Enterprise Ethereum Alliance Off-Chain Trusted Compute Specification V1.0

13 May 2019

Editors:

Sanjay Bakshi (Intel)

Yevgeniy (Eugene) Yarmosh (Intel)

Lei Zhang (iExec Blockchain Tech)

Contributors:

Bill Gleim (ConsenSys), Andreas Freund (ConsenSys), Thomas Bertani (Oraclize), Jean-Charles Cabelguen (iExec), Ben Towne (SAE ITC), Dan von Kohorn (ConsenSys / Independent), George Polzer (Everymans AI), Puneetha Karamsetty (Web3Labs), Mic Bowman (Intel), Michael Steiner (Intel), Bruno Vavala (Intel), Tom Willis (Intel), Junji Katto (Itau), Przemyslaw Jakub Siemion (Santander Digital), Chaals Nevile (EEA)

Abstract

This document specifies APIs that enable off-chain Trusted Computing for Enterprise Ethereum. In this release, The Trusted Computing specification enables privacy in blockchain translations, moving intensive processing from a main blockchain to improve scalability and latency, and support of attested Oracles.

1. Legal Notice

The copyright in this document is owned by Enterprise Ethereum Alliance. Inc. (“EEA” or “Enterprise Ethereum Alliance”).

No modifications, edits or changes to the information in this document are permitted. Subject to the terms and conditions described herein, this document may be duplicated for internal use, provided that all copies contain all proprietary notices and disclaimers included herein. Except as otherwise provided herein, no license, express or implied, by estoppel or otherwise, to any intellectual property rights are granted herein.

Use of this document and any related intellectual property incorporated herein, is also governed by the Bylaws, Intellectual Property Rights Policy and other governing documents and policies of EEA and is subject to the disclaimers and limitations described below.

No use or display of any of the following names or marks "Enterprise Ethereum Alliance", the acronym "EEA", the EEA logo, or any combination thereof, to claim compliance with or conformance to this document (or similar statements) is permitted absent EEA membership and express written permission from the EEA. The EEA is in process of developing a compliance testing and certification program only for the EEA members in good standing, which it expects to launch in second half of 2020.

THE CONTENTS OF THIS DOCUMENT ARE PROVIDED "AS IS" WITH NO WARRANTIES WHATSOEVER, INCLUDING ANY WARRANTY OF MERCHANTABILITY, NONINFRINGEMENT, FITNESS FOR ANY PARTICULAR PURPOSE, SATISFACTORY QUALITY, OR REASONABLE SKILL OR CARE, OR ANY WARRANTY ARISING OUT OF ANY COURSE OF DEALING, USAGE, TRADE PRACTICE, PROPOSAL, SPECIFICATION OR SAMPLE. EEA DOES NOT WARRANT THAT THIS DOCUMENT IS COMPLETE OR WITHOUT ERROR AND DISCLAIMS ANY WARRANTIES TO THE CONTRARY.

Each user of this document hereby acknowledges that software or products implementing the technology specified in this document ("EEA-Compliant Products") may be subject to various regulatory controls under the laws and regulations of various governments worldwide. Such laws and regulatory controls may govern, among other things, the combination, operation, use, implementation and distribution of EEA-Compliant Products. Examples of such laws and regulatory controls include, but are not limited to, airline regulatory controls, telecommunications regulations, finance industry and security regulations, technology transfer controls, health and safety and other types of regulations. Each user of this document is solely responsible for the compliance by their EEA-Compliant Products with any such laws and regulations and for obtaining any and all required authorizations, permits, or licenses for their EEA-Compliant Products related to such regulations within the applicable jurisdictions. Each user of this document acknowledges that nothing in this document or the relevant specification provides any information or assistance in connection with securing such compliance, authorizations or licenses. NOTHING IN THIS DOCUMENT CREATES ANY WARRANTIES WHATSOEVER REGARDING THE APPLICABILITY OR NON-APPLICABILITY OF ANY SUCH LAWS OR REGULATIONS OR THE SUITABILITY OR NON-SUITABILITY OF ANY SUCH PRODUCT OR SERVICE FOR USE IN ANY JURISDICTION.

EEA has not investigated or made an independent determination regarding title or noninfringement of any technologies that may be incorporated, described or referenced in this document. Use of this document or implementation of any technologies described or referenced herein may therefore

infringe undisclosed third-party patent rights or other intellectual property rights. The user is solely responsible for making all assessments relating to title and noninfringement of any technology, standard, or specification referenced in this document and for obtaining appropriate authorization to use such technologies, standards, and specifications, including through the payment of any required license fees.

NOTHING IN THIS DOCUMENT CREATES ANY WARRANTIES OF TITLE OR NONINFRINGEMENT WITH RESPECT TO ANY TECHNOLOGIES, STANDARDS OR SPECIFICATIONS REFERENCED OR INCORPORATED INTO THIS DOCUMENT.

IN NO EVENT SHALL EEA OR ANY OF ITS MEMBERS BE LIABLE TO THE USER OR TO A THIRD PARTY FOR ANY CLAIM ARISING FROM OR RELATING TO THE USE OF THIS DOCUMENT, INCLUDING, WITHOUT LIMITATION, A CLAIM THAT SUCH USE INFRINGES A THIRD PARTY'S INTELLECTUAL PROPERTY RIGHTS OR THAT IT FAILS TO COMPLY WITH APPLICABLE LAWS OR REGULATIONS. BY USE OF THIS DOCUMENT, THE USER WAIVES ANY SUCH CLAIM AGAINST EEA AND ITS MEMBERS RELATING TO THE USE OF THIS DOCUMENT.

EEA reserves the right to adopt any changes or alterations to this document as it deems necessary or appropriate without any notice. User is solely responsible for determining whether this document has been superseded by a later version or a different document.

Status of This Document

This section describes the status of this document at the time of its publication. Newer documents might supersede this document.

This document has been reviewed by the EEA Membership, Executive and Board, and is endorsed by the EEA Board for publication as an experimental API. It is a stable document and may be used as reference material or cited from another document.

This specification was developed by the EEA Technical Specification Working Group, Trusted Computing Task Force for review, improvement, and publication as an EEA Standard.

Please send any comments to the EEA Technical Steering Committee at https://entethalliance.org/contact/.

GitHub Issues are preferred for discussion of this specification.

Table of Contents

B.2 Informative references

2. Introduction

This section is non-normative.

This specification has four objectives:

- Support private transactions on a blockchain between mutually-untrusting parties without disclosing transaction details to other parties who also have access to the blockchain.
- Support disclosure of selected information to chosen parties on a blockchain, while maintaining the secrecy of other information from those same chosen parties ("selective Privacy").
- Move intensive processing from a main blockchain to an off-chain Trusted Compute capability thereby improving throughput and scalability.
- Support Attested Oracles.

These objectives are achieved by executing some parts of a blockchain transaction off the main chain in off-chain trusted computing. There are currently three types of Trusted Compute that are supported by this specification:

- Trusted Execution Environments (Hardware based)
- Zero-Knowledge Proofs (Software based)
- Trusted Multi-Party-Compute (MPC) (Software/Hardware based)

The APIs are grouped in registration, invocation and receipt handing sections. Attested Oracles are considered a special application of Trusted Compute used to create increased trust in an Oracle, and can be implemented using the defined APIs.

2.1 Background

Early blockchains delivered computational trust via massive replication but had limited throughput, and imperfect privacy and security. Adding trusted off-chain execution to a blockchain improves blockchain performance in these areas. In this specification, a main blockchain maintains a single authoritative instance of the objects, enforces execution policies, and ensures that transactions and results can be audited, while associated off-chain trusted computing allows greater throughput, increases Work Order integrity, and protects data confidentiality.

For terminology used in this specification please refer to the terminology section.

Figure 1 depicts an example Enterprise Ethereum blockchain with N member enterprises. Each enterprise has Requesters, an Ethereum blockchain client and one or more Workers (supported by a Worker Service). Requesters submit Work Orders, and Workers execute those Work Orders. Work Order receipts can be recorded on the blockchain by Ethereum clients running Smart Contracts. While each of the enterprises in figure 1 contains all three major components, this is not necessary. For example, Requesters from Enterprise 1 may send Work Orders to a Worker at Enterprise 2, and the results may be recorded by an Ethereum Client at Enterprise 1. Accessing resources across multiple enterprises increases network resilience, allows more efficient use of resources, and provides access to greater total capacity than most individual enterprises can afford.

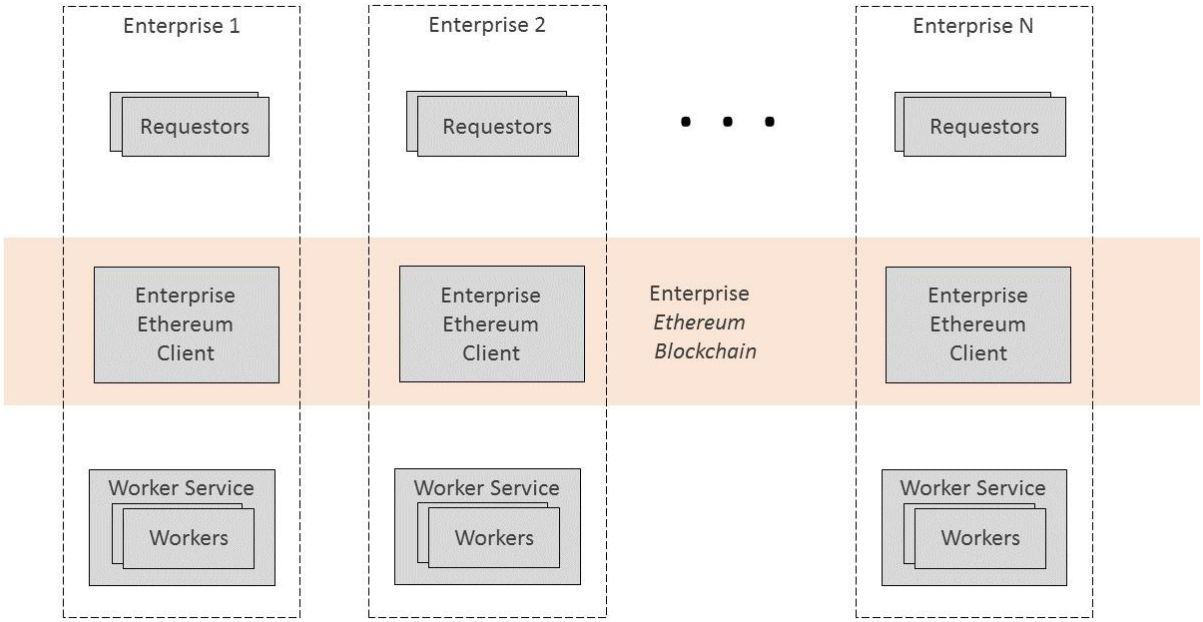

Figure 1 Figure 1 Enterprise Ethereum Blockchain with Off-Chain Workers

In order to get these benefits of cooperation, participating Enterprises must register their Requesters, Ethereum Clients, and Workers with the main blockchain. Each registered Requester, Ethereum Client, or Worker (or its Worker Service) will have its own unique Ethereum address (or a DID that can be resolved to an Ethereum address) from which to receive or send transactions.

2.1.1 Invocation Models

A Requester can submit a Work Order to a Worker via one of the following models:

2.1.1.1 Direct Model

In this invocation, model Workers are invoked via a JSON RPC network API. An organization registers its Workers with on-chain Smart Contract(s) where a ĐApp can discover them. Subsequent interactions between a ĐApp and the Worker are done off the chain. Optionally, transaction receipts can be stored on the chain.

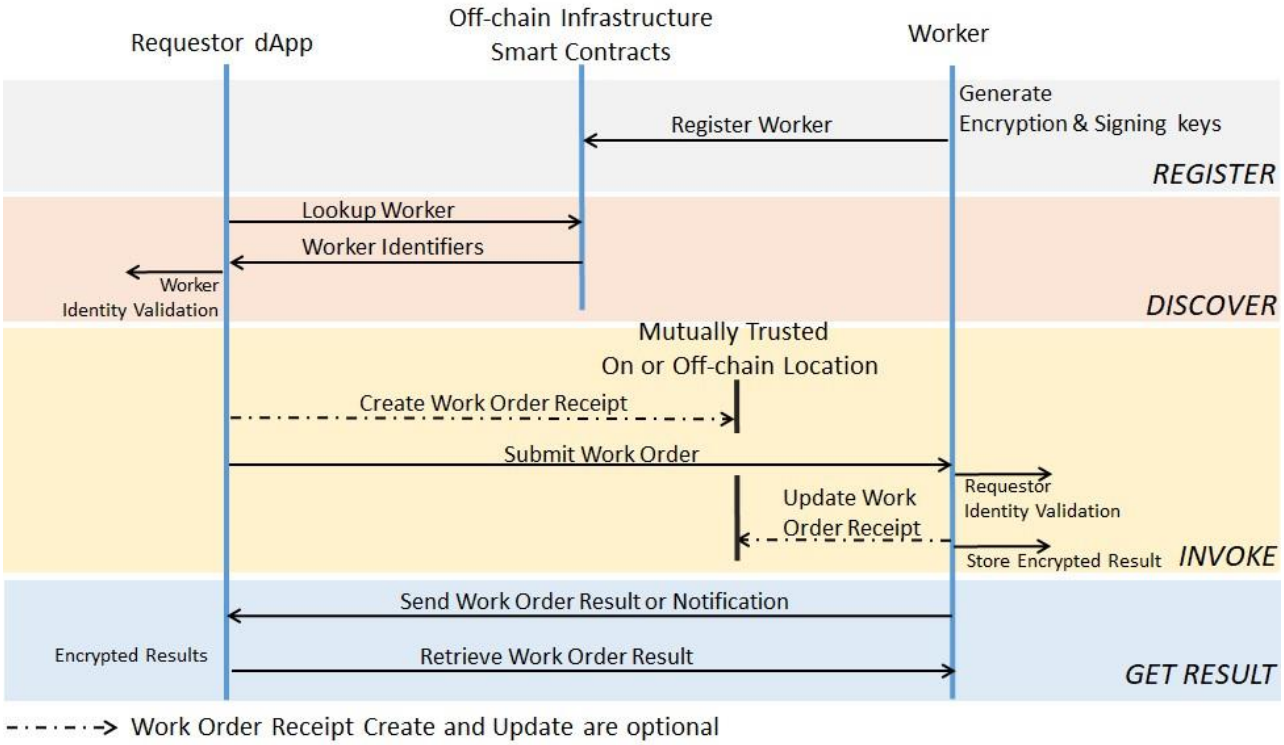

Figure 2 Figure 2 Direct Invocation

2.1.1.2 Proxy Model

In this invocation, model Workers are invoked via a proxy Smart Contract (see section Proxy Model Invocation). The Proxy Model is typically used to support uses in which an application Smart Contract or a ĐApp prefers not to or cannot invoke a Worker directly. An organization registers its Workers with an on-chain Smart Contract where a ĐApp or other smart contract can discover them. Optionally, transaction receipts can be stored on the chain.

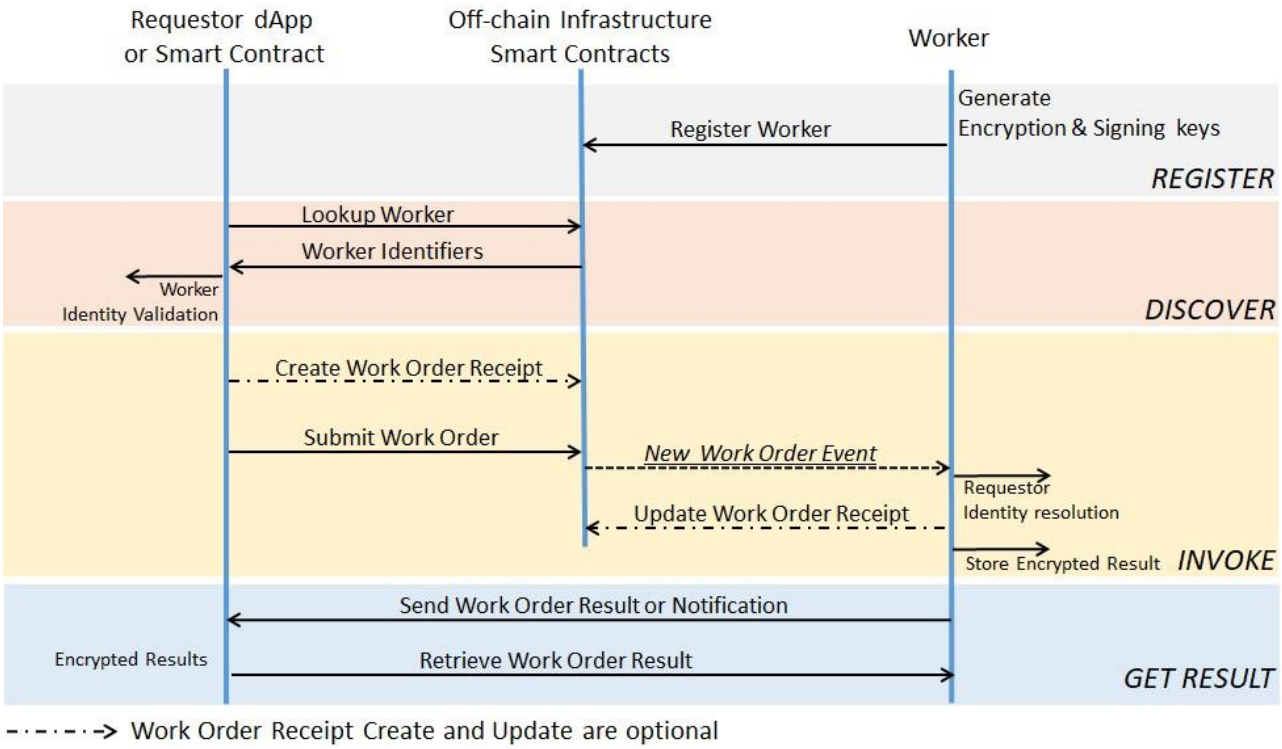

Figure 3 Figure 3 Proxy Invocation

This version of the specification addresses only stateless (from the Worker viewpoint) execution. Maintaining state between Work Order invocations has to be managed by the caller.

A future version of the specification may include an additional model, in which off-chain logic acts as both the Requester and the Worker and is the creator and controller of the Smart Contract. The Smart Contract is branded by its creator and maintains the state of the contract. Logic within the Smart Contract is minimally used for validating and enforcing security policies for state changes and local transactions. This version would rely on an external registry shared by contract participants.

3. Conformance

As well as sections marked as non-normative, all authoring guidelines, diagrams, examples, and notes in this specification are non-normative. Everything else in this specification is normative.

The key word *MUST* is to be interpreted as described in [RFC2119].

This Specification extends the capabilities and interfaces of the Enterprise Ethereum Alliance Client Specification, version 2.

The Application Programming Interfaces (APIs), JSON-RPC formats and parameters, "Smart Contract" functions and events described in this Specification are *experimental*. Experimental means that a requirement or API may change as implementation feedback is incorporated. Implementors are encouraged to implement these experimental requirements, with the knowledge that requirements in future versions of the Specification are not guaranteed to be compatible with the current version. Please send your comments and feedback on the experimental portions of this Specification to the EEA Technical Steering Committee at https://entethalliance.org/contact/.

4. Design Assumptions

This section is non-normative.

The APIs in this specification assume the existence of :

- A Worker Registry smart contract or an Off-Chain Worker Registry JSON RPC API for registering various Workers as a part of a deployment.
- An optional Work Order Receipts smart contract and/or support for Work Order Receipts JSON RPC API for Requesters to create Work Order receipts and for Workers to update Work Order receipts upon completion of execution.

RPC APIs in the current version of the specification use [JSON-RPC-API]. Future versions of the specification may support other mechanisms such as protobuf and gRPC.

For the Proxy Model, the APIs additionally assume the existence of a Work Order Invocation Proxy smart contract (deployed by a Worker Service) for sending work orders as defined in section Proxy Model Invocation. It is used by a ĐApp or an enterprise application smart contract to invoke Work Order execution in a Worker.

For the Direct Model, Workers support the Work Order Execution JSON RPC API to receive Work Orders from ĐApps.

The APIs assume that a Worker:

- Has an RSA [RFC8017] and potentially an ECDSA/SECP256K1 [secp256k1] public-private key pair, that can be used to encrypt data (e.g. a one-time symmetric session key) and to create digital signatures.
- Can publish its public keys.
- Never reveals its private keys.
- Provides proof data that defines and attests what mechanisms and capabilities are used to ensure Worker execution integrity, Requester's privacy, and data confidentiality.

This specification assumes, but doesn't define, a permissioning mechanism that would authorize access to defined APIs. It assumes that implementation-specific policies will be implemented and enforced by the Worker Registry smart contract, optional Requester Registry smart contract, optional Work Order Invocation Proxy smart, and/or optional Work Order Receipts smart contract.

The Delegate Model assumes a shared registry of workers and assumes that the associated repository for the registered packaging, binary (.jar, .dll, etc.), Docker image, or other is available to parties participating in the contracts using them.

5. RPC Encoding Conventions

All RPC APIs in this specification follow JSON RPC conventions: https://www.jsonrpc.org/specification.

JSON RPC payloads include the following common features applicable to all APIs:

- jsonrpc *MUST* be 2.0 as defined in [JSON-RPC-API] id is an id used to link
- request and response as described in [JSON-RPC-API]

5.1 Error and Status Formats

All errors and status are returned in the following generic JSON RPC error format.

```
{
  "jsonrpc": "2.0",   // as per JSON RPC spec
  "id": <integer>,   // the same as in input
  "error": {     // as per JSON RPC spec
    "code": <integer>,
    "message": <string>,
    "data": <implementation specific data>
  }
}
```

5.1.1 Parameters jsonrpc must be 2.0 per JSON RPC specification id is the same id that was

sent in a corresponding request error is a JSON object that defines an error or error status,

with the following parameters: code is an integer number defining an error or state.

Supported values:

- Values from -32768 to -32000 are reserved for pre-defined errors in JSON RPC spec
- 0 – success
- 1 – unknown error
- 2 – invalid parameter format or value
- 3 – access denied
- 4 – invalid signature
- 5 - no more look up results
- 6 - unsupported mode (e.g. synchronous, asynchronous, pull, or notification) message

is a string describing the errors and corresponding to code value. data contains additional details about the error. Its format is error and implementation specific.

5.2 JWT Signature Support

The APIs also support JSON Web Token [RFC7519] as an option for the signatures.

Header:

```
{
  "alg": "RSA" or "secp256k1"
  "type": "JWT"
}
```

Payload:

```
{
  "apiSpecific": <string>
  //...
}
```

Signature:

```
RSA or SECP256K1(
```

```
    base64UrlEncode(header) + "." +
base64UrlEncode(payload),      secret)
```

Where secret is a random nonce.

The parameter descriptions for APIs in this specification only specify API-dependent Payload objects.

6. Worker APIs

This chapter defines APIs for registering Workers, Worker Registries and Worker Registry Lists:

- Smart contract API to list worker registries, defined in Worker Registry List Smart Contract API.
- Smart contract API for maintaining worker registry on-chain, defined in Worker Registry Smart Contract API.
- JSON RPC API for maintaining worker registry off-chain, defined in Off-Chain Worker Registry JSON RPC API.

6.1 Worker Registry List Smart Contract API

APIs in this section are implemented as an Ethereum smart contract referred to as Worker Registry List.

An implementation-specific model will be used to enforce authorization policies for access to these APIs.

As mentioned in Registering a New Worker section, currently three workers types are defined based on trusted compute type attributes. In future, more attributes such as location, support for some special hardware or software feature, etc. will be defined. The Worker Registry Smart Contract allows attribute-based grouping of workers or grouping based on an organization e.g. those belonging to a single bank. The Worker Registry List Smart Contract API allows discovery of various worker registries.

6.1.1 Adding a New Registry

This function adds a new registry to the registry list.

Inputs

orgID identifies an organization that hosts the registry, e.g. a bank in the consortium, or an anonymous entity.

uri defines a URI for this registry that supports Off-Chain Worker Registry JSON RPC API.

scAddr defines an Ethereum address for the Worker Registry Smart Contract API smart contract.

appTypesIds is an optional parameter that defines application types supported by the worker.

```
function registryAdd(byte32 orgID, string uri, byte32 scAddr, bytes32[] a
```

6.1.2 Updating a Registry

This function updates a registry to the registry list.

Inputs and outputs are the same as for the function registryAdd() above.

```
function registryUpdate(byte32 orgID, string uri, byte32 scAddr, bytes32[
```

6.1.3 Setting Registry Status

This function sets a Registry's status.

Inputs

orgID identifies organization that hosts the registry, the same that is used in function registryAdd().

status defines registry status to set. The currently defined values are:

1. indicates that the registry is active
2. indicates that the registry is "off-line" (temporarily)
3. indicates that the registry is decommissioned

```
function registrySetStatus(byte32 orgID, uint8 status) public
```

6.1.4 Initiating Registry Lookup

This function retrieves a list of registry ids that match the input parameters.

The Worker must match all input parameters (AND mode) to be included in the list.

If the list is too big to fit into a single response (the maximum number of entries in a single response is implementation-specific), the smart contract should return the first batch of the results and provide a lookupTag that can be used by the caller to retrieve the next batch by calling registryLookUpNext.

All input parameters are optional and can be provided in any combination to select a

registry. Inputs appTypeId is an application type that must be supported by the workers.

Outputs

totalCount is the total number of entries matching a specified lookup criteria. If this number is larger than the size of the ids array, the caller should use the lookupTag to call workerLookUpNext to retrieve the rest of the ids.

lookupTag is an optional parameter. If it is returned, it means that there are more matching registry ids that can be retrieved by calling the function registryLookUpNext with this tag as an input parameter.

ids is an array of the registry organization ids that match the input parameters

```
function registryLookUp(bytes32 appTypeId) public view returns
   (int totalCount,     string LookupTag,
bytes32[] ids)
```

6.1.5 Getting Additional Registry Lookup Results

This function is called to retrieve additional results of the Registry lookup initiated by the

registryLookUp call. Inputs appTypeId is an application type that has to be supported by the

workers retrieved.

lookupTag is returned by a previous call to either this function or to registryLookUp. Outputs

totalCount is a total number of entries matching the lookup criteria. If this number is larger than the number of ids returned so far, the caller should use lookupTag to call registryLookUpNext to retrieve the rest of the ids.

newLookupTag is an optional parameter. If it is returned, it means that there are more matching registry ids that can be retrieved by calling this function again with this tag as an input parameter.

ids is an array of the registry ids that match the input parameters

```
function registryLookUpNext(bytes32 applTypeId, string lookUpTag) public returns( int totalCount, string newLookupTag, bytes32[] ids)
```

6.1.6 Retrieving Registry Information

This function retrieves information from the registry.

Inputs id is the id of the registry whose details are

requested.

Outputs

The same as the input parameters to the corresponding call to registryAdd() plus status as defined in registrySetStatus.

```
function registryRetrieve(byte32 workerId) public view returns ( string uri, byte32 scAddr, bytes32[] appTypeIds, uint8 status)
```

6.2 Worker Registry Smart Contract API

APIs in this section are implemented as an Ethereum smart contract referred to as Worker Registry.

6.2.1 Registering a New Worker

This function registers a Worker and is invoked from the Ethereum address of the Trusted Resource Service using [JSON-RPC] with the digital signature associated with the sending Ethereum address.

An Implementation-specific model enforces authorization for this call. Inputs workerID is a worker id, e.g. an Ethereum address or a value derived from the worker's DID.

workerType defines the type of Worker. Currently defined types are:

1. indicates "TEE-SGX": an Intel SGX Trusted Execution Environment
2. indicates "MPC": Multi-Party Compute
3. indicates "ZK": Zero-Knowledge

APIs specific to each type of Worker are given in Appendix A.

organizationID is an optional parameter representing the organization that hosts the Worker, e.g. a bank in the consortium or anonymous entity.

applicationTypeId is an optional parameter that defines application types supported by the Worker.

details is detailed information about the worker in JSON format as defined in Appendix A. This parameter includes either the worker data, or the registry URI where the data can be retrieved via JSON RPC API, as defined in section Off-Chain Worker Registry JSON RPC API.

```
function workerRegister(byte32 workerID, uint8
workerType, bytes32 organizationID, bytes32[]
applicationTypeId, string details) public
```

6.2.2 Updating a Worker

This function updates a Worker and is invoked from the Ethereum address of the Trusted Resource Service using [JSON-RPC] with the digital signature associated with the sending Ethereum address.

Inputs: refer to section "Registing a Worker"

```
function workerUpdate(byte32 workerID, string details) public
```

6.2.3 Setting Worker Status

This function sets a Worker's status. An implementation-specific model enforces authorization policies for this call. Inputs workerID is a worker id status defines Worker status. The currently defined values are:

1. indicates that the worker is active
2. indicates that the worker is "off-line" (temporarily)
3. indicates that the worker is decommissioned
4. indicates that the worker is compromised

```
function workerSetStatus(byte32 workerID, uint8 status) public
```

6.2.4 Initiating Worker lookup

This function retrieves a list of Worker ids that match the input parameters.

The Worker must match all input parameters (AND mode) to be included in the list.

If the list is too large to fit into a single response (the maximum number of entries in a single response is implementation specific), the smart contract should return the first batch of the results and provide a lookupTag that can be used by the caller to retrieve the next batch by calling workerLookUpNext.

All input parameters are optional and can be provided in any combination to select Workers. Inputs workerType is a characteristic of Workers for which you may wish to search organizationId is an id of an organization that can be used to search for one or more Workers that belong to this organization applicationTypeId is an application type that is supported by the Worker Outputs

totalCount is a total number of entries matching a specified lookup criteria. If this number is bigger than size of ids array, the caller should use lookupTag to call workerLookUpNext to retrieve the rest of the ids.

lookupTag is an optional parameter. If it is returned, it means that there are more matching Worker ids that can be retrieved by calling function workerLookUpNext with this tag as an input parameter. ids is an array of the Worker ids that match the input parameters.

```
function workerLookUp( uint8
workerType, bytes32 organizationId,
  bytes32 applicationTypeId) public view returns( int
totalCount, string LookupTag, bytes32[] ids)
```

6.2.5 Getting Additional Worker Lookup Results

This function is called to retrieve additional results of the Worker lookup initiated by workerLookUp call. Inputs workerType is a characteristic of Workers for which you may wish to search. organizationId is an organization to which a Worker belongs. applicationTypeId is an application type that has to be supported by the Worker. lookupTag is returned by a previous call to either this function or to workerLookUp.

Outputs

totalCount is a total number of entries matching this lookup criteria. If this number is larger than the number of ids returned so far, the caller should use lookupTag to call workerLookUpNext to retrieve the rest of the ids.

newLookupTag is an optional parameter. If it is returned, it means that there are more matching Worker ids than can be retrieved by calling this function again with this tag as an input parameter. ids is an array of the Worker ids that match the input parameters.

```
function workerLookUpNext( uint8
workerType, bytes32 organizationId,
bytes32 applicationTypeId, string lookUpTag)
public view returns( int totalCount,
string newLookupTag, bytes32[] ids)
```

6.2.6 Retrieving Worker Information

This function retrieves information for the Worker. It can be called from any authorized Ethereum address. Inputs workerId is the id of a Worker to be retrieved.

Outputs

The same as the input parameters to the corresponding call to workerRegister + status as defined in workerSetStatus.

```
function workerRetrieve(byte32 workerId) public view returns (
uint8 workerType, string workerTypeDataUri, bytes32
organizationId, bytes32[] applicationTypeId, string details,
uint8 status)
```

6.3 Off-Chain Worker Registry JSON RPC API

These are the JSON RPC version of the "Worker Registry Smart Contract API". All messages follow a request-response pattern and are completed synchronously during the same session.

Errors and status are returned using a generic JSON RPC error.

6.3.1 Worker Register JSON Payload

This message registers a Worker. It does not have a specific response payload; instead, a generic error response payload is sent back as a response.

```
{
  "jsonrpc": "2.0",
  "method": "WorkerRegister",
  "id": <integer>,
  "params": {
    "workerId":<hex string or DID>,
    "workerType":<uint>,
    "organizationId":<hex string>,
    "applicationTypeId": [<one or more hex strings>],
    "details":{
      <worker type specific data>
    }
  } } method must be
```

WorkerRegister,

params is a collection of the request parameters. Refer to section Registering a New Worker for a description of the parameters.

6.3.2 orker Update JSON Payload

This message updates a Worker. It does not have a specific response payload; instead, a generic error response payload is sent back as a response.

```
{
  "jsonrpc": "2.0",
  "method": "WorkerUpdate",
  "id": <integer>,
  "params": {
    "workerId":<hex string or DID>,      "details":{
      <worker type specific data>
    }
  } } method must be
```

WorkerUpdate,

params is a collection of the request parameters. Refer to section Updating a New Worker for a description of the parameters.

6.3.3 Worker Set Status JSON Payload

This message sets a Worker's status.

It does not have a specific response payload; instead, a generic error response payload is sent back as a response.

```
{
  "jsonrpc": "2.0",
  "method": "WorkerSetStatus",
  "id": <integer>,
  "params": {
    "workerId": <hex string or DID>,
    "status": <number>
  } } method must be
```

WorkerSetStatus.

workerId is an id of the worker for which to set status.

status can be one of active, offline, decommissioned, or compromised as defined in API "Setting Worker Status" in section "Worker Registry Smart Contract API".

6.3.4 orker Lookup JSON Payload

This message initiates a Worker lookup in the registry.

Its response is defined in section Worker Lookup JSON Response Payload.

```
{
  "jsonrpc": "2.0",
  "method": "WorkerLookUp",
  "id": <integer>,
  "params": {
    "workerType": <uint>,
    "organizationId": <hex string>,
    "applicationTypeId": [<one or more hex strings>]
  } } method must be
```

WorkerLookUp,

params is a collection of the request parameters. Refer to section Initiating Worker Look Up for a description of the parameters.

6.3.5 Worker Lookup Next JSON Payload

This message continues retrieving results initiated by a previous WorkerLookUp message. Its response is defined in section Worker Lookup Response JSON Payload>.

```
{
  "jsonrpc": "2.0",
  "method": "WorkerLookUpNext",
  "id": <integer>,
  "params": {
    "workerType": <uint>,
    "organizationId": <hex string>,
    "applicationTypeId": [<one or more hex strings>]
    "lookUpTag": <string>
  } } method must be
```

WorkerLookUpNext,

params is a collection of the request parameters. Refer to section Getting Additional Worker Lookup Results for a description of the parameters.

6.3.6 Worker Lookup Response JSON Payload

This payload is sent back to a Requester in response to the request defined in sections Worker Lookup JSON Payload and Worker Lookup Next JSON Payload.

```
{
  "jsonrpc": "2.0",
  "id": <integer>,
  "result": {
    "totalCount":<integer,    "lookupTag":<string,
    ids:[<one or more hex strings>]
  }
}
```

'result' is a collection of the response specific parameters. Refer to the output parameters in section Initiating Worker lookup for a description of elements in this object.

6.3.7 Worker Retrieve JSON Payload

This message retrieves a Worker by its ID. Its response is defined in section Worker Retrieve Response JSON Payload.

```
{
  "jsonrpc": "2.0",
  "method": "WorkerRetrieve"
  "id": <integer>,
  "params": {
    "workerId": <hex string or DID>
  } } method must be
```

WorkerRetrieve,

params is a collection of the request parameters. Refer to section Retrieving Worker Information for a description of the parameters.

6.3.8 Worker Retrieve Response JSON Payload

This payload is sent back to a Requester in response to the request defined in section Worker Retrieve JSON Payload.

```
{
  "jsonrpc": "2.0",
  "id": <integer>,
  "result": {
    "workerType":<uint>,
    "organizationId":<hex string>,
    "applicationTypeId": [<one or more hex strings>],
    "details":{
      <worker type specific data>
    },
    "status": <number>
  } }
```

result is a collection of the response-specific parameters. Refer to the output parameters in section Retrieving Worker Information for a description of the elements in this object.

workerTypeDataURI defines a URI where detailed Worker type specification information can be retrieved.

7. Work Orders

- Direct Model: JSON RPC Work Order invocation API over network
- Proxy Model: Work order invocation using Ethereum (proxy) smart contract

7.1 Direct Model Invocation

This section defines a mechanism for executing Work Orders over a network JSON RPC API outside of the blockchain.

This API can be used in several modes:

- Synchronous request-response mode. The exchange of Work Order request and completion result happen in the same HTTP session. Synchronous mode is for Work Orders that don't require a long time to execute.
- Result pulling mode. In this mode the ĐApp disconnects after submitting a Work Order request, then periodically polls the JSON endpoint for the Work Order result
- Asynchronous mode. In this mode, a ĐApp provides a URI for receiving the Work Order result as a part of the Work Order request. The ĐApp disconnects after submitting the Work Order.

On completion of the Work Order, the Worker submits the Work Order result to the URI provided.

- Notification mode. In this mode, A ĐApp provides a URI to receive a notification when the Work Order is completed. The ĐApp disconnects after submitting the Work Order. When the Work Order is completed, the Worker sends an event to the URI provided in the Work Order request. On receiving the event, the client retrieves the Work Order result from the JSON endpoint.

7.1.1 Work Order Request Payload

First, Requester sends a Work Order Request payload in the JSON-RPC based format defined below.

```
{
  "jsonrpc": "2.0",
  "method": "WorkOrderSubmit",
  "id": <integer>,
  "params": {
    "responseTimeoutMSecs": <integer>,
    "payloadFormat": <string>
    "resultUri": <string>,
    "notifyUri": <string>,
    "workOrderId": <hex string>,
    "workerId": <hex string or DID>,
    "workloadId": <hex string>,
    "requesterId": <hex string>,
    "workerEncryptionKey": <hex string>,      "dataEncryptionAlgorithm": <string>,
"encryptedSessionKey": <hex string>,
    "sessionKeyIv": <hex string>,
    "requesterNonce": <hex string>,
    "encryptedRequestHash": <hex string>,
    "requesterSignature": <BASE64 string>,
    "inData": [
      <object>
    ],
    "outData": [        <object>
    ]
  }
}
```

Parameters method is set to

WorkOrderSubmit. params is a

collection of parameters as per JSON

RPC specification. The parameters

are defined below.

responseTimeoutMsecs is a maximum timeout in milliseconds that the caller will wait for the response. Setting this timeout to zero means that the work order is submitted in the asynchronous (if resultUri is present), notify (if notifyUri is present), or pull mode (if neither resultUri nor notifyUri is present). In this case, the TCS should schedule the request for execution and immediately return an error response with error code set to scheduled. If the timeout is not zero, the work order is in synchronous mode. The TCS should wait for the work order completion before returning the response to the participant. If the request cannot be completed within the allocated interval, the work order should be cancelled and a corresponding error should be returned to the participant.

payloadFormat defines how signatures and data items are formatted in this work order request and corresponding response. Its values are defined in section Appendix A.

resultUri is an optional parameter. If it is specified, the WorkerService should submit the Work Order result to this URI. See section Work Order Asynchronous Result.

notifyUri is an optional parameter. If it is specified, the WorkerService should send an event to this URI upon the Work Order completion. See section Work Order Completion Event.

workOrderId is an id assigned to the Work Order by the Requester and can be registered using the Work Order Receipts API. workerId is a worker id to process the work order, e.g. an Ethereum address or its DID.

workloadId is the id of the workload to be executed by the worker. It is optional if the worker includes a single workload. requesterId is either the Requester's Ethereum address or its DID.

workerEncryptionKey is an optional parameter containing the worker encryption key used for this Work Order. It is useful if a Worker frequently updates its encryption key in the registry and allows some time overlap in utilizing multiple keys. We assume here that the 'details' submitted during the registration of a worker contain one or more public keys associated with the worker.

dataEncryptionAlgorithm is an optional parameter that defines an algorithm for encrypting the data in this work order. The default is the first value in the corresponding parameter for the worker (defined by workerId). See section Common Data for All Worker Types.

encryptedSessionKey is a one-time encryption key generated by the participant submitting the work order. It is sent encrypted with the worker's public encryption key. It is used to encrypt encryptedRequestHash and data item specific data encryption keys. For the latter see Work Order Data Formats.

sessionKeyIv is an initialization vector if required by the data encryption algorithm (encryptedSessionKey). The default is all zeros.

requesterNonce is a random string generated by the participant. It is used to calculate a hash of this work order request.

encryptedRequestHash is a hash of the work order request encrypted with the key provided in encryptedSessionKey. See section Work Order Signing for the details. requesterSignature is an optional parameter. See section Work Order Signing for the details.

inData contains either a JWT of the specified data or an array of one or more Work Order inputs, e.g. state, message containing input parameters. See Work Order Data Formats.

outData contains information about what and how the work order execution results should be delivered. See Work Order Data Formats.

After a Work Order request is received, the Worker Service can respond in one of three ways:

- Complete a (short running) Work Order and return the result
- Return an error if the Work Order was rejected or its execution failed
- Schedule a Work Order to be executed later and return a corresponding status

7.1.2 Work Order Result Payload

If a submitted Work Order is completed, its Work Order Result is returned in the following format.

```
{
  "jsonrpc": "2.0",
  "id": <integer>,
  "result": {
    "workOrderId": <hex string>,
```

```
    "workloadId": <hex string>,
    "workerId": <hex string>,
    "requesterId": <hex string>,
    "workerNonce": <string>,      "workerSignature": <BASE64 string>,
    "outData": [        <object>
    ]
  }
}
```

Parameters

result is a collection of parameters as per JSON RPC specification. The parameters are defined below. workOrderId is a Work Order id sent in the corresponding Work Order request.

workloadId is optional; if present, it must match a corresponding value from the work order request.

workerId is optional; if present, it must match a corresponding value from the work order request.

requesterId is optional; if present, it must match a corresponding value from the work order request.

workerNonce is a random string generated by the worker. It is used to calculate a hash of this work order response.

workerSignature is a signature of the work order response. See section Work Order Signing for the details. outData contains the work order execution results. See Work Order Data Formats.

7.1.3 Work Order Status Payload

If the work request fails, is rejected, is scheduled for later execution, or its execution requires a long time; the Work Order Error Response payload is sent in the following format defined in section RPC Encoding Conventions.

```
{
  "jsonrpc": "2.0",
  "id": <integer>,
  "error": {
    "code": "integer",
    "message": <string>,
    "data": {
      "workOrderId": <hex string>
    }
  }
```

```
}
```

Parameters code is an integer number defining an error or Work Order state.

Supported values:

- Values from -32768 to -32000 are reserved for pre-defined errors in JSON RPC spec
- 5 means that the Work Order status is "pending" – scheduled to be executed, but not started yet
- 6 means that the Work Order status is "processing" – its execution has started, but it has not been completed yet
- Values from 7 to 999 are reserved
- All other values can be used by the Worker Service (a.k.a. implementation

specific) data contains additional details about the error that includes: workOrderId

which is a Work Order id as a hexadecimal string.

7.1.4 Work Order Pull Request Payload

If a Requester receives a response stating that its Work Order state is "scheduled" or "processing", it should pull the Worker Service later to get the result. The Requester has two pulling options:

- Pull the Worker Service periodically until the Work Order is completed successfully or in error
- Wait for the Work Order Receipt complete event and retrieve a final result. Refer to Work Order Receipts for more details.

In either case, the Work Order Pull Request must follow the format below:

```
{
  "jsonrpc": "2.0",
  "method": "WorkOrderGetResult",
  "id": <integer>,
  "params": {
    "workOrderId": <hex string>
  }
}
```

Parameters method is set to

WorkOrderGetResult.

params is a collection of parameters as per JSON RPC specification. The parameters are defined below. workOrderId is a Work Order id that was sent in the corresponding WorkOrderSubmit request.

7.1.5 Work Order Asynchronous Result

If the client provides resultUri in the Work Order request payload, the Worker will send the Work Order execution result to the URI provided in the same format as defined in "Work Order Result Payload".

The client responds with a payload as defined in section Work Order Status Payload.

7.1.6 Work Order Completion Event

If the client provides notifyUri in the Work Order request payload, the Worker sends an event to the Requester on completion of the Work Order, regardless of whether the Work Order was completed successfully or not.

The event payload format:

```
{
  "jsonrpc": "2.0",
  "id": <integer>,
  "result": {
    "workOrderId": <hex string>
  }
}
```

Parameters result is a collection of JSON RPC parameters, as

follows:

workOrderId is the ID of the Work Order sent in the Work Order request.

On receiving this event, the client will pull the Work Order result as defined in section "Work Order Pull Request Payload" above.

7.1.7 Work Order Data Formats

This section defines formats for inData and outData elements within Work Order requests and responses. The format depends on the value of payloadFormat in the work order request. inData and outData are sent as arrays of objects with the following JSON elements:

```
{
  "index": <number>,
  "dataHash": <hex string>,
  "data": <BASE64 string>,
  "encryptedDataEncryptionKey": <hex string>,
  "iv": <hex string>
}
```

If payloadFormat is set to a custom value, it is an application-specific format.

Below are descriptions of the JSON elements:

index is an index that determines the order of the data items for hash generation. It can also be used by the worker to identify different inputs and outputs.

dataHash is an optional hash value of the data. It is only applicable to inData in the work order request and outData in the response.

data contains either data inline within the JSON document or a reference (e.g. URI) to the data. It is up to the worker to determine how to interpret the data content. This parameter is applicable to

- inData in the work order request
- outData in the request if it contains a reference for the output outData in
- the response

encryptedDataEncryptionKey defines whether data is encrypted and what key to use. It is only included in the work order request as one of the options below

- If this key is not provided or set to "null" or to "", the data is encrypted using encryptedSessionKey from the Work Order request
- If the key value is set to "-", the data item is not encrypted, a.k.a. sent as clear text
- Otherwise, the data item is sent encrypted with a one-time encryption key generated by a 3rd party that owns this data item (it may be different from the Work Order requester). encryptedDataEncryptionKey contains this encryption key in double encrypted format

- First, it is encrypted with the worker's public encryption key (e.g. by a 3rd party that owns the data so the requester cannot see the data)
- Then the result of the previous encryption above is encrypted with the key from encryptedSessionKey by the requester, to enforce the work order integrity.

iv is an initialization vector, if required by the data encryption algorithm. The default is all zeros. If the same encryption key is used to encrypt more than one data item or the hash value of the work order request, iv must be a unique random number for every encryption operation. It is included only in the Work Order request.

7.1.8 Work Order Signing

Work Order request and response signing depends on the value of payloadFormat in the work order request.

- for JSON-RPC signing mechanism is defined in section Signing for JSON-RPC format.
- for JSON-RPC-JWT signing mechanism is defined in section Signing for JSON-RPC-JWT format.
- for custom values the signing mechanism is application-specific, and is not defined in this specification.

Note that there are two mechanisms ensuring the integrity of a work order request:

- encryptedRequestHash contains the hash of the work order request encrypted with a key from encryptedSessionKey. This mechanism ensures the request integrity even in the case of an anonymous requester. The hash value in this case is verified by the worker.
- requesterSignature is an optional signature that uses the same calculated hash value and signs it with the requester's private signing key. Distribution of the corresponding public keys is outside of the scope of this specification. The signature is verified either by the service or by the worker. The signature use is application-specific, and it is optional for the request integrity verification. It can be used for the requester authentication and the Work Order authorization outside or inside of the worker itself.

A Work Order request's signature is always generated by the worker that processed the Work Order.

7.1.8.1 Request Hash Calculation and Encryption

This section defines steps for a Work Order request's hash calculation and encryption. All data transmitted in BASE64 or HEX format have to be decoded for the hash calculation.

The hashing algorithm is defined in the Worker's parameter hashingAlgorithm, as per section Common Data for All Worker Types.

First, the requester calculates the Work Order request hash:

- Generate a random number and store its hash in requesterNonce.
- Calculate a hash value of the string concatenating the following values: requesterNonce, workOrderId, workerId, workloadId, and requesterId.
- For each item in the inData array calculate a hash value of the array concatenating dataHash, data, encryptedDataEncryptionKey, iv. The array items are ordered according to the index field. If data is encrypted, the hash is calculated over the encrypted data. If `encryptedDataEncryptionKey contains an encrypted key, the hash is calculated over the encrypted data.
- For each item in the outData array, calculate a hash value of the array concatenating dataHash, data, encryptedDataEncryptionKey, iv. The array items are ordered according to the index field. If data is encrypted, the hash is calculated over the encrypted data. If `encryptedDataEncryptionKey contains an encrypted key, the hash is calculated over the encrypted data.
- Combine all hashes calculated above into a single array in the order they were generated. Calculate another hash of the combined array.

Then the requester must encrypt the calculated hash and include it in the Work Order request:

- Encrypt the combined hash with a key from encryptedSessionKey
- Format the encrypted hash as a HEX value
- Place it in the encryptedRequestHash parameter of the Work Order request

Optionally, the calculated hash can be signed by the requester in one of the two formats as defined in sections Signing for JSON-RPC Format and Signing for JSON-RPC-JWT Format.

7.1.8.2 Response Hash Calculation

This section defines the Work Order response hash calculation. All data transmitted in BASE64 or HEX format have to be decoded for the hash calculation.

The hashing algorithms is defined in Worker's parameter hashingAlgorithm, See section Common Data for All Worker Types.

The worker performs the following steps:

- Generate a random number and store its hash in workerNonce.
- Calculate a hash value of the string concatenating following values workerNonce, workOrderId, workerId, workloadId, and requesterId.
- For each item in the outData array, calculate a hash value of the array concatenating dataHash and data. The array items are ordered according to the index field. If data is encrypted, the hash is calculated over the encrypted data.
- Combine all hashes calculated above into a single array in the order they were generated. Calculate another hash of the combined array.

7.1.8.3 Signing for JSON-RPC Format

This section defines a Work Order signing mechanism for the JSON-RPC payload format.

For a Work Order request, the signature is optional. If a signature is provided, it is generated according to the following steps:

- The signing algorithm is defined in Worker's parameter signingAlgorithm, as per section Common Data for All Worker Types.
- The Work Order request hash defined in section Request Hash Calculation and Encryption is signed with requester's private signing key.
- The signature is formatted as a BASE64 string.
- The resulting string is placed in the requesterSignature of the Work Order request payload.

For a Work Order Response, the following steps are performed by the worker:

- The signing algorithm is defined in Worker's parameter signingAlgorithm, as per section Common Data for All Worker Types.
- The Work Order response hash defined in section Response Hash Calculation is signed with the worker's private signing key.
- The signature is formatted as a BASE64 string.
- The resulting string is placed in the workerSignature of the Work Order response payload.

7.1.8.4 Signing for JSON-RPC-JWT Format

This section defines the Work Order signing mechanism for the JSON-RPC-JWT payload format.

The Work Order request requesterSignature and response workerSignatures are JWT formatted strings.

The JWT header must include hashing and signing algorithms that match hashingAlgorithm and signingAlgorithm parameters of the Worker. See section Common Data for All Worker Types.

The JWT payload is in the following format:

```
{
"hash": <HEX string>
}
```

The Work Order request hash parameter contains a value as defined in Request Hash Calculation and Encryption.

The Work Order response hash parameter contains a value as defined in Response Hash Calculation.

7.1.9 Get Encryption Key Request Payload

This section defines a JSON RPC request that is called by a requester to receive a Worker's key. Normally, it is used if the Worker supports requester-specific encryption keys in addition to or instead of the encryptionKey defined in section Appendix A: Worker Specific Detailed Data.

If this request fails, then a Work Order Status Payload is returned. The following values are defined for the code parameter:

- 1 - generic error
- 2 - operation is not supported
- 3 - invalid parameter
- 4 - access denied
- 5 - not ready, retry later. This is a recoverable error that may happen if the requester makes its first request for keys, or the requester retrieves keys, faster than the worker produces them. The requester should retry later.

If this request succeeds, the response is defined in section Get Encryption Key Response Payload.

The payload format:

```
{
  "jsonrpc": "2.0",
  "method": "EncryptionKeyGet",
  "id": <integer>,
  "request": {
    "workerId": <hex string>,
    "lastUsedKeyNonce": <hex string>,
    "tag": <hex string>,
    "requesterId": <hex string>,
    "signatureNonce": <hex string>,
    "signature": <BASE64 string>
  }
}
```

method must be EncryptionKeyGet. workerId is the id of the worker whose encryption key is requested. lastUsedKeyNonce is an optional nonce associated with last retrieved key. If it is provided, the key retrieved should be newer than this one. Otherwise any key can be retrieved.

tag is tag that should be associated with the returned key, e.g. requester id. This is an optional parameter. If it is not provided, requesterId is used as a key.

requesterId is the id of the requester that plans to use the returned key to submit one or more work orders using this key. signatureNonce is an optional parameter and is used only if signature below is also provided.

signature is an optional signature of workerId, lastUsedKeyNonce, tag, and signatureNonce. The hashing and signing algorithms are defined in hashingAlgorithm and encryptionAlgorithm for the Worker in section Common Data for All Worker Types.

7.1.10 Get Encryption Key Response Payload

This section defines a payload returned by the Worker Service in response to successful Get Encryption Key Request Payload.

```
{
```

```
"jsonrpc": "2.0",
"id": <integer>,
"request": {
  "workerId": <hex string>,
  "encryptionKey": <hex string>,      "encryptionKeyNonce": <hex string>,
  "tag": <hex string>,
  "signature": <BASE64 string>
} }
```

workerId is the id of the worker that created the encryption key.

encryptionKey is an encryption key. encryptionKeyNonce is a

nonce associated with the key. tag is tag associated with the key.

signature is a signature generated by the worker. The hashing and signing algorithms are defined in hashingAlgorithm and encryptionAlgorithm for the worker in section Common Data for All Worker Types. The signature is calculated as follows:

- a hash is calculated over the concatenation of workerId, encryptionKey, encryptionKeyNonce, and tag.
- the hash is signed by the worker's signing key corresponding to verificationKey defined in Appendix A. • the hash is formatted as BASE64 string.

7.1.11 Set Encryption Key Request Payload

This section defines a JSON RPC request that is called by a Worker or Worker Service to receive a Worker's key.

The response to this request is a Work Order Status Payload. The error values are defined in section

Set Encryption Key Request Payload The payload format:

```
{
  "jsonrpc": "2.0",
  "method": "EncryptionKeySet",
  "id": <integer>,
  "request": {
    "workerId": <hex string>,
    "encryptionKey": <hex string>,      "encryptionKeyNonce": <hex string>,
    "tag": <hex string>,
```

"signatureNonce": <hex string>,
"signature": <BASE64 string>
}} method must be EncryptionKeySet.

Other parameters are defined in section Get Encryption Key Response Payload.

7.2 Proxy Model Invocation

This section defines a Work Order execution mechanism using the Invocation proxy Ethereum smart contract created by the WorkerService for its associated Worker(s). Its address is available in the Worker Service Registry.

7.2.1 Submitting a New Work Order

This function creates a new Work Order. It is called by a ĐApp or an enterprise application smart contract from the Requester's address. The Proxy Smart Contract is called by a ĐApp or an enterprise application from the Requester's Ethereum address.

As a side effect this function may create a Work Order Receipt for the Work Order.

This function emits event workOrderNew.

This function uses an implicit parameter transaction sender that is included in the event workOrderNew and returned by workOrderGetRequest.

Inputs:

workOrderId should match the corresponding field in the workOrderRequest. workerId should match the corresponding field in the workOrderRequest. requesterId is a requester id that must match the corresponding field in the workOrderRequest. workOrderRequest is a Work Order request data in one of the following formats

- JSON payload as defined in the details of the workOrderRequest in section Direct Model Invocation.
- HTTP(S) URI that points to the same JSON payload as above. A URI is normally used to minimize the storage required to store the work order on the blockchain.

Outputs:

errorCode is an error code, 0 - success, otherwise an error.

```
function workOrderSubmit(    bytes32
workOrderId,     bytes32 workerId,
bytes32 requesterId,
   string workOrderRequest) public returns uint32 errorCode
```

7.2.2 New Work Order Event

This event is emitted by workOrderSubmit function; workOrderId is the submitted Work Order id.

This event is intended for the Worker(s) that is supposed to execute the Work Order.

Parameters

workOrderId, workerId, requesterId, workOrderRequest, and errorCode are defined in section Submitting a New Work Order.

senderAddress is an Ethereum address from which a corresponding workOrderSubmit() call was done. version is a version of the API, e.g. value 0x01020381 corresponds to version "1.2.3.129".

```
event workOrderNew (bytes32 indexed workOrderId,      bytes32
indexed workerId,      bytes32 indexed requesterId,      string
workOrderRequest,      uint32 errorCode,      address
senderAddress,      byte4 version)
```

7.2.3 Completing a Work Order

This function is called by the Worker Service to complete a Work Order successfully or in error.

This function can be executed either from the Worker or Worker Service address.

This function may emit event workOrderDone.

Inputs:

workOrderId is an id of the Work Order provided during call to workOrderSubmit. workOrderStatus is the work order completion status. Zero is success, otherwise is an error. workOrderResponse is the Work Order response data in one of the following formats

- JSON payload as defined in the details of the workOrderRequest in section Direct Model Invocation.
- HTTP(S) URI that points to the same JSON payload as above. A URI is normally used to minimize storage required to store the work order on the blockchain.

Outputs:

errorCode is an error code, 0 - success, otherwise an error.

```
function workOrderComplete(bytes32 workOrderId,      uint32
workOrderStatus,
    string workOrderResponse) public returns (uint32 errorCode)
```

7.2.4 Work Order Done Event

This event is emitted by workOrderComplete. This event is intended for the Requester who submitted the Work Order.

Inputs:

requesterId is an Id of the Requester that submitted the Work Order

workOrderId, workOrderStatus, workOrderResponse, and errorCode are defined in section Completing a Work Order.

version is the version of the API, e.g. value 0x01020381 corresponds to version "1.2.3.129"

```
event workOrderDone(bytes32 requesterId,      bytes32
workOrderId,      uint32 workOrderStatus,      string
workOrderResponse,      uint32 errorCode,      bytes4
version)
```

7.2.5 Retrieving Work Order Response Information

This is an optional function that returns a Work Order response. It is recommended to only allow this function to be called from the address of the Requester who submitted this Work Order.

Inputs:

workOrderId is an id of the Work Order for which to retrieve a response

Outputs:

workOrderId, workOrderStatus, workOrderResponse, and errorCode are defined in section Completing a Work Order.

```
function workOrderGetResult(bytes32 workOrderId) public view returns(uint32
errorCode, bytes32 workOrderId, uint32 workOrderStatus, string
workOrderResponse)
)
```

7.2.6 Setting Encryption Key

This function is used by the Worker Service to set an encryption key, e.g. for a specific requester.

Inputs:

workerId, keyNonce, tag, and signature are defined in section Get Encryption Key Response Payload. Outputs:

errorCode is the result of the operation. Zero is success, other values indicate an error occurred.

This function may emit the event encryptionKeySet defined in section Encryption Key Set Event

```
function encryptionKeySet(bytes32 workerId, bytes32
tag, bytes32 nonce,
    bytes signature) public returns (uint32 errorCode) )
```

7.2.7 Encryption Key Set Event

This event is emitted by encryptionKeySet.

Inputs:

Its parameters defined in section Setting Encryption Key.

```
event encryptionKeySet(bytes32 indexed workerId, bytes32
indexed tag, bytes32 nonce, bytes signature uint32
indexed errorCode)
```

7.2.8 Retrieving Encryption Key

This function is used by the requester to get an encryption key.

Inputs:

Input parameters are defined in section Set Encryption Key Request Payload.

Outputs:

errorCode is the result of the operation. Its values are defined in section Get Encryption Key Request Payload. If errorCode is "5 - not ready, retry later", the requester should call function encryptionKeyStart() defined in section Starting Encryption Key Generation.

Other output parameters are defined in section Setting Encryption Key.

```
<pre class="solidity">function encryptionKeyRetrieve(bytes32 workerId,  bytes32
lastUsedKeyNonce,  bytes32 tag,
  bytes32 requesterId) public view
returns(uint32 errorCode,  bytes32 workerId,
bytes32 tag,  bytes32 nonce,  bytes
signature

)
```

7.2.9 Starting Encryption Key Generation

This function is used by the requester to inform the Worker that it should start encryption key generation for this requester (and/or tag).

Inputs:

tag is an optional parameter. If it is zero, the transaction sender's address is used as a tag.

Outputs:

errorCode is a result of operation. Its values are defined in section Get Encryption Key Request Payload.

This function may emit the event encryptionKeyStart defined in section Encryption Key Start Event

```
function encryptionKeyStart(bytes32 tag) returns(uint32 errorCode), )
```

7.2.10 Encryption Key Start Event This event

is emitted by encryptionKeyStart.

Inputs:

requesterAddress is a sender address who submitted a corresponding call to encryptionKeyStart.

tag and serrorCode are defined in section Starting Encryption Key Generation.

```
event encryptionKeyStart(bytes32 indexed requesterAddress,      bytes32 indexed
tag,      uint32 indexed errorCode)
```

8. Work Order Receipts

This chapter defines two modeLs for Work Order Receipts:

- Smart contract mode when receipts are managed by an Ethereum Work Order Receipts smart contract.
- Direct model when receipts are managed off-chain and accessed via JSON RPC API.

8.1 Proxy Model Receipt Handling

This API provides the following capabilities:

- Creating Work Order receipt by the Requester.
- Updating Work Order by the Requester, by the Worker assigned to execute the work order, or any other authorized participant.
- Retrieving a Work Order Receipt information.
- Retrieving Work Order Receipt updates.
- Enumerating Work Order Receipts.
- Generating Ethereum log events on the receipt create and update.

8.1.1 Creating a Work Order Receipt

This function is implemented by Work Order Receipts smart contract and is called by a Requester, or on behalf of a Requester, to create a Work Order Receipt in the submitted state.

The sender address is an implicit parameter that is saved as a part of the receipt.

Inputs:

workOrderId is the id of the Work Order.

workerId is the Worker id that should execute the Work Order. workerServiceId is an

id of the Worker Service that hosts the Worker. requesterId is the id of the requester.

receiptCreateStatus is an initial receipt status, it can be

- 0 - "pending". The work order is waiting to be processed by the worker
- 1 - "completed". The worker processed the Work Order and no more worker updates are expected
- 2 - "processed". The worker processed the Work Order, but additional worker updates are expected, e.g. oracle notifications
- 3 - "failed". The Work Order processing failed, e.g. by the worker service because of an invalid workerId
- 4 - "rejected". The Work Order is rejected by the smart contract, e.g. invalid
- workerServiceId values from 5 to 254 are reserved value 255 indicates any status
- values above 255 are application specific values
- workOrderRequestHash is a hash value of the work order request as defined in section Submitting a New Work Order.

Outputs:

errorCode is a result of the function. A value of '0' indicates success, other values indicate an error occurred.

```
function workOrderReceiptCreate(bytes32 workOrderId, bytes32
workerId, bytes32 workerServiceId, bytes32 requesterId, uint32
receiptCreateStatus,
    bytes workOrderRequestHash) public returns (uint32 errorCode)
```

8.1.2 Updating a Work Order Receipt

This API is implemented by a Work Order Receipts smart contract and it can be called by one of the following participants:

- By or on the behalf of the Worker identified during the receipt creation, e.g. to notify about the work order completion
- By or on the behalf of other Workers, e.g. to submit an oracle notification
- By the Work Order Receipt creator (requester)
- By other participants, e.g. to acknowledge the Work Order results in case of multi-party Work Order processing

Inputs:

`workOrderId` is the id of the Work Order.

`updaterId` is the id of the updating entity. It is optional, if it is the same as the transaction sender address.

`updateType` is a type of the Work Order update that defines how the update should be handled

- If `updateType` is from 0 to 255, the update sets the receipt status to `updateType` value, as descrined in section Creating a Work Order Receipt
- If it is any other value, the processing is application-specific `updateData` are

update-specific data that depend on the 'workOrderStatus` as follows:

- If the update sets the Work Order Receipt status to `completed` or `processed`, it is a hash value of the Work Order Response
- in all other cases, `updateData` has an application-specific value not define by this specification.

`updateSignature` is an optional signature of `workOrderId`, `updateType`, and `updateData`. It is required only if the `updaterId` is not the same as the transaction sender address. The hashing and signing algorithms are defined by `signatureRules`.

`signatureRules` defines hashing and signing algorithms, that are separated by forward slash '/', e.g. "SHA-256/RSA-OAEP-4096". It is an optional parameter and it is required if signing algorithms are different from the algorithms defined for the Worker defined during the receipt creation.
Outputs:

errorCode is the result of the function. A value of '0' indicates success, other values indicate that an error occurred.

```
function workOrderReceiptUpdate(bytes32 workOrderId, bytes32
updaterId, uint32 updateType, bytes updateData, bytes
updateSignature,
    string signatureRules) public returns (uint32 errorCode)
```

8.1.3 Retrieving a Work Order Receipt

Authorized entities can retrieve a Work Order receipt using this function.

Inputs:

workOrderId is the id of the Work Order to be retrieved.

Outputs:

workerServiceId, requesterId, ``workOrderId, receiptCreateStatus, and workOrderRequestHash are defined in section Creating a Work Order Receipt.

currentReceiptStatus matches

- the receiptCreateStatus at the time of the receipt creation, if there have not been any receipt updates changing its status
- the status set by the latest receipt update

```
function workOrderReceiptRetrieve(bytes32 workOrderId) public view returns(bytes32
workerServiceId, bytes32 workerId, bytes32 requesterId, uint32 receiptCreateStatus,
bytes workOrderRequestHash, uint32 currentReceiptStatus)
```

8.1.4 Retrieving a Work Order Receipt Update

Authorized entities can retrieve a Work Order receipt update using this function.

Inputs:

workOrderId is the id of the Work Order to be updated. updaterId is the id of

the updating entity. If it null, updaterId is ignored. updateIndex is an index of

the update to retrieve. Value "0xFFFFFFFF" is reserved to retrieve the last

received update.

Outputs:

updaterId, updateType, updateData, updateSignature, and signatureRules are defined in section Updating a Work Order Receipt.

updateCount contains the total number of updates for this receipt, if updaterId is null, otherwise the total number of updates made by 'updaterId'.

```
function workOrderReceiptUpdateRetrieve(bytes32 workOrderId, bytes32 updaterId,
    int4 updateIndex) public view returns(bytes32 updaterId, uint32 updateType,
bytes updateData, bytes updateSignature, string signatureRules, uint32
updateCount)
```

8.1.5 Work Order Receipt Lookup

This function retrieves a list of receipt Ids filtered by one or more input parameters. If more than one input parameter is provided, a receipt must match all parameters to be included in the list (Boolean AND operation).

Inputs:

workerServiceId is the id of the Worker Service whose receipts will be retrieved.

workerId is the Worker id whose receipts are requested. requesterId is the id of the

entity requesting receipts.

receiptStatus defines the status of the receipts retrieved. The supported values are defined in section Creating a Work Order Receipt.

Outputs:

totalCount is the total number of receipts matching the lookup criteria. If this number is larger than the size of the ids array, the caller should use a lookupTag to call workOrderReceiptLookUpNext to retrieve the rest of the receipt ids.

lookupTag is an optional parameter. If it is returned, it means that there are more matching receipts than can be retrieved by calling workOrderReceiptLookUpNext with this tag as an input parameter. ids is an array of the Work Order receipt ids that match the input parameters.

```
function workOrderReceiptLookUp(bytes32 workerServiceId, bytes32
workerId, bytes32 requesterId, uint32 receiptStatus) public view returns(
int totalCount, string lookUpTag, //OPTIONAL bytes32[] ids)
```

8.1.6 Work Order Receipt Lookup Next

This function is called to retrieve additional results of the Work Order receipt lookup initiated by the workOrderReceiptLookUp call.

Inputs:

workerServiceId, workerId, and requesterId are input parameters and lastLookupTag is one of the output parameters for the function workOrderReceiptLookUp defined in section Work Order Receipt Lookup.

Outputs:

totalCount is the total number of receipts matching the lookup criteria.

lookupTag is an optional parameter. If it is returned, it means that there are more matching receipts, that can be retrieved by calling this function again with this tag as an input parameter.

ids is an array of the Work Order receipt ids that match the input criteria from the corresponding call to workOrderReceiptLookUp.

```
function workOrderReceiptLookUpNext(byte32 workerServiceId, bytes32
workerId, bytes32 requesterId, uint32 receiptStatus, string lastLookUpTag)
public view returns( int totalCount, string lookUpTag, bytes32[] ids)
```

8.1.7 Work Order Receipt Update Event

The smart contract can use the Ethereum event log for receipt updates.

An event as described below is emitted by the function workOrderReceiptUpdate. Its parameters are defined in section Updating a Work Order Receipt.

```
event workOrderReceiptUpdated(bytes32 indexed workOrderId, bytes32
indexed updaterId, bytes32 indexed updateType, bytes updateData,
bytes updateSignature, string signatureRules, uint32 errorCode)
```

8.1.8 Work Order Receipt Create Event

The smart contract can use the Ethereum event log to capture receipt creation.

An event as decribed below is emitted by function workOrderReceiptCreate. Its parameters are defined in section Creating a Work Order Receipt.

```
<pre class="solidity">event workOrderReceiptCreated(bytes32 indexed workO   bytes32 indexed
workerServiceId,  bytes32 workerId,  bytes32 indexed requesterId,  bytes32 receiptStatus,  bytes
workOrderRequestHash,  uint32 errorCode)</pre>
```

8.2 Direct Model Receipt Handling

Work Order Receipt management can be done off-chain by a dedicated service, or in some cases by a Worker Service itself.

This section defines the Work Order Receipts JSON RPC API to support this model. Requesters and Workers invoke this API without relying on blockchain smart contracts, hence this is called the "direct model".

This API assumes a synchronous request-response model when the result is returned during the same HTTP session. An asynchronous mode is not defined in this revision, but may be added later.

8.2.1 Status and Error Payload Structure

All error responses and status are reported in the format defined in the section RPC Encoding Conventions. This format is also used to report a successful request if the request does not assume any return values, e.g. creating a Work Order receipt.

8.2.2 Receipt Create Request Payload

This request is sent by a Requester to create a new Work Order Receipt. This request does not have a specific corresponding response payload, hence the error and status payload is used as a response.

```
{
  "jsonrpc": "2.0",
  "method": "WorkOrderReceiptCreate",
  "id": <integer>,
  "params": {
    "workOrderId": <hexadecimal string>
```

"workerServiceId": <hexadecimal string>,
"workerId": <hexadecimal string>,
"requesterId":< <hexadecimal string>,
"receiptCreateStatus":< <number>,
"workOrderRequestHash": <base64 string>
"requesterGeneratedNonce": <hexadecimal string>,
"requesterSignature": <base64 string>,
"signatureRules": <string>
} } method must be WorkOrderReceiptCreate.

params is a collection of the request parameters below:

workOrderId, workerServiceId, workerId, requesterId, receiptCreateStatus, and workOrderRequestHash are defined in section Creating a New Work Order Receipt.

requesterGeneratedNonce is a random number or a monotonic counter generated by the Requester and included in the signature below.

requesterSignature is a signature of the hash calculated over the the concatenated workOrderId, workerServiceId, workerId, requesterId, workOrderStatus, workOrderRequestHash and requesterGeneratedNonce parameters. Hashing and signing algorithms are defined by the signatureRules parameter. Verification key provisioning is application-specific, e.g. it can be an explicit mapping between the key and requesterId or requesterId can be an Ethereum address or a verification key itself.

signatureRules defines hashing and signing algorithms, that are separated by a forward slash '/', e.g. "SHA-256/RSA-OAEP-4096". It is an optional parameter and it is required if the signing algorithms are different from the algorithms defined for the Worker defined by workerId.

8.2.3 Receipt Update Request Payload

This request payload is sent to update the Receipt. This request does not have a specific corresponding response payload - a generic error response is returned.

```
{
  "jsonrpc": "2.0",
  "method": " WorkOrderReceiptUpdate",
  "id": <integer>,
  "params": {
    "workOrderId": <hexadecimal string>,
    "updaterId": <hexadecimal string>,
```

```
    "updateType": <number>,
    "updateData": <base64 strings>,
    "updateSignature": <base64 string>,
    "signatureRules": <base64 string>
  } } method must be WorkOrderReceiptUpdate.
```

params is a collection of the parameters defined in section Updating a New Work Order Receipt with the following exceptions:

- updateSignature is always required
- If an updating entity is a Worker, than the Worker's verification key is used to verify the signature
- If an updating entity is not a worker (requester, participant), the verification key provisioning is application-specific, e.g. it can be an explicit mapping between the key and requesterId or requesterId can be an Ethereum address or a verification key itself.

8.2.4 Receipt Retrieve Request Payload

This request is sent to retrieve a Work Order Receipt. The response to this request is defined in section Receipt Retrieve Responce Payload.

```
{
  "jsonrpc": "2.0",
  "method": "WorkOrderReceiptRetrieve",
  "id": <integer>,
  "params": {
    "workOrderId": <hexadecimal string>
  } } method must be WorkOrderReceiptRetrieve.
```

params is a collection of the parameters defined in section Updating a New Work Order Receipt.

8.2.5 Receipt Retrieve Response Payload

This payload is returned to a Requester in response to the request defined in section Receipt Retrieve Request Payload.

```
{
  "jsonrpc": "2.0",
  "id": <integer>,
  "result": {
```

```
    "workOrderId": <hexadecimal string>
    "workerServiceId": <hexadecimal string>,
    "workerId": <hexadecimal string>,
    "requesterId":< <hexadecimal string>,
    "requestCreateStatus":< <number>,
    "workOrderRequestHash": <base64 string>
    "requesterGeneratedNonce": <hexadecimal string>,
    "requesterSignature": <base64 string>,
    "signatureRules": <string>,
    "receiptCurrentStatus": <number>
  } }
```

result is a collection of the parameters defined in section Receipt Create Request Payload.

8.2.6 Receipt Update Retrieve Request Payload

This request is sent to retrieve a Work Order Receipt Update. The response to this request is defined in section Receipt Update Retrieve Response Payload.

```
{
  "jsonrpc": "2.0",
  "method": "WorkOrderReceiptUpdateRetrieve",
  "id": <integer>,
  "params": {
    "workOrderId": <hexadecimal string>,
    "updaterId": <hexadecimal string>,      "updateIndex": < number>
  } } method must be WorkOrderReceiptUpdateRetrieve.
```

params is a collection of the input parameters defined in section Retrieving a Work Order Receipt Update.

8.2.7 Receipt Update Retrieve Response Payload

This payload is sent back to a Requester in response to the request defined in section Work Order Receipt Retrieval Request Payload.

```
<pre class="JSON">{
"jsonrpc": "2.0",
"id": <integer>,
"result": {
  "updaterId": <hexadecimal string>,
```

```
    "updateType": <number>,
    "updateData": <base64 strings>,
    "updateSignature": <base64 string>,
    "signatureRules": <base64 string>,
    "updateCount": < number>
}
```

} result is a collection of the parameters below:

updaterId, updateType, updateData, updateSignature, and signatureRules are defined in section Receipt Update Request Payload.

updateCount contains the total number of updates for this receipt, if the updaterId is null. Otherwise it returns the total number of updates made by the entity whose id is 'updaterId.

8.2.8 Receipt Lookup Request Payload

This payload is sent by a Requester to get a list of Work Order Receipt Ids matching the input parameters. If more than one input parameter is provided, a receipt must match all parameters to be included in the list (Boolean AND Operation).

The response to this request is defined in section Receipt Lookup Response Payload. Note that the response may not provide a complete list of matching ids, and the Requester may need to send one or more WorkOrderReceiptLookUpNext calls in order to retrieve the complete list.

```
{
   "jsonrpc": "2.0",
   "method": "WorkOrderReceiptLookUp",
   "id": <integer>,
   "params": {
      "workerServiceId": <hex string>,
      "workerId": <hex string>,
      "requesterId": <hex string>,
      "receiptStatus": <hex string>
   } } method must be WorkOrderReceiptLookUp.
```

params is a collection of the input parameters defined in section Work Order Receipt Lookup.

8.2.9 Receipt Lookup Response Payload

This payload is sent back to a Requester in response to the request defined in sections Receipt Lookup Request Payload and Receipt Lookup Next Request Payload.

```
{
  "jsonrpc": "2.0",
  "id": <integer>,
  "result": {
    "totalCount": <int>,
    "lookUpTag": < string>, //OPTIONAL
    "ids":[ <hexadecimal string>]
  } }
```

result is a collection of the output parameters defined in section Work Order Receipt Lookup.

8.2.10 Receipt Lookup Next Request Payload

This function is called to retrieve additional results of the Work Order receipt lookup initiated by the request defined in section Receipt Lookup Request Payload. Since the call may not return the complete list of ids, more than one lookup may be necessary to retrieve the complete list. Each call should use the value of lookupTag returned by the previous call.

The response to this request is defined in section Work Order Receipt Lookup Response Payload.

```
{
  "jsonrpc": "2.0",
  "method": "WorkOrderReceiptLookUpNext",
  "id": <integer>,
  "params": {
    "workerServiceId": <hexadecimal string>,
    "workerId": <hexadecimal string>,
    "requesterId": <hexadecimal string>,
    "receiptStatus": < hexadecimal string>,
    "lastLookUpTag": <hexadecimal string>
  } } method must be WorkOrderReceiptLookUpNext,
```

params is a collection of the input parameters defined in section Work Order Receipt Lookup.

9. Appendix A: Worker Specific Detailed Data

This appendix defines data to be included in the details parameter defined in section Registering a New Worker. These include data common to all worker types, and data specific to each worker type.

9.1 Common Data for All Worker Types

```
{
  "workOrderSyncUri": <hex string>,
  "workOrderAsyncUri": <hex string>,
  "workOrderPullUri": <hex string>,
  "workOrderNotifyUri": <hex string>,
  "receiptInvocationUri": <hex string>,
  "workOrderInvocationAddress": <hex string>,
  "receiptInvocationAddress": <hex string>,
  "fromAddress": <hex string>,
  "hashingAlgorithm": <string>,
  "signingAlgorithm": <string>,
  "keyEncryptionAlgorithm": <string>,
  "dataEncryptionAlgorithm": <string>,
  "workOrderPayloadFormats": [<hex string>],
  "workerTypeData" : { ...} }
```

workOrderSyncUri, workOrderAsyncUri, workOrderPullUri, and workOrderNotifyUri are URIs to be used in the direct model to submit work orders, and correspondingly in synchronous, asynchronous, pull, and notification modes. Multiple Workers belonging to the same organization can share the same URI.

receiptInvocationURI is a URI that should be used to manage Work Orders receipts processed by this Worker in the direct model. Multiple Workers such as Workers belonging to the same organization can share the same URI.

workOrderInvocationAddress is an address of the Work Order Invocation Proxy smart contract that should be used to submit Work Orders for the Worker in the proxy model.

receiptInvocationAddress is an address of the Work Order Receipt smart contract that should be used to submit Work Order receipts in the proxy model. Multiple Workers can share the same address.

fromAddress is an Ethereum address that is used by or on behalf of this Worker to submit transactions. It can be the same as the trusted compute service's id.

hashingAlgorithm is an optional string containing a comma-separated list of the supported hashing algorithms. The default is SHA-256. Defined values are:

- SHA-256
- KECCAK-256

signingAlgorithm is an optional string containing a comma-separated list of the supported signing algorithms. The default is SECP256K1. Examples:

- SECP256K1
- RSA-OAEP-3072

keyEncryptionAlgorithm is a string containing an asymmetric encryption algorithm used by the worker to encrypt symmetric data encryption key, e.g. RSA-OAEP-3072.

dataEncryptionAlgorithm is an optional string containing a comma-separated list of the supported data encryption algorithms, e.g. AES-GCM-256. For an AES GSM encrypted message starts with the tag that is followed by the cypher text.

workOrderPayloadFormats defines the formats for Work Order requests and responses. A Worker may support multiple formats. This specification currently defines the following payload formats (this list is expected to grow):

- JSON-RPC – the payloads are provided in JSON RPC format as defined in section RPC Encoding Conventions without using JWT format for signatures.
- JSON-RPC-JWT – the payloads are provided in JSON RPC format as defined in section RPC Encoding Conventions. In this case JWT format is used for the signatures.
- Custom Work Order payload formats should start with tilde "~".

workerTypeData contains the Worker type specific details defined below.

9.2 TEE Worker Data

This section currently includes details for Intel SGX. Details for other TEEs may be added in the future.

9.2.1 Intel SGX Worker Type Data

This subsection defines data for the "TEE-SGX" worker type. Refer to parameter workType in section Registering a New Worker.

Identification and attestation payload:

```
{
  "workerTypeData": {
    "verificationKey": <hex string>,
    "extendedMeasurements": [<string>],
    "proofDataType": <hex string>,
    "proofData": {...},
    "encryptionKey": <optional hex string>,
    "encryptionKeyNonce": <optional hex string>,
    "encryptionKeySignature": <optional hex string>,
    "enclaveCertificate": <optional string>
  } }
```

workerType is defined in section Worker Registry Smart Contract API.

verificationKey is a hex string representing an ECDSA/SECP256K1 public key used to verify signatures created by the Enclave. This field must be included in the proofData.

extendedMeasurements is a string that implements application-specific logic. For example, in the case of an enclave pool including more than one type of enclave (each having a different MRENCLAVE value), this parameter may contain a comma separated list of hexadecimal strings, one for each MRENCLAVE. See Load Balancing and Enclave pools

Requesters are expected to verify the Enclave measurements off-chain unless the blockchain client contains a pre-compiled contract that verifies the Enclave measurements on-chain at the Enclave's registration time. For details on the matching mechanism refer to https://software.intel.com/enus/sgx.

proofDataType is one of the "TEE-" prefixed data types. Currently only the Intel SGX proof data type is defined. More types may be defined later.

- TEE-SGX-IAS indicates that an Intel Attestation Server (IAS) issued a verification report for this worker.

proofData is the worker attestation data. Its format is defined below in sub-section "Proof Data Format".

encryptionKey is a hex string representing an RSA public key used to encrypt data sent to the Enclave. This field must be signed as described in encryptionKeySignature.

encryptionKeyNonce is a hex string. It is either a random number or a monotonic counter updated every time a new encryptionKey is generated.

encryptionKeySignature is a hex string representing a signature of encryptionKey signed with the worker's private signing key. The requester must verify the signature using 'verificationKey'. It is not required if 'enclaveCertificate` is provided.

enclaveCertificate is a string representing the enclave certificate signed by worker's private signing key, and must include the encryptionKey and encryptionKeyNonce. The enclave certificate should be in X.509 format.

Please note that at least one of the two options are mandatory for a requester's verification:

- Provide encryptionKey and encryptionKeySignature in the workerTypeData payload.
- Fill enclaveCertificate in the workerTypeData payload.

Proof Data Format

For `TEE_SGX-IAS`, the `proofData` is in the following format:

```
{
  “X-IASReport-Signature”:<required, base64>,
  ”X-IASReport-Signing-Certificate”: <required, string>,
  “Advisory-URL”: <optional, string>,
  “Advisory-IDs”: <optional, string>,
  “Verification-report”:<required string> }
```

Refer to IAS API for parameter descriptions.

Verification-report:isvEnclaveQuoteBody:REPORTBODY:REPORTDATA contains a SHA256 hash (first 32 bytes) of the concatenation of the decoded verificationKey and (UTF8) extendedMeasurements values. The last 32 bytes are set to zero.

9.2.2 TEE-SGX Load Balancing and Enclave pools

This specification assumes that Enclaves can be grouped into groups for better load balancing and manageability. In such a case the same verification and encryption keys are used to invoke workloads by any of the Enclaves in a pool.

- If the pool includes more than one type of enclave (each having a different MRENCLAVE value), extendedMeasurements includes a comma-separated list of hexadecimal strings, one for each MRENCLAVE. This list is provided to the enclave as an input by the hosting service.
- The master Enclave generates verification and encryption keys
- The master enclave calculates a SHA256 hash andd places it in Verificationreport:isvEnclaveQuoteBody:REPORTBODY:REPORTDATA as defined in section Intel SGX Worker Type Data.
- A master Enclave is registered in the worker registry
- The master Enclave uses the verification and encryption keys to forward and securely execute workloads, only through the enclaves on the list provided in step 1

The exact Enclave pool provisioning mechanism is outside the scope of this specification. The MRENCLAVE value included in the proofData:Verification-report can be used by the requester to determine what mechanism is used to operate the enclave pool.

9.3 MPC Worker Data

This section defines data for worker type MPC. Refer to parameter workType in section Registering a New Worker.

```
{
  "workerTypeData":{
    "workerType":<hex string>,   //"MPC-" prefix
    "verificationKey": <hex string>,
    "encryptionKey": <hex string>,
    "proofDataType": <hex string>,
    "proofData": <hex string>,
  }}
```

workerType is defined in section Worker Registry Smart Contract API.

verificationKey is a hex string representing a public key used to verify data provided by the MPC worker. This field must be included in the proofData.

encryptionKey is as a hex string representing an asymmetric public key used to encrypt data sent to the MPC worker. This field must be included in the proofData. proofDataType is one of the "MPC-" prefixed data types to be defined in the future.

proofData is proof data corresponding to the proofDataType. Refer to the section Implementation Notes for more details.

9.4 ZK Worker Data

This section defines data for worker types "ZK". Refer to parameter workType in section Registering a New Worker.

```
<dfn>Identification and attestation payload</dfn>
{
  "workerTypeData": {
    "workerType":<hex string>,    //"ZK-" prefix
    "verificationKey": <hex string>,
    "encryptionKey": <hex string>,
    "proofDataType": <hex string>,
    "proofData": <hex string>,
  } }
```

workerType is defined in section Worker Registry Smart Contract API.

verificationKey is as a hex string representing a public key used to verify proofs generated by the ZK worker. This field must be included in the proofData.

encryptionKey is as a hex string representing an asymmetric public key used to encrypt data sent to the ZK worker. This field must be included in the proofData. proofDataType is one of the "ZK-" prefixed data types to be defined in future.

proofData is proof data corresponding to the proofDataType. Refer to Implementation Notes chapter for more details.

10. Implementation Notes

This section is non-normative.

10.1 Note 1: Receipts

Work Order Receipts are created upon each successful or unsuccessful Work Order execution. They are created and signed by the Requester and later updated and signed by the Worker making

sure that parties involved in the transaction can use it for tracking, auditing, dispute resolution, and fraud detection. This specification supports per Work Order Receipt as an option.

It is left to the implementation to decide whether to store Receipts on the main blockchain or somewhere else. Similarly, when supporting high volume use-cases (e.g. an IOT use-case), it is left to the implementation to decide if Receipt updates to the main blockchain should be batched. Implementations may decide to keep Receipts off the main blockchain such as in a side-chain or trusted database.

10.2 Note 2: Worker Service

Details of a Worker Service are left for an implementation to decide. The following functions are expected:

- A Worker Service can assist in registering a Worker with either the main blockchain or an offchain registry.
- A Worker Service can support a single Worker or a group of Workers. Workers may be registered individually, enabling Requesters to select a particular Worker for Work Order execution, or multiple Workers may be registered and managed collectively as a Worker Pool. When a Requester selects a Worker Pool, the Worker Service chooses which Worker(s) in the pool will be used to execute a particular Work Order. Requesters should be able to look up all Workers belonging to the same Worker Service. In this specification, a Worker Pool is used synonymously with a Worker.
- A Worker is represented using a URI and an RSA and/or an ECDSA/SECP256K1 public key set by its Worker Service on the main blockchain. A URI can represent a single Worker or multiple individually addressable Workers, one or multiple Worker Pools, or any combination of individually addressable Workers or Worker Pools. All Workers in a pool represented by a single URI can share a common RSA and/or ECDSA/SECP256K1 public key set. Appropriate pooling of Workers is left for deployments to decide. For example, pools could be based on location of Workers, type of Workers, etc.
- A Worker Service can act as a proxy for signing Ethereum transactions originating in a Worker (e.g. the Worker Service has the Ethereum address that is used to create an Ethereum transaction that contains the updated Work Order Receipt generated by a Worker).
- The same from Ethereum address (and its associated account address) can be used by the Worker Service on behalf of all or some of its Workers. Alternatively, an individual Ethereum address can be assigned to each Worker.

- Proxy JSON APIs are for Work Order invocation by a Worker in direct mode.
- If a Worker Service supports the Work Order proxy invocation mode, it should deploy at least one Work Order Invocation Proxy smart contract, but may deploy more. The same smart contract address can be used for submitting Work Orders to multiple Workers.

Given that a Worker Service and a Worker can have 1-to-1 or 1-to-many relationships, the term Worker as used in this specification may imply a Worker Service as appropriate.

10.3 Note 3: Proof Data

This specification supports different work proof data types for different types of Trusted Computes. The following proof data types are currently defined:

- ***TEE-Proof*** is an attestation produced by a TEE that the requester can use to validate that the work order was executed inside a TEE.
- ***ZK-Proof*** is the witness produced by a prover that verifiers can use to verify a claim with zero-knowledge (for example, the correct execution of a piece of software on a specific input set with a specific output). The proof data for non-interactive zero-knowledge proofs contains the Witness and the Verification circuit together with the signature of the issuing trusted compute resource.
- ***MPC-Proof*** are the m outputs computed by the m compute resources each holding a share of the original input. The reconstruction of the shared m outputs yields the complete result of the program that was converted into a randomized logical circuit of AND and XOR gates and used to compute each output. The order of the gates itself has been randomized and both inputs and outputs are encrypted. After the circuit construction has been shared amongst all counterparties, the proof data consists of the m outputs of the application of the circuit signed by the trusted resource and distributed to all participating compute resources.
- ***DID-Proof*** refers to the DID document object (DDO) as defined by the W3C. The proof data contains the attesting DID and the information from the DDO required to verify the Worker. This proof data type is reserved and may be defined in detail in the future versions of this specification.
- Custom Proofs must start with a tilde "~"

10.4 Note 4: Security Consideration

The parameter`encryptedDataEncryptionKey` is defined in "Work Order Data Formats":

If the data item is sent encrypted with a one-time encryption key generated by a 3rd party that owns this data item (it may be different from the work order requester), encryptedDataEncryptionKey contains this encryption key in double encrypted format:

- First, it is encrypted with the worker's public encryption key (e.g. by a 3rd party that owns the data so the requester cannot see the data)
- Then the result of the previous encryption above is encrypted with the key from encryptedSessionKey(by the requester to enforce work order integrity)

The first step returns the encryption result that the data one-time encryption key is encrypted by a worker's public key.

The second step returns the encryption result that the first step encryption result is further encrypted by encryptedSessionKey.

The possible security issue is that since the first step encryption result refers to encryption by a worker's public key, this encryption result seems no longer a "secret" and can be transferred to different requesters to proceed with step 2. However, if this first step encryption result is disclosed, a man-in-the-middle attacker knowing a worker's public key can then intercept the requester payload, modify the payload parameter values, and generate the attacker's fake encryptedSessionKey which is then used to generate fake encryptedDataEncryptionKey as well as fake encryptedRequestHash, and the worker is not able to detect this malicious attack at the run time.

If a data item belongs to a 3rd party other than requester, we recommend that the implementation should be based on one of the following two options:

- Option 1: use the requesterSignature parameter in the payload: generate the requester payload's signature with the requester's private signing key, verified by the corresponding public key on the worker side.
- Option 2: if the requesterSignature parameter is not used, the third party should establish a secure channel with an authorized and verified requester to transfer the first step encryption result.

A. Additional Information

A.1 Terminology

A ***Requester*** is an entity that issues Work Orders using either a ĐApp or an application smart contract. Requesters are identified by an Ethereum address or a DID that can be resolved to an Ethereum address. Requester management is out of scope for this specification and will be covered as part of EEA Requester permissioning.

A ***Worker*** is a computational resource for Work Order execution. A Worker may be identified by an Ethereum address or a DID.

Trusted Compute is a trusted computational resource for Work Order execution. It preserves data confidentiality, execution integrity and enforces data access policies. All Workers described in this specification are also Trusted Compute. Trusted Compute may implement those assurances in various ways. For example, Trusted Compute can base its trust on software-based cryptographic security guarantees, a service's reputation, virtualization, or a hardware-based Trusted Execution Environment such as Intel's SGX.

A ***Trusted Execution Environment*** (***TEE***) is a hardware-based technology that executes only validated tasks, produces attested results, provides protection from malicious host software, and enxures confidentiality of shared encrypted data.

An ***Enclave*** is an instantiation of Trusted Compute within a hardware based TEE. Certain hardware based TEEs, including Intel SGX, allow multiple instances of Enclaves executing concurrently. For simplification, in this specification the terms TEE and Enclave are used interchangeably.

A ***Worker Service*** is an implementation dependent middleware entity that acts as a bridge for communications between Ethereum Blockchain and a Worker. A Worker Service may belong to an enterprise, a cloud service provider, or an individual sharing his or her available computational resources (subject to provisioning).

A ***Work Order*** (WO) is a unit of work submitted by a Requester to a Worker for execution. Work Orders may include one or more inputs (e.g. messages, input parameters, state, and datasets) and one or more outputs. Work Order inputs and outputs can be sent as part of the request or response body (a.k.a. inline) or as links to remote storage locations. Work Order inputs and outputs are normally sent encrypted.

The ***Direct Model*** is a Work Order execution model in which a Requester ĐApp directly invokes a JSON RPC network API for Work Order execution in a Worker.

The ***Proxy Model*** is a Work Order execution model in which a Work Order Invocation Proxy smart contract is used by an enterprise application smart contract to invoke Work Order execution in a Worker.

An ***Attested Oracle*** is a device that uses Trusted Compute to attest some data (e.g. environmental characteristics, financial values, inventory levels).

B. References

B.1 Normative references

[JSON-RPC]

JavaScript Object Notation - Remote Procedure Call. JSON-RPC Working Group. URL: http://www.jsonrpc.org/specification

[JSON-RPC-API]

Ethereum JSON-RPC API. Ethereum Foundation. URL: https://github.com/ethereum/wiki/wiki/JSON-RPC

[RFC2119]

Key words for use in RFCs to Indicate Requirement Levels. S. Bradner. IETF. March 1997. Best Current Practice. URL: https://tools.ietf.org/html/rfc2119

[RFC7519]

JSON Web Token (JWT). M. Jones; J. Bradley; N. Sakimura. IETF. May 2015. Proposed Standard. URL: https://tools.ietf.org/html/rfc7519

B.2 Informative references

[RFC8017]

PKCS #1: RSA Cryptography Specifications Version 2.2. K. Moriarty, Ed.; B. Kaliski; J. Jonsson; A. Rusch. IETF. November 2016. Informational. URL: https://tools.ietf.org/html/rfc8017

[secp256k1]

SEC2: Recommended Elliptic Curve Domain Parameters. Certicom Research. URL: http://www.secg.org/sec2-v2.pdf