For each of the following predicates,

$\mathcal{S}_{\boldsymbol{ORDER}}^{\boldsymbol{SE_1}}(X)$ Assume that in execution $X \in \mathbb{X}$, there was an order agreement $OrderAgr \in \mathcal{AGR}^{ORDER}$, s.t., $\exists s \in \mathbb{P}$, $Order.ValidOpen(X, OrderAgr, s, \text{'S'}) = \texttt{true}$ and there was an event $<s, \text{'VendRequestResult'}, (OrderAgr, goods)> \in X$, that was preceded by a matching $\xi = <s, \text{'VendRequest'}, (OrderAgr, order)> \in X$ event, with a valid order, i.e., $OrderAgr.ValidateTrade(order, t(\xi), \perp, \perp) = OrderOk$. By $Order.ValidOpen(X, OrderAgr, s, \text{'S'}) = \texttt{true}$ all server issued goods are valid, and by Figure 5, lines 4–10, all path lead to *'OrderResult'*, with an evidence $e$ where $e.type = EOGD \vee EOFGD \vee EOCO$. From Theorem 2, $\mathcal{S}_{ORDER}^{I\text{-}OR}(X) = \texttt{false}$, therefore $Order.Validate(OrderAgr, e) = \texttt{true}$.

$\mathcal{L}_{\boldsymbol{ORDER}}^{\boldsymbol{SE_2}}(X)$ Same as previous $\mathcal{S}_{ORDER}^{SE_1}(X)$, with the restriction of a sustained link between a notary and a server. Therefore, by $\mathcal{S}_{ATT}^{LINK}(X) = \texttt{true}$, in Figure 5, server code, *'OrderResult'* could only be output on line 6 or line 8.

$\mathcal{L}^{\boldsymbol{CS}}_{\boldsymbol{ORDER}}(X)$ Assume that in execution $X \in \mathbb{X}$, there was an order agreement $OrderAgr \in \mathcal{AGR}^{ORDER}$, s.t., client, server and notary are validly open, and there were no attestation failures, for the respective bi-directional attestation agreements, as specified in Definition 18. Since no attestation failures were indicated, in follows from $\mathcal{S}^{LINK}_{ATT}(X) = \texttt{true}$, that EOD evidences were obtained for all messages, within $\Delta_{att}$, from the moment of sending the message. Therefore, for client, Figure 7, after an *'Order'* event, line 3 was executed, followed by line 5, and as the result, for server, Figure 5, line 1 event was handled, within $\Delta_{att}$, followed by line 4, and consequently followed by line 6, within additional $\Delta_{att}$. The execution of former line 4, at Figure 5, triggers line 13 event at the client, Figure 7, where the check on line 17 must succeed, since server is honest and validly open. Therefore, for the client, *'OrderResult'* with an evidence $e_c$, was executed, with $e_c.type = EOGR$, and for the server, *'OrderResult'* with an evidence $e_s$ was executed, with $e_s.type = EOGD$.

■

**Theorem 4 (Liveness 2)** *Let $X \in \mathbb{X}$ be an execution with an OL-protocol machine, where $\mathcal{S}_{ATT}.AW(X) = \texttt{false}$, $\mathcal{S}_{\mathcal{DS}}.AW(X) = \texttt{false}$ and $\mathcal{S}^{Sound}_{\mathcal{DS}}(X) = \texttt{true}$, then,*

$$\mathcal{S}^{INIT}_{COMM}(X) \wedge \mathcal{S}^{LINK}_{COMM} \wedge \mathcal{S}^{LINK}_{ATT}(X) \wedge \mathcal{S}^{INIT}_{ATT}(X) \Rightarrow \mathcal{L}^{CE}_{ORDER}(X)$$

*Discussion.* In a given execution, a honest client which did maintain attestation and communication link to honest notary would either receive EOFS for submitting an order, receive notary signed EOFO if goods were not supplied, or just receive the goods from the server (while having EOD for the order). In each case, it would be able to assemble an evidence for upper layer, as shown in the protocol (Figure 7).

Proof 4 Assume that in execution $X \in \mathbb{X}$, there was an order agreement $OrderAgr \in \mathcal{AGR}^{ORDER}$, s.t., $\exists c \in \mathbb{P}$,
$Order.ValidOpen(X, OrderAgr, c, \text{'C'}) = \texttt{true}$ and $\exists n \in \mathbb{P}$, $Order.ValidOpen(X, OrderAgr, n, \text{'N'}) = \texttt{true}$ with no attestation and communication failures between the two, as specified in Definition 17. Since the placed order is valid, by definition, and $\mathcal{S}^{LINK}_{ATT}(X) = \texttt{true}$ - from Figure 7, lines 4–5 would be executed. In the case line 4 was executed, *'OrderResult'* did take place, with an evidence $e$ where $e.type = EOFO$. From Theorem 2, $\mathcal{S}^{L\text{-}OR}_{ORDER}(X) = \texttt{false}$, therefore $Order.Validate(OrderAgr, e) = \texttt{true}$. Otherwise, line 5 was executed and $timer_{server}$ was set. Since the former timer was set, either line 13 or line 7 was executed. In the case line 13 was executed, either *'OrderResult'* happened with an evidence $e$ where $e.type = EOGR \vee EOFO$ or $\mathcal{S}^{I\text{-}Recv}_{ATT}(X) = \texttt{false}$ (which is a contradiction), and from Theorem 2, $\mathcal{S}^{L\text{-}OR}_{ORDER}(X) = \texttt{false}$, therefore $Order.Validate(OrderAgr, e) = \texttt{true}$. In the other case, line 7 was executed and since $\mathcal{S}^{LINK}_{COMM}(X) = \texttt{true}$ - Figure 6 line 9 was executed within $\Delta_{comm}$, for the notary. Similarly, within $2 \cdot \Delta_{comm}$ either line 1,5 or line 13 would be executed for the notary. The case of line 1 is ruled out by $\mathcal{S}^{F-EOD}_{ATT}(X)$, Theorem 2 (EOD forgery), therefore within additional $\Delta_{comm}$, Figure 7 line 14 or line 10 would be executed, where again all code path lead to *'OrderResult'* with an evidence $e$ where $e.type = EOGR \vee EOFO$; and since $\mathcal{S}^{L\text{-}OR}_{ORDER}(X) = \texttt{false}$, by Theorem 2, $Order.Validate(OrderAgr, e) = \texttt{true}$. ■

# 6 Related Work

We distinguish between two basic approaches to the problem of disputes and cheating in e-commerce: *prevention of disputes*, typically by *fair exchange*, vs. *dispute resolution*, typically by *evidences ('non-repudiation')*. In many cases dispute prevention (fair exchange) is not a viable or sufficient solution. In particular, failure (refusal) to participate by the server, can

result in severe damages to the client. For example, consider a customer ordering payment by her bank, e.g. ordering of Certified Payment Order. If the bank simply ignores such requests, it effectively takes over the customer's deposited funds. Similarly, consider a broker that fails to execute an order, or a supplier failing to provide goods as per agreed-upon schedule.

In the field of fair exchange we should mention works of Nenadic and Zhang [26], Asokan, Shoup and Waidner [3] for fair exchange of digital signatures (over digital items) with possible adaption for confidentiality of exchanged signatures; and Garay and Pomerance [13] for timed, gradual exchange. For an overview of various levels of trusted third party involvement see Zhou,Deng and Bao [33], and for a survey of non-repudiation protocols, see Kremer, Markowitch and Zhou [23, 32].

Unlike fair exchange, with non-repudiation, parties receive evidences regarding exchange properties, such as exchange parties identities, keys involved in the transactions, the time of the exchanged messages, et cetera. The evidences could be validated by third parties, at later time. Non-repudiation was also standardized by ISO [20, 21, 22]. However, existing works on non-repudiation, including the standards, do not provide well-defined specifications, interfaces or proofs of security. Furthermore, these works do not allow the application to define the required properties of orders and goods (as with our 'trade validation function'). Finally, the works do not define an (automated) dispute resolution process.

To our knowledge, much less attention was given to automated, provably-secure dispute resolution, compared to just having a collection of evidences without a precise process for using them. There are several, widely deployed systems for "orders", such as IFX [11], FIX [9], OFX [8], SWIFT [30], ebXML [10]. In such systems, dispute resolution is manual, and depends on participant records and goodwill, or on a trusted arbiter.

Asokan, Herreweghen and Steiner [2] have initiated the discussion of precise dispute resolution with an architecture to support formalization, via a dispute language, and resolution of claims. Tang, Fu and Veijalainen [31] have presented arbitrable e-commerce transaction, via a *benefit sets* which are for disputing parties to prove in case of a dispute, and Ray,Ray and Narasimhamurthy [29] have presented a payment protocol with a promise of automatic dispute resolution, which involves a trusted party as goods advertising proxy between payer and payee; however dispute resolution in [29] actually involves the trusted party completing the transaction (e.g., sending the product to customer). Similarly, the protocol by Markowitch and Saeednia [25] requires participation of both disputing parties to resolve the dispute, by executing abort protocol at the notary, and involves the trusted party in the process of signature recovery.

Herreweghen [15] discussed collecting SET[3] signatures towards proving SET transaction properties to an external verifier. In the protocol by Nenadic et. al. [27] disputes are resolved by notary recovering the digital signature over e-goods (either online or offline), where the goods are assumed to be pre-certified for correctness and there is no notion of bad order by a client.

Finally, all mentioned works [2, 31, 29, 27, 25] do not handle failed or invalid submission of orders or communication failures. In addition, in these works there is no notion of incorrect order or goods, and agreements which govern the bound of goods delivery or issual time and matching goods to orders. Thus we believe this makes the former protocols less suitable as underlying infrastructure for secure e-commerce services. Furthermore, to the best of our knowledge, in past works, protocol-independent specifications for e-commerce layers have never appeared.

We should also mention SEMPER [24], a layered security platform, supports both customer-to-business and business-to-business trade and relies on the accountability of digital signatures. Dispute resolution is supported in SEMPER by means of exporting transaction record to an

[3]SET is a standard adopted by MasterCard and Visa, based on *iKP* Bellare et al. [4], family of protocols for secure credit-card payments. The SET standard seems to have been abandoned.

arbiter running its own instance of SEMPER and examining digital records of the disputing parties, along with parties evidences, in the form of digital signature. Notable feature of SEMPER's legal framework is the notion of electronic commerce agreements, according to which (manual) dispute resolution takes place.

Cox, Tygar and Sirbu's NetBill [6] is an online distributed transactional customer-to-merchant system, which facilitates trade by acting as a trusted intermediate. NetBill does not feature order-to-goods correspondence automatic resolution, and dispute resolution in NetBill is a manual process where the customer presents his signed order and a signed agreement by the merchant for the usage of the security context (e.g., cryptographic checksums and decryption keys), to a NetBill arbiter. Additional types of disputes involve transaction status disputes (e.g., claim that transaction was aborted, but customer was charged anyway) where the parties must present NetBill signed transaction receipts to an arbiter.

# 7 Conclusions

We have introduced a simple yet versatile trade protocol, with arbitrable transactions and concrete, well-defined specifications, which allow provable security and resolution of disputes with arbitrary validation of goods to order fitness in presence of malicious faults or communication failures. An interested reader may refer to [18] regarding how to use the protocol for further construction of final and conditional final payments between principals, or how to conduct trade when a PSP is a trusted party. Our design is practical, layered, and attains automatic dispute resolution, based on precise agreements and relatively simple cryptographic constructions assumed.

# References


[1] X.509 Certificate Specification, PKIX Working Group, IETF, 2006. `http://www.ietf.org/html.charters/pkix-charter.html`.

[2] N. Asokan, E. Van Herreweghen, and M. Steiner. Towards a Framework for Handling Disputes in Payment Systems. In *Third USENIX Workshop on Electronic Commerce*, pages 187–202, September 1998.

[3] N. Asokan, V. Shoup, and M. Waidner. Optimistic Fair Exchange of Digital Signatures. *IEEE Journal on Selected Areas in Communications*, 18:593–610, 2000.

[4] M. Bellare, J.A. Garay, R. Hauser, A. Herzberg, H. Krawczyk, M. Steiner, E. Van Herreweghen, and M. Waidner. Design, Implementation and Deployment of the iKP Secure Electronic Payment System. In *Journal on Selected Areas in Communication, special issue on Network Security*, volume 18, pages 611–627, April 2000.

[5] S. Blake-Wilson, M. Nystrom, D. Hopwood, J. Mikkelsen, and T. Wright. Transport Layer Security (TLS) Extensions. RFC 3546, June 2003. `http://www.rfc-editor.org/rfc/rfc3546.txt`.

[6] B. Cox, J. D. Tygar, and M. Sirbu. NetBill security and Transaction Protocol. In *The First USENIX Workshop on Electronic Commerce*, pages 77–88, 1995.

[7] T. Dierks and C. Allen. The TLS Protocol — Version 1.0. RFC 2246, January 1999. `http://www.rfc-editor.org/rfc/rfc2246.txt`.

[8] Open Financial Exchange. OFX Specification, Version 2.0.2, 2004. `http://www.ofx.net/`.

[9] Financial Information eXchange (FIX). FIX Protocol, Version 4.4, 2004. `http://www.fixprotocol.org/specifications/fix4.4fixml`.

[10] Organization for the Advancement of Structured Information Standards. ebxml messaging services tc, version 3.0, 2006. `http://www.oasis-open.org/committees/documents.php?wg_abbrev=ebxml-msg`.

[11] IFX Forum Inc. Interactive Financial Exchange, Version 1.7, 2006. http://www.ifxforum.org/standards/standard/.

[12] A. O. Freier, P. Karlton, and P. C. Kocher. The SSL Protocol Version 3.0. draft-ietf-tls-ssl-version3-00.txt, November 1996. http://www1.tools.ietf.org/wg/tls/draft-ietf-tls-ssl-version3/.

[13] J. Garay and C. Pomerance. Timed Fair Exchange of Standard Signatures. In *Financial Cryptography*, volume 2742 of *LNCS*, pages 190–207. Springer-Verlag, 2003.

[14] S. Goldwasser, S. Micali, and R. Rivest. A Digital Signature Scheme Secure Against Adaptive Chosen-Message Attacks. *SIAM*, 17(2):281–308, April 1988.

[15] E. Van Herreweghen. Non-repudiation in SET: Open Issues. In *Proceedings of the 4th Conference on Financial Cryptography*, 2000.

[16] A. Herzberg and I. Yoffe. Foundations of Secure E-Commerce: The Attestation Layer. (work in progress), 2006.

[17] A. Herzberg and I. Yoffe. Layered Adversarial Specifications Framework and Applications to Secure E-Commerce. (work in progress), 2006.

[18] A. Herzberg and I. Yoffe. Layered Architecture for Secure E-Commerce Applications. In *SECRYPT'06 - International Conference on Security and Cryptography*, pages 118–125. INSTICC Press, 2006.

[19] A. Herzberg and I. Yoffe. On Secure Orders in the Presence of Faults. In *Proceedings of Secure Communication Networks (SCN)*, volume 4116 of *LNCS*, pages 126–140. Springer-Verlag, 2006.

[20] ISO/IEC 13888-1. *Information Technology - Security Techniques - Non-Repudiation - Part 1: General*. ISO/IEC, 1997.

[21] ISO/IEC 13888-2. *Information Technology - Security Techniques - Non-Repudiation - Part 2: Mechanisms using symmetric techniques*. ISO/IEC, 1998.

[22] ISO/IEC 13888-3. *Information Technology - Security Techniques - Non-Repudiation - Part 3: Mechanisms using asymmetric techniques*. ISO/IEC, 1997.

[23] S. Kremer, O. Markowitch, and J. Zhou. An Intensive Survey of Non-repudiation Protocols. *Computer Communications*, 25(17):1606–1621, November 2002.

[24] G. Lacoste, B. Pfitzmann, M. Steiner, and M. Waidner, editors. *SEMPER - Secure Electronic Marketplace for Europe*, volume 1854 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[25] O. Markowitch and S. Saeednia. Optimistic Fair Exchange with Transparent Signature Recovery. In *FC '01: Proceedings of the 5th International Conference on Financial Cryptography*, pages 339–350, London, UK, 2002. Springer-Verlag.

[26] A. Nenadic and N. Zhang. Non-repudiation and Fairness in Electronic Data Exchange. In *Proceedings of 5th International Conference on Enterprise Information Systems (ICEIS)*, pages 55–62, Angers, France, 2003.

[27] A. Nenadic, N. Zhang, B. Cheetham, and C. Goble. RSA-based Certified Delivery of E-Goods Using Verifiable and Recoverable Signature Encryption. *Journal of Universal Computer Science*, 11(1):175–192, 2005.

[28] J. B. Postel. Transmission Control Protocol. RFC 793, September 1981. http://www.rfc-editor.org/rfc/rfc793.txt.

[29] I. Ray, I. Ray, and N. Narasimhamurthy. A Fair-exchange E-commerce Protocol with Automated Dispute Resolution. In *Proceedings of the IFIP TC11/ WG11.3 Fourteenth Annual Working Conference on Database Security*, pages 27–38, Netherlands, 2001. Kluwer, B.V.

[30] SWIFT. Society for Worldwide Interbank Financial Telecommunication, 2006. http://www.swift.com/.

[31] J. Tang, A. Fu, and J. Veijalainen. Supporting Dispute Handling in E-commerce Transactions, a Framework and Related Methodologies. In *Electronic Commerce Research Journal*, volume 4, pages 393–413. Kluwer Academic, October 2004.

[32] J. Zhou. *Non-repudiation in electronic commerce*. Computer Security Series. Artech House, August 2001.

[33] J. Zhou, R. H. Deng, and F. Bao. Evolution of Fair Non-repudiation with TTP. In *ACISP '99: Proceedings of the 4th Australasian Conference on Information Security and Privacy*, pages 258–269, London, UK, 1999. Springer-Verlag.

# A Execution model; Attestation and Communication specifications

## A.1 Protocol machines , interfaces and executions

We begin by defining *protocol machines*. Protocol machines are state machines that accepts input on one of few *interface*, and produces output on one or more interfaces. The transition function maps the input, current state and random bits, to a new state and outputs on the different interfaces.

**Definition 19 (Protocol machine)** A protocol machine $\pi$ is a tuple $<\pi.S, \pi.init, \pi.I_{IN}, \pi.I_{OUT}, \pi.\delta>$ where:

1. $\pi.S$ is a set of states,
2. $\pi.init \in \pi.S$ is an initial state,
3. $\pi.I_{IN}$ is a set of input interface identifiers,
4. $\pi.I_{OUT}$ is a set of output interface identifiers,
5. $\pi.\delta : \pi.IN \rightarrow \pi.OUT$ is a transition function, with:
   - Input domain $\pi.IN = \pi.S \times \pi.I_{IN} \times \{0,1\}^* \times \{0,1\}^*$ (current state, input interface, input value, random bits).
   - Output domain $\pi.OUT = \pi.S \times \prod_{i \in \pi.I_{OUT}} \{0,1\}^*$. The outputs consist of a new state, denoted $\pi.\delta.S \in \pi.S$, and output values $\pi.\delta.ov(\iota) \in \{0,1\}^*$ for each interface $\iota \in \pi.I_{OUT}$.

We consider the case of analysis of *single protocol executions*. Single protocol executions result from the interaction of individual protocol machines , possibly spanning multiple processors, with an adversary representing the rest of the world. A single protocol execution is defined by a sequence of *rounds*, where each round consists of one invocation of the adversary $\mathcal{A}$, and then one invocation of the protocol $\pi$. We model the adversary $\mathcal{A}$ as a function from the sequence of all outputs of the protocol so far, to the next input to the protocol.

We consider multiple instances of the protocol, each running with its own state, to represent multiple processors running the protocol. At each round $i$, the adversary $\mathcal{A}$ invokes a specific *instance* (processor) $p_i$ of the protocol.

**Definition 20 (Single protocol execution)** Let $\mathbb{P}$ be a set of protocol machine processors identities. An event of protocol $\pi$ is a tuple $\xi = <p, \iota, iv, ov[\iota' \in \pi.I_{OUT}]>$, where $p \in \mathbb{P}$ is processor identity, $\iota \in \pi.I_{IN}$ is input interface, $iv \in \{0,1\}^*$ is input value, and $ov[\iota'] \in \{0,1\}^*$, for each interface $\iota' \in \pi.I_{OUT}$, are output values.

Let an *adversary* $\mathcal{A}$ for $\pi$ be a function from finite sequences of events of $\pi$, denoted $\{\xi_i\}_{i=1,\ldots,t}$, to triples $<\mathcal{A}.p, \mathcal{A}.\iota, \mathcal{A}.iv>$, where $\mathcal{A}.p \in \mathbb{P} \cup \perp$ (adversary selected processor/instance), $\mathcal{A}.\iota \in \Gamma(\mathcal{A}.p).I_{IN}$ (adversary selected input interface) and $\mathcal{A}.iv \in \{0,1\}^*$ (adversary selected input value).

Given protocol machine $\pi$, adversary $\mathcal{A}$ for $\pi$ and sequence (of random bits) $R = \{R_{i \in \mathbb{N}} \in \{0,1\}^*\}$, the *single protocol execution* of $\pi, \mathcal{A}$ and $R$, denoted $X(\pi, \mathcal{A}, R)$, is the view $\{\xi_i\}$ of $\pi$ resulting from the following process:

FOR ALL $p \in \mathbb{P}$, LET $s[p] = \pi.init$;
FOR $i = 0, 1, 2, \ldots$ DO:

1. $<p, \iota, iv>:= \mathcal{A}(\{\xi_j\}_{0<j<i})$;
2. IF $p = \perp$ THEN: $\xi_i = <\perp, \perp, \perp, \perp>$ AND **break loop**;
3. $<s[p], ov[\iota \in \pi.I_{OUT}]>:= \pi.\delta(s[p], \iota, iv, R_i)$;
4. $\xi_i = <p, \iota, iv, ov[\iota \in \pi.I_{OUT}]>$;

**Remark 3** With single protocol executions all protocol machine instances have all their interfaces connected to the adversary, and thus interact with each other only through the adversary. This simple case is of importance when analyzing multiple interacting instances of a single protocol (e.g., TCP [28], SSL/TLS [12, 7, 5]). When analyzing such protocol we typically "plug" it with an adversarial upper layer and adversarial lower (typically, communication) layer.

We use $\mathbb{X}$ to denote the set of single protocol executions (of any $\pi, \mathcal{A}$ and $R$).

**Definition 21 (Finite execution)** A single protocol execution $X$ is *finite* if $\exists\, k \in \mathbb{N}$ such that, $X = \{\xi_i\}_{i=0,\ldots,k}$ and $\xi_k = <\perp, \perp, \perp, \perp>$.

## A.2 Correctness specifications

We now proceed to define adversarial win (correctness) specifications $\mathcal{S}$ for protocol machines . First we define deterministic correctness specification, for executions of a single protocol.

**Definition 22 (Deterministic adversarial win $\mathcal{S}.AW$ predicate)** Let $\mathcal{S}.AW : \mathbb{X} \to \{\texttt{true}, \texttt{false}\}$ be a predicate over executions. Protocol $\pi$ *securely-prevents* $\mathcal{S}.AW$ *against set of adversaries* $\mathbb{A}$, if for $\mathcal{A} \in \mathbb{A}$ and every sequence $R$ of random inputs, $\mathcal{S}.AW(X(\pi, \mathcal{A}, R)) = \texttt{false}$.

Deterministic specifications are useful in benign settings, e.g. for proving tolerance to (non-malicious) failures. However, deterministic correctness specifications are hard to satisfy, against arbitrary (byzantine, malicious) adversaries. Byzantine adversaries usually have some probability of winning, which depends on the *security parameter*. We now extend our definition of specifications to allow for (limited) probability of winning by the adversary.

We now generalize the concept of specification, and allow the adversary to 'win' with some limited probability, which we denote $\mathcal{S}.\epsilon$. The adversary's winning probability may depend on the resources available to the implementation, identified by the *security parameter* $k$. For example, the security parameter $k$ for a message authentication code scheme may define the length of the output tag; the adversary may 'win' by simply guessing a correct code, with probability $2^{-k}$. To simplify notations, we assume that the security parameter is an additional input parameter to the execution (and to both adversary and protocol). Namely, we denote execution of protocol $\pi$ with adversary $\mathcal{A}$, randomness $R$ and security parameters $k$, by $X(\pi, \mathcal{A}, R, k)$.

**Definition 23 (Probabilistic adversarial win $\mathcal{S}.AW$ predicate)** Let $\mathcal{S}.AW : \mathbb{X} \to \{\texttt{true}, \texttt{false}\}$ be a predicate over executions. Protocol $\pi$ *securely-prevents* $\mathcal{S}.AW$ *against poly-time adversaries* if for every probabilistic polynomial time function $\mathcal{A}$ and every polynomial $\epsilon : \mathbb{N} \to (0, 1]$, there is a security parameter $k_0 \in 1^*$, such that for every $k \geq k_0$ holds:

$$Prob_{R=\{R_i \in_R \{0,1\}^*\}}[\mathcal{S}.AW(X(\pi, \mathcal{A}, R, k))] \leq \epsilon(k). \tag{1}$$

We also generalize the definition, to support 'concrete security', where the adversary's winning probability may depend on the resources available to the adversary, e.g. computing time. The specifications include a partially-ordered set of resources $\mathcal{S}.R$, as well as a function $\mathcal{S}.AR : \mathbb{X} \to \mathcal{S}.R$, such that $\mathcal{S}.AR(X)$ represent the resources used by the adversary in execution $X \in \mathbb{X}$.

**Definition 24 (Concrete security correctness)** A *single protocol correctness specification* $\mathcal{S}$ is a tuple
$< \mathcal{S}.AW, \mathcal{S}.R, \mathcal{S}.K, \mathcal{S}.AR, \mathcal{S}.\epsilon, \mathcal{S}.\mathcal{A} >$ where:

$\mathcal{S}.AW : \mathbb{X} \to \{\texttt{true},\texttt{false}\}$ is a predicate ('Adversary Wins') over executions,

$\mathcal{S}.R$ is a partially-ordered set (of resources),

$\mathcal{S}.AR : \mathbb{X} \to \mathcal{S}.R$ maps executions to 'adversary resources',

$\mathcal{S}.K$ is a partially-ordered set (of security parameters),

$\mathcal{S}.\epsilon : \mathcal{S}.R \times \mathcal{S}.K \to (0, 1]$ is the 'maximal error probability' function,

$\mathcal{S}.\mathcal{A} \subseteq \mathbb{A}$ is a set of adversary functions.

Protocol machine $\pi$ satisfies $\mathcal{S}$, if for every $\mathcal{A} \in \mathcal{S}.\mathcal{A}$ and every $r_{max} \in \mathcal{S}.R$, there is a security parameter $k_0 \in \mathcal{S}.K$, such that for every $k \geq k_0$ holds:

$$Prob_{R=\{R_i \in_R \{0,1\}^*\}}[\mathcal{S}.AW(x) \wedge \mathcal{S}.AR(x) \leq r_{max} | x = X(\pi, \mathcal{A}, R, k)] \leq \mathcal{S}.\epsilon(r_{max}, k). \tag{2}$$

## A.3 Modeling time and synchronization

Our definitions so far were oblivious to issues of time and synchronization. Since these issues are critical aspects of distributed systems and models, we now explain how to deal with these issues. Specifically, we focus on modeling (partially) synchronous systems.

In a synchronous system, the protocol has access to (perfectly or partially) synchronized clocks. We model this by assuming that the input from the adversary includes a designated 'time' field, e.g. $iv.time$, containing the (perfectly or partially) synchronized time. We use the shorthand notation $t(\xi)$ to refer to the time of event $\xi$ in an execution, i.e. $t(\xi) = \xi.iv.time$.

We can also use 'time' values on specific output interfaces, e.g. to request 'wake-up' clock service as required to implement time-out mechanisms.

Then, we use these values of 'time' fields as part of the specifications, specifically as part of the $\mathcal{S}.AW$ predicates. In particular, when using perfectly synchronized clocks, we require the 'time' values to be monotonously increasing, and 'wake-up' calls to occur in precisely the requested time.

To model specific or bounded communication delays, we can add appropriate restrictions on the $\mathcal{S}.AW$ predicate. Namely, 'adversary wins' only if it causes a 'bad execution', while conforming to the delay bounds.

We use additional notation of $\xi \in X[t_0, t_1]$, to denote that event's occurrence time, $t(\xi)$, was in the interval $[t_0, t_1] \subset \mathbb{R}$, or just $\xi \in X[t_0]$ to denote specific, $t(\xi) = t_0$, event time.

## A.4 Layered Specifications

For simplicity, we provide specifications and perform analysis of a single protocol at a time. However, clearly, real systems involve many protocols, often interacting to provide a complete service - most typically, in a layered architecture, where lower-layer protocols provide services to higher-layer protocols.

Consider a simple, and typical, case of modular design of a system using layered protocol architecture. In this case, we can often take advantage of the interfaces defined for each protocol, to analyze the operation of each protocol separately and draw conclusions on the use of them in a layered manner.

Specifically, consider two protocols $\pi_L$ and $\pi_U$, interacting (only) via a shared interface $\iota \in \pi_L.I \cap \pi_H.I$. Suppose the adversary win predicates of $\pi_L, \pi_H$ are of the following 'layered' form: $\mathcal{S}_H(X) = \mathcal{S}_\iota(X) \Rightarrow \mathcal{S}'_H(X)$, $\mathcal{S}_L(X) = \mathcal{S}'_L(X) \Rightarrow \mathcal{S}_\iota(X)$, where $\mathcal{S}'_L(X), \mathcal{S}'_H(X)$ do not depend on events on interface $\iota$. In a separate paper [17] we plan to prove that in a *composite execution*, where $\pi_L$ and $\pi_H$ are connected (only) via $\iota$, it holds $\mathcal{S}'_L(X) \Rightarrow \mathcal{S}'_H(X)$. Though in this manuscript, we consider single layer only (and thus do not consider composite executions), former considerations motivate defining specifications of each layer in this manner, i.e. in the form $\mathcal{S}_L.AW(X) = \mathcal{S}'_L(X) \not\Rightarrow \mathcal{S}_\iota$, where $\mathcal{S}'_L(X)$ does not depend on events on the 'higher layer' interface $\iota$.

## A.5 Communication layer specifications

First we define weak reliability specification for the communication (transport) layer. We require protocol machine processors to be connected, with respect to a given communication interval, where by connected we mean no communication failures are reported in the given interval and sent messages are acknowledged.

**Definition 25 (No communication failures indication $\mathcal{S}^{LinkOk}_{COMM}$ predicate)** Predicate $\mathcal{S}^{LinkOk}_{COMM}(X, p_0, p_1, t_0, t_1, \Delta)$ is `true` for execution $X \in \mathbb{X}$, protocol machine processors $p_0, p_1 \in \mathbb{P}$ and time interval and delay bound $t_0, t_1, \Delta \in \mathbb{R}$, if for every event $<p_0, \text{'Comm.Send'}, (p_1, msg)> \in X[t_0, t_1 - \Delta]$ the result $<p_0, \text{'Comm.SendResult'}, (p_1, msg, res)> \in X[t_1 - \Delta, t_1]$ is with output value $res \neq CommFail$.

We now make our only requirement (specification) from the communication layer. We require that if link is sustained, between protocol machine processors, see previous Definition 25, then sent messages are received (but not vice versa).

**Definition 26 (Communication layer delivery takes place $\mathcal{S}^{LINK}_{COMM}$ predicate)** Predicate $\mathcal{S}^{LINK}_{COMM}(X)$ is `true` for execution $X \in \mathbb{X}$, if $\forall\ p_0, p_1 \in \mathbb{P},\ t_0, t_1, \Delta \in \mathbb{R}$ :

$\mathcal{S}^{LinkOk}_{COMM}(X, p_0, p_1, t_0, t_1, \Delta) = \texttt{true} \Rightarrow$ for every $\zeta = <p_0, \text{'Comm.Send'}, (p_1, m)> \in X[t_0, t_1 - \Delta]$, there exists $\xi = <p_1, \text{'Comm.Receive'}, (p_0, m)> \in X[t(\zeta), t(\zeta) + \Delta]$, and (in the other direction) $\mathcal{S}^{LinkOk}_{COMM}(X, p_0, p_1, t_0, t_1, \Delta) = \texttt{true} \Rightarrow$ for every $\zeta' = <p_1, \text{'Comm.Send'}, (p_0, m)> \in X[t_0, t_1 - \Delta]$, there exists $\xi' = <p_0, \text{'Comm.Receive'}, (p_1, m)> \in X[t(\zeta'), t(\zeta') + \Delta]$.

For simplicity we, next define a global, uniform $\Delta_{comm}$ bound on delays shared by all initialized machines.

**Definition 27 (Uniform and bounded initialization $\mathcal{S}^{INIT}_{COMM}(X)$ predicate)** Predicate $\mathcal{S}^{INIT}_{COMM}(X)$ is `true` for execution $X \in \mathbb{X}$, if $\exists \Delta_{comm} \in \mathbb{R}$ s.t., for every event $<p_0, \text{'Comm.Init'}, \perp, (addr, \Delta)> \in X : \Delta = \Delta_{comm}$, where $p_0 \in \mathbb{P}$, $addr \in \{0,1\}^*$. $<p_0, \text{'Comm.Init'}, addr> \in X$ there is a following output event $<p_0, \text{'Comm.InitResult'}, \Delta> \in X : \Delta = \Delta_{comm}$, where $p_0 \in \mathbb{P}$, $addr \in \{0,1\}^*$.

## A.6 Signature Scheme

We adapt standard signature scheme definition [14], and specify signature scheme's interface.

**Definition 28 (Signature scheme)** A digital signature scheme $\mathcal{DS}$=($\mathcal{DS}.\pi$,$\mathcal{DS}.\mathsf{Verify}$) over a message space $\{0,1\}^*$, is a protocol $\mathcal{DS}.\pi$, which consists of four interfaces $\mathcal{DS}.\pi.I_{IN} = (\mathcal{DS}.\mathsf{Gen}, \mathcal{DS}.\mathsf{Sign})$, $\mathcal{DS}.\pi.I_{OUT} = (\mathcal{DS}.\mathsf{GenResult}, \mathcal{DS}.\mathsf{SignResult})$, and $\mathcal{DS}.\mathsf{Verify}$ poly-time algorithm, such that,

1. The (randomized) key generation protocol interface $\mathsf{Gen}$ takes an unary security parameter $k$ as an input, at returns a public validation key $vk$.
2. The signing protocol interface $\mathsf{Sign}$ takes as an input a message $m \in \{0,1\}^*$ and returns a pair $(m, \sigma)$ where $\sigma$ is the signature.
3. The verification algorithm $\mathsf{Verify}$ takes as an input $(vk, m, \sigma)$ tuple, where $m$ is a message, $\sigma$ is a signature and $vk$ is a validation key, and returns a value from {`true`,`false`}.

The following predicate is signature scheme's basic soundness requirement, that specifies that every signed message should pass verification.

**Definition 29 (Sound signature scheme)** Predicate $\mathcal{S}^{Sound}_{\mathcal{DS}}(X)$ is `true` for execution $X \in \mathbb{X}$ of protocol $\mathcal{DS}.\pi$ if for any processor $v \in \mathbb{P}$, message and signature $m, \sigma \in \{0,1\}^*$, and event $\zeta = <v, \text{'}\mathsf{Gen}\text{'}, (1^k, r), vk>$, $k \in \mathbb{N}, r \in \mathbb{R}$, for subsequent $\xi = <v, \text{'}\mathsf{Sign}\text{'}, m, \sigma> \in X$, where $t(\xi) > t(\zeta)$, $\mathsf{Verify}(vk, m, \sigma) = \texttt{true}$.

Next, we present an existential-forgery specification of signature scheme security. Intuitively, there is an existential forgery with respect to signature scheme's interface, if there exists a validation key output by a party and a valid signature on a message that party was never requested to sign.

**Definition 30 (Signature scheme's security)** Predicate $\mathcal{S}_{\mathcal{DS}}.AW(X(\mathcal{DS}.\pi, \mathcal{A}, R, k))$ is `true` for execution $X \in \mathbb{X}$ of protocol $\mathcal{DS}.\pi$ with adversary $\mathcal{A}$, randomness $R$ and security parameters $k$ if for any message and signature $m, \sigma \in \{0,1\}^*$, adversary $\mathcal{A}$ outputs validation key, message and signature tuple $(vk, m, \sigma)$, such that,

1. $\mathsf{Verify}(vk, m, \sigma) = \texttt{true}$ (respective signature passes verification with validation key), and
2. Let $<v, \text{'}\mathsf{Gen}\text{'}, (1^k, r), vk> \notin X$, (no key generation event which had output respective $vk$ validation key, for $v$ party),
3. $<v, \text{'}\mathsf{Sign}\text{'}, m, \sigma> \notin X$ (party never signed the respective message $m$),

where length of $r, vk, m$ and $\sigma$ is bounded by $k^c$ for some constant $c > 0$.

## A.7 Attestation layer specifications

### A.7.1 Initialization specifications

Initialized attestation protocol machines are protocol machines where the interface identifier *'Att.Init'* (see Table 3) was invoked. Furthermore, protocol machines which has valid specified role in the execution are protocol machines where the attestation channel was opened using aforementioned role with an attestation agreement that includes machine's correct identity. However, we first assume, for simplicity, that all protocol machines, use the same uniform bound $\Delta_{att}$ on delays.

**Definition 31 (Uniform Initialization $\mathcal{S}_{ATT}^{INIT-U}$ predicate)** Predicate $\mathcal{S}_{ATT}^{INIT-U}(X)$ is `true` for execution $X \in \mathbb{X}$, if $\exists \Delta_{att} \in \mathbb{R}$, s.t., for all $p \in \mathbb{P}$, $<p, \text{'Att.InitResult'}, (vk, \Delta)> \in X : \Delta = \Delta_{att}$, where $vk \in \{0,1\}^*$.

Next we define *properly initialized* protocol machine processor, as a processor for which an execution contains initialization events, and successful initialization result, after a bounded time, where the bound, is for simplicity, the previous $\Delta_{att}$ delay bound.

**Definition 32 (Bounded initialization $\mathcal{S}_{ATT}^{INIT-B}$ predicate)** Predicate $\mathcal{S}_{ATT}^{INIT-B}(X)$ is `true` for execution $X \in \mathbb{X}$, if for every event

$$\xi =< p, \text{'Att.Init'}, (1^k, r, addr) > \in X,$$

there is an event

$$< p, \text{'Att.InitResult'}, (vk, \Delta) > \in X[t(\xi), t(\xi) + \Delta_{att}],$$

where $p \in \mathbb{P}$ and $vk, addr, r \in \{0,1\}^*$ and $\Delta \in \mathbb{R}$. In that case we also say that in execution $X$, protocol machine $p$ is *properly initialized for attestation* with address $addr$ and validation key $vk$.

Next, denote by $\mathcal{AGR}^{ATT}$ the domain of possible attestation agreements. We define that protocol machine processor has opened a valid attestation channel if it was properly initialized, and had returned successful channel open indication when was supplied an attestation agreement which includes correct processor's identity (generated at the initialization).

**Definition 33 (Valid attestation role *Att.ValidOpen* predicate)** Let $X \in \mathbb{X}$ be an execution, $AttAgr \in \mathcal{AGR}^{ATT}$ be an attestation agreement, $p \in \mathbb{P}$ a protocol machine processor and $\rho \in \{\text{'S'}, \text{'C'}, \text{'N'}\}$ processor's role. Predicate $Att.ValidOpen(X, AttAgr, p, \rho)$ is `true` if $p$ is *properly initialized for attestation* with address $addr \in \{0,1\}^*$, validation key $vk \in \{0,1\}^*$, and for every event

$$\xi =< p, \text{'Att.OpenChannel'}, (AttAgr, \rho) > \in X,$$

where $AttAgr.\rho = (vk, addr)$ (correct identity was used), there was an event

$$< p, \text{'Att.OpenChannelResult'}, \texttt{true} > \in X[t(\xi), t(\xi) + \Delta_{att}]$$

**Remark 4** For simplicity, we do not allow adversarial 'break-in' in the middle of an execution. Thus protocol machine processors which had opened a valid attestation channel (Definition 33) are not adversarial with respect to that channel.

We next define an additional liveness predicate, to bound the time of opening an attestation channel,

**Definition 34 (Bounded open channel $\mathcal{S}_{ATT}^{BOC}$ predicate)** Predicate $\mathcal{S}_{ATT}^{BOC}(X)$ is `true` for execution $X \in \mathbb{X}$, if for every event

$$\xi =< p, \text{'Att.OpenChannel'}, (AttAgr, \rho) > \in X,$$

there was an event

$$< p, \text{'Att.OpenChannelResult'}, res > \in X[t(\xi), t(\xi) + \Delta_{att}],$$

where $p \in \mathbb{P}, AttAgr \in \mathcal{AGR}^{ATT}, res \in \{0,1\}^*, \rho \in \{\text{'S'}, \text{'C'}, \text{'N'}\}$.

The next predicate combines previous initialization predicates.

**Definition 35 (Initialization $\mathcal{S}_{ATT}^{INIT}$ predicate)** For execution $X \in \mathbb{X}$,

$$\mathcal{S}_{ATT}^{INIT}(X) \equiv \mathcal{S}_{ATT}^{INIT-B}(X) \wedge \mathcal{S}_{ATT}^{INIT-U}(X) \wedge \mathcal{S}_{ATT}^{BOC}(X)$$

### A.7.2 Liveness specifications

Similarly to communication layer, we define indication for non-failing channels,

**Definition 36 (No attestation failures indication $\mathcal{S}_{ATT}^{LinkOk}$ predicate)** Predicate $\mathcal{S}_{ATT}^{LinkOk}(X, AttAgr, t_0, t_1, \Delta, \text{'}C\text{'})$ is `true` ('no attestation failures for client role') for execution $X \in \mathbb{X}$, attestation agreement $AttAgr \in \mathcal{AGR}^{ATT}$, interval $t_0, t_1 \in \mathbb{R}$, and delay bound $\Delta \in \mathbb{R}$ if there were no events $<c, \text{'}Att.SendResult\text{'}, (AttAgr, e)> \in X[t_0, t_1 + \Delta]$ for $c \in \mathbb{P} : c = AttAgr.C$, with failure indication, i.e., $e = CommFail$.

Similarly, predicate $\mathcal{S}_{ATT}^{LinkOk}(X, AttAgr, t_0, t_1, \text{'}S\text{'})$ is `true` ('no attestation failures for server role') if there were no events $<s, \text{'}Att.Receive\text{'}, (AttAgr, e)> \in X[t_0, t_1 + \Delta]$ for $s \in \mathbb{P} : s = AttAgr.S$, with failure indication, i.e., $e = CommFail$.

Having defined indication for connected link we now define a specification regarding the meaning of a connected link. Since attestation channel involves three parties, we define pairwise specifications. The first predicate asserts, if honest client and notary are connected then client always receives evidences (EOD or EOFS) for messages it sends.

**Definition 37 (No attestation failures between notary and client $\mathcal{S}_{ATT}^{LINK:C,N}$ predicate)** Predicate $\mathcal{S}_{ATT}^{LINK:C,N}(X, AttAgr, t_0, t_1, \Delta)$ is `true` for execution $X \in \mathbb{X}$, attestation agreement $AttAgr \in \mathcal{AGR}^{ATT}$, time interval $t_0, t_1 \in \mathbb{R}$ and delay bound $\Delta \in \mathbb{R}$, if for $c = AttAgr.C$, and every $msg \in \{0,1\}^*$ and $\zeta = <c, \text{'}Att.Send\text{'}, (AttAgr, msg)> \in X[t_0, t_1]$ there is a corresponding event $<c, \text{'}Att.SendResult\text{'}, (AttAgr, e)> \in X[t(\zeta), t(\zeta) + \Delta]$ with non-failing result, i.e., $e.type \in \{EOD, EOFS\}$ and $Att.Validate(AttAgr, e) =$ `true`.

Next predicate asserts that for a connected attestation link for a server an EOFS could not be obtained[4] (for messages sent to that server).

**Definition 38 (No attestation failures between notary and server $\mathcal{S}_{ATT}^{LINK:S,N}$ predicate)** Predicate $\mathcal{S}_{ATT}^{LINK:S,N}(X, AttAgr, t_0, t_1, \Delta)$ is `true` for execution $X \in \mathbb{X}$, attestation agreement $AttAgr \in \mathcal{AGR}^{ATT}$, time interval $t_0, t_1 \in \mathbb{R}$ and delay bound $\Delta \in \mathbb{R}$, if there does not exists evidence $e$ output by adversary $\mathcal{A}$, where

1. $Att.Validate(AttAgr, e)$ is `true`, and
2. $e.type = EOFS$, and
3. $e.ctime \in [t_0, t_1 + \Delta]$

The following predicate combines the previous two predicates, with attestation failure indication, and asserts that if no failure was indicated (by an adversary) then there are evidence for sent messages, and if pairwise links are sustained, sent messages are delivered.

**Definition 39 (Attestation layer link liveness $\mathcal{S}_{ATT}^{LINK}$ predicate)** Predicate $\mathcal{S}_{ATT}^{LINK}$ is `true` for execution $X \in \mathbb{X}$, if $\forall\, AttAgr \in \mathcal{AGR}^{ATT}$, $t_0, t_1, \Delta \in \mathbb{R}$ :

**Case I**
1. $c \in \mathbb{P}$, $Att.ValidOpen(X, AttAgr, c, \text{'}C\text{'})$ is `true`, and
2. $n \in \mathbb{P}$, $Att.ValidOpen(X, AttAgr, n, \text{'}N\text{'})$ is `true`, and
3. $\mathcal{S}_{ATT}^{LinkOk}(X, AttAgr, t_0, t_1, \Delta, \text{'}C\text{'}) =$ `true` $\Rightarrow \mathcal{S}_{ATT}^{LINK:C,N}(X, AttAgr, t_0, t_1, \Delta)$

**Case II**
1. $s \in \mathbb{P}$, $Att.ValidOpen(X, AttAgr, s, \text{'}S\text{'})$ is `true`, and
2. $n \in \mathbb{P}$, $Att.ValidOpen(X, AttAgr, n, \text{'}N\text{'})$ is `true`, and
3. $\mathcal{S}_{ATT}^{LinkOk}(X, AttAgr, t_0, t_1, \Delta, \text{'}S\text{'}) =$ `true` $\Rightarrow \mathcal{S}_{ATT}^{LINK:S,N}(X, AttAgr, t_0, t_1, \Delta)$

**Case III**
1. $c \in \mathbb{P}$, $Att.ValidOpen(X, AttAgr, c, \text{'}C\text{'})$ is `true`, and
2. $s \in \mathbb{P}$, $Att.ValidOpen(X, AttAgr, s, \text{'}S\text{'})$ is `true`, and
3. $n \in \mathbb{P}$, $Att.ValidOpen(X, AttAgr, n, \text{'}N\text{'})$ is `true`, and
4. $\mathcal{S}_{ATT}^{LinkOk}(X, AttAgr, t_0, t_1, \Delta, \text{'}C\text{'}) \wedge \mathcal{S}_{ATT}^{LinkOk}(X, AttAgr, t_0, t_1, \Delta, \text{'}S\text{'}) \Rightarrow$

[4] Also, see Definition 44.

(a) $\mathcal{S}_{ATT}^{LINK:S,N}(X, AttAgr, t_0, t_1, \Delta) = \texttt{true}$, and
(b) $\mathcal{S}_{ATT}^{LINK:C,N}(X, AttAgr, t_0, t_1, \Delta) = \texttt{true}$, and
(c) for every $\zeta = <c, \text{'Att.Send'}, (AttAgr, msg)> \in X[t_0, t_1]$ there is $<s, \text{'Att.Receive'}, (AttAgr, e)> \in X[t(\zeta), t(\zeta) + \Delta]$, where $e.type = EOO$ and $e.msg = msg$.

**Remark 5** We are not to consider cases when the notary was not in a valid role. For example, previous Definition 39 Case III, could be redefined considering honest client and server only, having a sustained link. In that case, we could expect the (honest) parties to fairly exchange EOO for EOD, however, such requirement would limit and force an optimistic implementation of attestation layer, where parties first try to obtain evidences for messages directly from one another.

**Remark 6** Notice that from joining properties 4a–4b, it follows, by previous Definitions 37–38, that not only a sent message is delivered with an EOO, but that the client would obtain an EOD for the message: $<c, \text{'Att.SendResult'}, (AttAgr, e)> \in X[t(\zeta), t(\zeta) + \Delta]$, where $e.msg = msg$ and $e.type = EOD$.

### A.7.3 Correctness specifications

We begin by defining two predicates to identify delivery of invalid evidences to upper layer.

**Definition 40 (Invalid receive $\mathcal{S}_{ATT}^{I\text{-}Recv}$ predicate)** Predicate $\mathcal{S}_{ATT}^{I\text{-}Recv}(X)$ is `true` for execution $X \in \mathbb{X}$, if there exists processor $p \in \mathbb{P}$, attestation agreement $AttAgr \in \mathcal{AGR}^{ATT}$, and event $<p, \text{'Att.Receive'}, (AttAgr, e)> \in X$, s.t.,

$$Att.ValidOpen(X, AttAgr, p, \text{'S'}) = \texttt{true} \wedge Att.Validate(AttAgr, e) = \texttt{false}$$

**Definition 41 (Invalid send result $\mathcal{S}_{ATT}^{I\text{-}Send}$ predicate)** Predicate $\mathcal{S}_{ATT}^{I\text{-}Send}(X)$ is `true` for execution $X \in \mathbb{X}$, if there exists processor $p \in \mathbb{P}$, attestation agreement $AttAgr \in \mathcal{AGR}^{ATT}$ and event $<p, \text{'Att.SendResult'}, (AttAgr, e)> \in X$, where $e \neq CommFail$, s.t.,

$$Att.ValidOpen(X, AttAgr, p, \text{'C'}) = \texttt{true} \wedge Att.Validate(AttAgr, e) = \texttt{false}$$

We now define adversarial win predicates on the attestation layer interfaces. The predicates would define whether the attestation interfaces exhibited **incorrect** behavior with respect to an execution. The first predicate describes a fake EOO evidence, where we require a valid evidence, and valid client, but the message described by the EOO was not sent.

**Definition 42 (Forging evidence of origin $\mathcal{S}_{ATT}^{F-EOO}$ predicate)** Predicate $\mathcal{S}_{ATT}^{F-EOO}(X)$ is `true` for execution $X \in \mathbb{X}$ , if an adversary $\mathcal{A}$ outputs an attestation agreement and evidence, $(AttAgr, e)$, s.t.,

1. $Att.Validate(AttAgr, e)$ is `true`, and
2. $e.type = EOO$, and
3. $\exists c \in \mathbb{P}$, $Att.ValidOpen(X, AttAgr, c, \text{'C'})$ is `true`, and
4. $<c, \text{'Send'}, (AttAgr, e.msg)> \notin X[e.ctime]$.

The next definition would define when EOD evidence, is considered fake. The evidence is fake if no message described by the EOD was received, by a valid server.

**Definition 43 (Forging evidence of delivery $\mathcal{S}_{ATT}^{F-EOD}$ predicate)** Predicate $\mathcal{S}_{ATT}^{F-EOD}(X)$ is `true` for execution $X \in \mathbb{X}$, if an adversary $\mathcal{A}$ outputs an attestation agreement and evidence $(AttAgr, e)$, s.t.,

1. $Att.Validate(AttAgr, e)$ is `true`, and
2. $e.type = EOD$, and
3. $\exists s \in \mathbb{P}$, $Att.ValidOpen(X, AttAgr, s, \text{'S'})$, and
4. $<s, \text{'Receive'}, (AttAgr, e')> \notin X[e.ctime]$, where $e'.msg = e.msg$, and $e'.type = EOO$.

Next we define when invalid EOFS conditions are attained. The evidence is invalid when both server and notary are valid and communication link is sustained between the server and the notary, however, adversary is able to show valid, notary signed, evidence of failed message delivery to the server.

**Definition 44 (Forging evidence of failure and submission $\mathcal{S}_{ATT}^{F-EOFS}$ predicate)** Predicate $\mathcal{S}_{ATT}^{F-EOFS}(X)$ is `true` for execution $X \in \mathbb{X}$, if an adversary $\mathcal{A}$ outputs an attestation agreement and evidence $(AttAgr, e)$, s.t.,

1. $Att.Validate(AttAgr, e)$ is `true`, and
2. $e.type = EOFS$, and
3. $\exists s \in \mathbb{P}$, $Att.ValidOpen(X, AttAgr, s, \text{'}S\text{'})$ is `true`, and
4. $\exists n \in \mathbb{P}$, $Att.ValidOpen(X, AttAgr, n, \text{'}N\text{'})$ is `true`, and
5. $\mathcal{S}_{COMM}^{LinkOk}(X, s, n, e.ctime - \Delta_{att}, e.ctime, \Delta_{comm})$ is `true`.

We now combine the above predicates to define adversarial win specification for attestation layer (informally, as $\mathcal{S}_{COMM} \not\Rightarrow \mathcal{S}_{ATT}$).

**Definition 45 (Adversarial win $\mathcal{S}_{ATT}.AW$ predicate)** Let $\mathcal{S}_{ATT}.AW(X)$ be attestation layer adversarial win specification for execution $X \in \mathbb{X}$,

$$\mathcal{S}_{ATT}.AW(X) \equiv \mathcal{S}_{COMM}^{LINK}(X) \wedge \mathcal{S}_{COMM}^{INIT}(X) \wedge (\mathcal{S}_{ATT}^{I\text{-}Recv}(X) \vee \\ \vee \mathcal{S}_{ATT}^{I\text{-}Send}(X) \vee \mathcal{S}_{ATT}^{F-EOO}(X) \vee \mathcal{S}_{ATT}^{F-EOD}(X) \vee \mathcal{S}_{ATT}^{F-EOFS}(X))$$