# Enterprise Ethereum Alliance Off-Chain Trusted Compute Specification V0.5

15 October 2018

**Editors:**

Sanjay Bakshi (Intel)

Yevgeniy Yarmosh (Intel Lab)

Lei Zhang (iExec)

Andreas Freund (ConsenSys)

**Contributors:**

Bill Gleim (ConsenSys), Thomas Bertani (Oraclize), Jean-Charles Cabelguen (iExec), Ben Towne (SAE ITC), Dan von Kohorn (ConsenSys /Independent), George Polzer (Everymans.ai), Puneetha Karamsetty (blk.io), Tom Willis (Intel), Junji Katto (Itau)

---

## Abstract

This document specifies APIs that enable Off-Chain Trusted Compute for Enterprise Ethereum, to support private transactions, offload for compute intensive processing and attested Oracles.

## Legal Notice §

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 2019.

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 non-infringement 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 non-infringement 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.

## 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 approved by the EEA Board of Directors for publication. This pre-release specification was developed by the EEA Technical Specification Working Group and Trusted Execution Task Force for review, improvement, and publication as an EEA standard.

The TSWG *expects* at time of writing to produce a new revision of this specification for release in the second quarter of 2019 which would obsolete this version, either as a stand-alone specification or by integrating it into an updated version of the the Enterprise Ethereum Alliance Client Specification.

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

## Table of Contents



 

# 1. 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.
- Offload intensive computational processing from a main blockchain to an off-chain Trusted Compute for improved throughput and scalability.
- Enable Attested Oracles.

These objectives are achieved by executing some parts of a blockchain transaction off the main chain in off chain trusted compute. 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 of Trusted Compute used to create increased trust in an Oracle. Actual API refinements to support Attested Oracles will be addressed in the next release of this specification.

## 1.1 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 is proposed as way to improve blockchain performance in these areas. In this specification, a main blockchain maintains a single authoritative instance of the objects, enforces execution policies, and ensures transaction and results auditability, while associated off-chain trusted compute 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 Requestors, an Ethereum blockchain client and one or more Workers (supported by a Worker Service). Requestors submit Work Orders, and Workers execute those Work Orders. Work Order receipts can be recorded on the blockchain by the Ethereum clients running Smart Contracts. While each of the enterprises in figure 1 contain all three major components, this is not necessary. For example, Requestors 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.

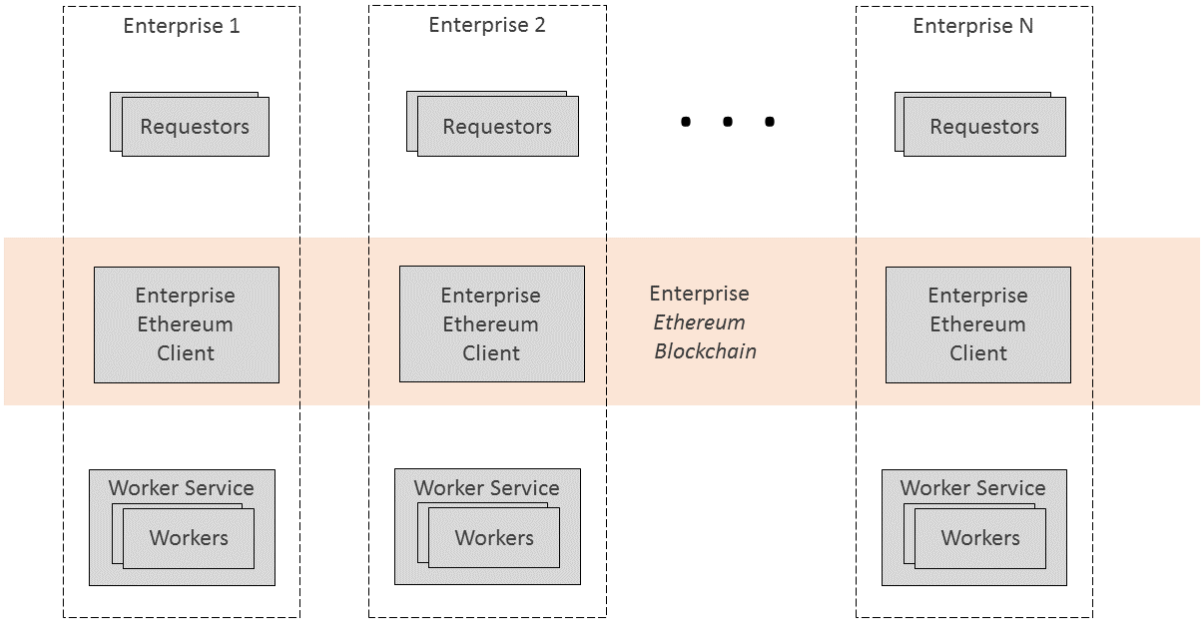


*Figure 1 Figure 1 Enterprise Ethereum Blockchain with Off-Chain Workers*

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

### 1.1.1 Invocation Models §

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

#### *1.1.1.1 Direct Model* §

In this invocation model Workers are invoked via JSON RPC network API. An organization registers its workers with on chain smart contract(s) where a ĐApp can discover them. Post discovery all the interactions between ĐApp and worker are done off the chain. Optionally, transaction logs aka receipts maybe be stored on the chain.

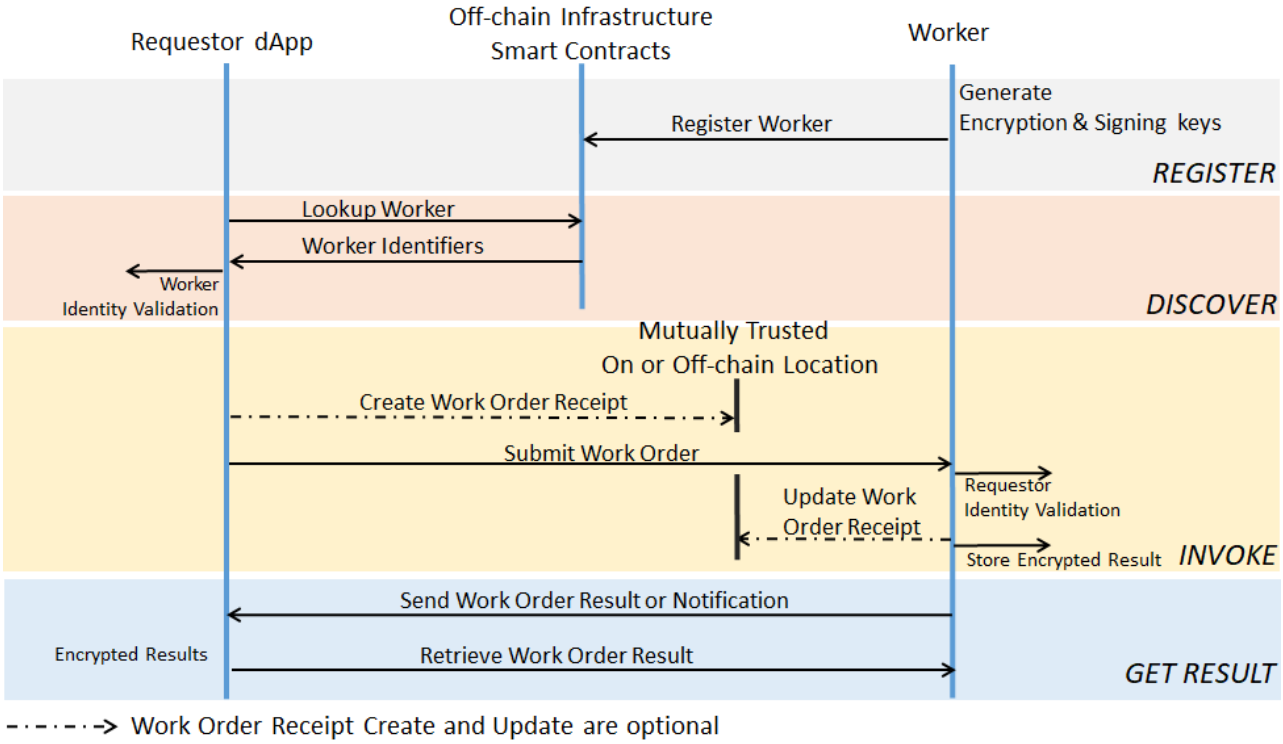


*Figure 2 Figure 2 Direct Invocation*

### *1.1.1.2 Proxy Model* §

In this invocation model Workers are invoked via a Work Order Invocation Proxy smart contract. The Proxy Model is typically used to support usages in which an application smart contract or a ĐApp does not prefer or cannot invoke a worker directly. An organization registers its workers with on chain smart contract(s) where a ĐApp or another requesting smart contract can discover them. Optionally, transaction logs aka receipts maybe be stored on the chain.

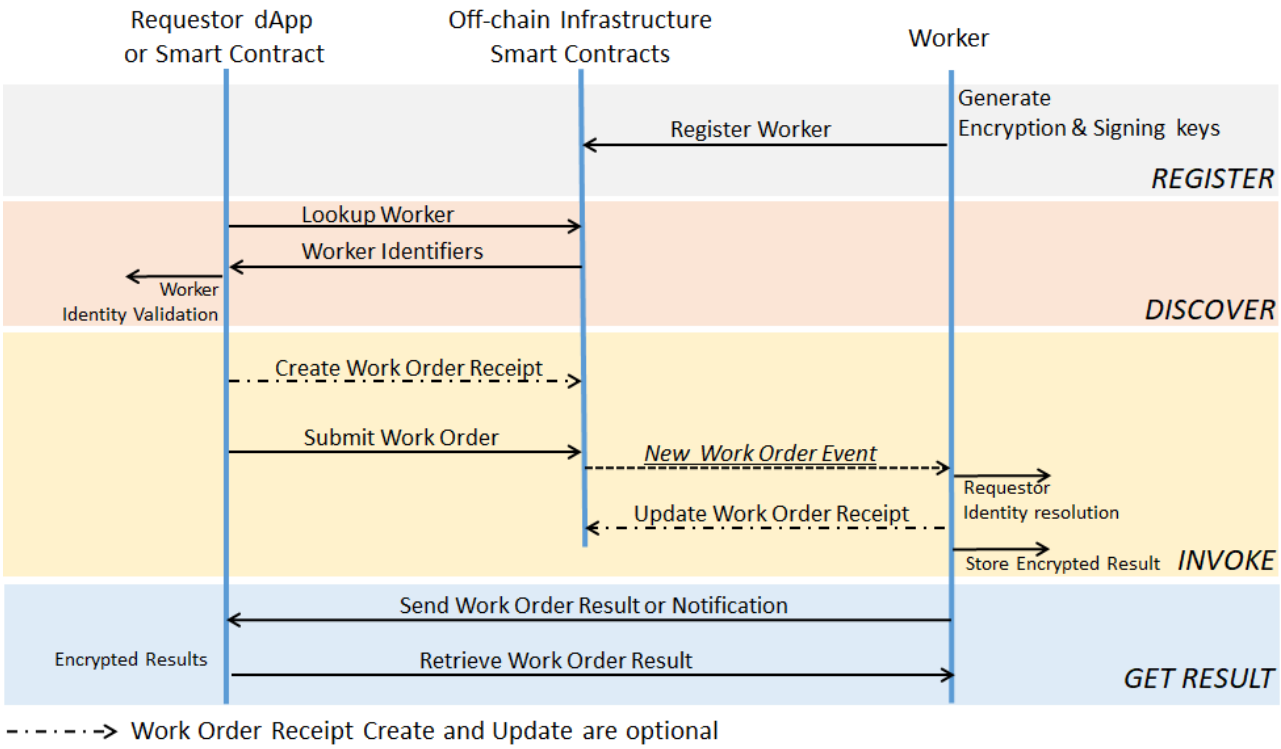


*Figure 3 Figure 3 Proxy Invocation*

This version of the specification addresses only stateless (from the Worker viewpoint) execution, e.g. the burden of maintaining state between Work Order invocations (if required) is on the caller.

A future version of the specification may include an additional model in which the off-chain logic, acts as both the requestor 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 relies on an external registry shared by contract participants.

## 2. 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.

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/.

## 3. 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 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 Requestors 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. 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, Requestor’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 Requestor Registry smart contract, optional Work Order Invocation Proxy smart contract, and/or optional Work Order Receipts smart contract.

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

# 4. 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]

## 4.1 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>
    }
}
```

### 4.1.1 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 collection of parameters as per JSON RPC specification, defines an error or status, includes

`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

`message` is a string describing the errors and corresponding to `code` value

`data` contains additional details about the error, format is error and implementation specific

## 4.2 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 will specify API-dependent Payload object only.

# 5. Worker Registry §

This chapter defines interfaces for registering Workers:

- Smart contract API that registers generic Worker information not specific to a Worker type
- JSON RPC API for registering generic Worker information
- JSON RPC API for registering type specific, e.g. TEE, MPC etc. Worker information

Refer to section Implementation Notes for a list of supported proof data types and related details.

## 5.1 Worker Registry Smart Contract API §

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

***Registering a New Worker***

This function registers a Worker and is invoked from the Ethereum address of the Trusted Resource Service using the web3 JSON-RPC Ethereum transaction model with the digital signature associated to the sending Ethereum address.

An Implementation specific model will be used to enforce authorization policies on who can make this call.

Inputs

`workerID` is a DID or a unique id (hexadecimal string) such as an Ethereum public key

`workerType` defines Worker type, currently defined types are: -

1. indicates "TEE" or Trusted Execution Environment
2. indicates "MPC" or Multi-Party Compute
3. indicates "ZK" or Zero Knowledge

For associated APIs for each type please refer to Appendix A.

4. indicates "OFF-CHAIN-REGISTRY". This record in the Worker Registry is not a Worker, but a record that defines how to connect to an Off-chain Worker Registry to discover Workers. In this case `workerTypeDataURI` defines a URI that can be used to access the Off-chain

Worker Registry using JSON RPC messages defined in section Off-chain Worker Registry JSON RPC API

`workerTypeDataURI` defines a URI from where additional Worker information can be retrieved. This URI provides data in a format that corresponds to the Worker type. Supported types:

- `TEE` - section "TEE workerTypeData" in Appendix A defines for retrieving identification and attestation payload for TEE worker types.
- `MPC` - section "MPC workerTypeData" in Appendix A defines identification and attestation payload for MPC worker types.
- `ZK` - section "ZK workerTypeData" in Appendix A defines identification and attestation payload for ZK worker types.
- `OFF-CHAIN-REGISTRY` – section "Off-Chain Worker Registry JSON RPC API" defines an API for retrieving generic Worker information from an Off-chain Worker Registry. In this case workerTypeDataUri redirects the client to an Off-chain Worker Registry.
- `DDOURI` defines a URI used only if a Worker ID is a DID

`organizationID` is an optional parameter. 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.

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

***Initiating Worker lookup***

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

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

If the list is too big to fit into a single response (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 has to be 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)
```

***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 bigger than numbers 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 that 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)
```

***Retrieving Worker Information***

This function retrieves information for the Worker and it can be called from any authorized public key (Ethereum address) or DID.

Inputs

`workerId` is id of a Worker to retrieve

Outputs

The same as input parameters to the corresponding call to `workerRegister`.

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

### 5.1.1 Off-Chain Worker Registry JSON RPC API §

These are the JSON RPC version of "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.

***Register Worker JSON Payload***

This message registers a Worker. It doesn't 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>,
        "workerType":<uint>,
        "workerTypeDataUri":<hex string>,
        "organizationId":<hex string>,
        "applicationTypeId": [<one or more hex strings>]
    }
}
```

`method` must be `WorkerRegister`,

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

***Worker Lookup JSON Request Payload***

This message registers 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 Lookup for description of the parameters

***Worker Lookup Next JSON Request Payload***

This message continues retrieving results initiated by a previous `WorkerLookUp` message. Its response is defined in section Worker Lookup JSON Response 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 description of the parameters

***Worker Lookup JSON Response Payload*** This payload is sent back to a Requestor in response to the request is defined in sections Worker Lookup JSON Request Payload and Worker Lookup Next JSON Request 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 the description of elements in this objects.

***Retrieve Worker JSON Request Payload***

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

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

`method` must be `WorkerRetrieve`,

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

***Retrieve Worker JSON Response Payload***

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

```
{
    "jsonrpc": "2.0",
    "id": <integer>,
    "result": {
        "workerType":<uint>,
        "workerTypeDataUri":<hex string>,
        "organizationId":<<ex string>,
        "applicationTypeId": [<one or more hex strings>]
    }
}
```

`result` is a collection of the response specific parameters. Refer to the output parameters in section Retrieving Worker information for the description of elements in this objects.

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

# 6. Work Orders §

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

## 6.1 Direct Model Invocation §

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

This API can be used in several modes:

- Synchronous request-response when the exchange of Work Order request and completion result happens in the same HTTP session. Synchronous mode is for Work Orders that don't require long time to execute.
- Result pulling mode. In this mode ĐApp disconnects after submitting a Work Order request and then periodically polls the JSON endpoint for the Work Order result
- Asynchronous mode. In this mode ĐApp provides a URI for receiving the Work Order result as a part of the Work Order request. ĐApp disconnects after submitting the Work Order. Upon the Work Order completion, the Worker submits the Work Order result to the provided URI.
- Notification mode. In this mode ĐApp provides a URI for receiving a notification about the Work Order completion as part of the Work Order request. ĐApp disconnects after submitting the Work Order. Upon the Work Order completion, the Worker sends an event to the URI provided in the Work Order request. Upon receiving the event, the client retrieves the Work Order result from the JSON endpoint.

***Work Order Request Payload***

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

```
{
    "jsonrpc": "2.0",
    "method": "WorkOrderSubmit",
    "id": <integer>,
    "params": {
        "responseTimeoutMSecs": <integer>,
        "requestorSignature": <hex string[]>,
        "requestorGeneratedNonce": <string>
        "workerId": <hex string>,
        "workOrderId": <hex string>,
        "requestorId": <hex string>,
        "resultUri": <string>,
        "notifyUri": <string>,
        "Data": [
            <BASE64 string> or <object>
        ]
    }
}
```

Parameters

`method` is set to `WorkOrderSubmit`

`params` is a collection of parameters as per JSON RPC specification. The parameters defined below

`responseTimeout` is a maximum timeout in milliseconds that the caller will wait for the response

`requestorSignature` is an optional parameter.

If the requestor decides to not sign this message, both `requestorGeneratedNonce` and `requestorSignature` are blank.

If the requestor decides to sign the message and `workOrderPayloadFormats` is set to `JSON-RPC-JWT` (refer to section `Register Worker JSON Payload`), the signature is in JSON Web Tokens (JWT) format and provided in `requestorGeneratedNonce` parameter below. `requestorSignature` is blank in this case.

If the requestor decides to sign the message and `workOrderPayloadFormats` is set to `JSON-RPC`, the digital signature is generated by the Requestor signing a SHA256 hash of the message containing `requestorGeneratedNonce`, `workOrderId`, `workerId`, `requestorID`, and its `dataHash` followed by `EncryptedDataEncryptionKey` for each input data item. The signature has the following array format

`[signed_message, SHA256_message, curve_type]`

where `curve_type` is either `RSA` or `secp256k1`. The Worker Service will verify this signature.

`requestorGeneratedNonce` is the hash of a random number generated by the Requestor as a part of signature above. It is an optional parameter, if the `requestorSignature` is blank. It is submitted either directly as a plain string or as a JWT with Payload set as follows:

Payload:

```
{
    "workOrderId": <string>
    "WorkerId": <string>
    "requestorId": <string>
}
```

`workerId` is either the worker public key or its DID used during registration and will be used to sign the work order results.

`workOrderId` is an id assigned to the Work Order by the Requestor and can be registered using the Work Order Receipts API

`requestorId` is either the Requestor's public key or its DID

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

`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 Result Submission.

`Data` 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.

If `workOrderPayloadFormats` is set to `JSON-RPC-JWT` (refer to section Register Worker JSON Payload), it is a JWT with Payload set as follows:

Payload:

```
{
    "Type": <string>,
    "dataHash": <hex string>,
    "inputDataURI": <string>,
    "outputDataURI": <string>,
    "BLOB": <BASE64 string>,
    "EncryptedDataEncryptionKey": <hex string>,
}
```

If `workOrderPayloadFormats` is set to `JSON-RPC-` (refer to section Register Worker JSON Payload), it is an array:

```
{
    "Type": <string>,
    "dataHash": <hex string>,
    "inputDataURI": <string>,
    "outputDataURI": <string>,
    "BLOB": <BASE64 string>,
    "EncryptedDataEncryptionKey": <hex string>,
}
```

`Type` defines an input data type. Supported data types are:

- `code` indicates that this `Data` items defines the application code to execute. It can be an id of the pre-built application in case of fully precompiled static Worker application. Alternatively, it can be a script, e.g. Solidity, that is loaded into the Workerat the runtime if the Worker has a corresponding built-in script interpreter
- `message` contains input parameters
- `state` is a persistent state persevered across relevant Work Order executions by the Requestor

- `dataset` additional data (like reference database) used by the application
- Custom data types that should start with tilde "~".

  `dataHash` is a SHA256 hash of the data

  `inputDataURI` indicates how the Work Order input data are provided for execution. It can be @@
- If `inputDataURI` is not present, null, or a null string, this is output data only
- If it is set to `#inline`, than the input data are sent inline in the `BLOB` parameter
- Otherwise, it contains a URL from where Worker Service must download the input data

  `outputDataURI` defines how the Work Order output data should be returned to the participant.
- If `outputDataURI` is not present, null, or a null string, this is input data only
- If it is set to `#inline`, than the output data will be sent inline in the `BLOB` parameter in Work Order Result Payload.
- Otherwise, it contains a URL to where Worker Service must upload the output data

  `BLOB` is an optional parameter that contains input data if `inputDataURI` is set to `#inline`

  `EncryptedDataEncryptionKey` represents a symmetric key used to encrypt this item input and/or output data. The key itself is sent in the encrypted form. It is encrypted using the Worker encryption key

  After a Work Order request is received, the Worker Service can respond in one of three ways:
- Complete a (short running) Work Order and returns 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

  ***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>,
        "workerId": <hex string>,
        "requestorId": <hex string>,
        "workerSignature": <hex string[]>,
        "workerGeneratedNonce": <string>,
        "Data": [
            <BASE64 string> or <object>
        ]
    }
}
```

Parameters

`result` is a collection of parameters as per JSON RPC specification. The parameters defined below are

`workOrderId` is a Work Order sent in the Work Order request

`workerId` is id of the worker who completed the work order

`requestorId` is id of the requestor who submitted the work order

`workerSignature` is an optional parameter.

If the worker decides to not sign this message, both `workerSignatureNonce` and `workerSignature` are blank.

If the worker decides to sign the message and `workOrderPayloadFormats` is set to `JSON-RPC-JWT` (refer to section Register Worker JSON Payload), the signature is in JSON Web Tokens (JWT) format and provided in `workerSignatureNonce` parameter below. `workerSignature` is blank in this case.

If the worker decides to sign the message and `workOrderPayloadFormats` is set to `JSON-RPC`, the digital signature is generated by the Requestor signing a SHA256 hash of the message containing `workerSignatureNonce`, `workOrderId`, `workerId`, `requestorId`, and `dataHash` followed by `EncryptedDataEncryptionKey` for each input data item (from the corresponding request message) and for each output data item (in this message). The signature has the following array format

`[signed_message, SHA256_message, curve_type]`

where `curve_type` is either `RSA` or `secp256k1`. The Worker Service will verify this signature.

`workerSignatureNonce` is the `requestorGeneratedNonce` from the submitted work order. It is an optional parameter, if the `workerSignature` is blank. It is submitted either directly as an insecure string or as a JWT with Payload set as follows:

Payload:

```
{
    "workOrderId": <string>
    "WorkerId": <string>
    "requestorId": <string>
    "dataHash": <hex string>
}
```

`Data` is either a JWT or an array of one or more Work Order outputs, e.g. state, result

If `workOrderPayloadFormats` is set to `JSON-RPC-JWT` (refer to section Register Worker JSON Payload), it is a JWT with Payload set as follows:

Payload:

```
{
    "Type": <string>,
    "dataHash": <hex string>,
    "outputDataURI": <string>,
    "BLOB": <BASE64 string>,
    "EncryptedDataEncryptionKey"
}
```

If `workOrderPayloadFormats` is set to `JSON-RPC` (refer to section Register Worker JSON Payload) , it is an array:

```
{
    "Type": <string>,
    "dataHash": <hex string>,
    "outputDataURI": <string>,
    "BLOB": <BASE64 string>,
    "EncryptedDataEncryptionKey"
}
```

`Type` defines an output data type. Supported data types are:

- `result` contains return value(s) from the Work Order execution
- `state` is a persistent state to be persevered across repeated invocation associated with the same Work Order executions by the Requestor

- `dataset` additional data (like reference database) used by the application
- Custom data types that should start with tilde "~".

  `dataHash` is a SHA256 hash of the data element `outputDataURI` is the same as provided in a corresponding Work Order request.

  `BLOB` is an optional parameter that contains output data if `outputDataURI` is set to `#inline`". The data are encrypted with `EncryptedWorkOrderKey` from the Work Order request and then BASE64 encoded.

  ***Work Order Status or Error Response Payload***

  If the work request fails or rejected or scheduled for the execution later or its execution requires 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 "pending" – scheduled to be executed, but not started yet
- 6 means that the Work Order status "processing" – its execution started, but it is 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` is a Work Order id as a hexadecimal string

  ***Work Order Result Pull Payload***

If a Requestor 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 Requestor has two pulling options

*Poll 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 Handling for more details.

In either case the Work Order Result Pull request must follow the format below

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

Parameters

`method` is set to `WorkOrderGetResult`

`params` is a collection of parameters as per JSON RPC specification. The parameters defined below

`workOrderId` is a Work Order id that was sent in the corresponding `WorkOrderSubmit` request

`requestorSignature` is an optional parameter.

If the requestor decides to not sign this message, both `requestorGeneratedNonce` and `requestorSignature` are blank.

If the requestor decides to sign the message and `workOrderPayloadFormats` is set to `JSON-RPC-JWT` (refer to section Register Worker JSON Payload), the signature is in JSON Web Tokens (JWT) format and provided in `requestorGeneratedNonce` parameter below. `requestorSignature` is blank in this case.

If the worker decides to sign the message and workOrderPayloadFormats is set to `JSON-RPC`, the digital signature is generated by the Requestor signing a SHA256 hash of the message containing `requestorGeneratedNonce` and `workOrderId`. The signature has the following array format

`[signed_message, SHA256_message, curve_type]`

where `curve_type` is either `RSA` or `secp256k1`. The Worker Service will verify this signature.

`requestorGeneratedNonce` is the original requestorGeneratedNonce, if one was created. If not, it is blank. It is submitted either directly as an insecure string or as a JWT with Payload set as follows:

Payload:

```
{
    "workOrderId": <string>
}
```

***Work Order Result Submission***

If the client provides `resultUri` in the Work Order request payload, the Worker will send the Work Order execution result to provided URI.

In this case the Worker submits the result in the same format as defined in section Work Order Result Payload. The client responds with a payload as defined in section Work Order Status or Error Response Payload.

Work Order Completion Event

If the client provides `notifyUri` in the Work Order request payload, the Worker sends an event to the Requestor upon the Work Order completion irrespective 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 parameters as per JSON RPC specification. The parameters defined below are

`workOrderId` is a Work Order sent in the Work Order request

Upon receiving this event, the client will pull the Work Order result as defined in sections Work Order Result Pull Payload and Work Order Result Payload above



©2018 Enterprise Ethereum Alliance inc. All Rights Reserved

## 6.2 Proxy Model Invocation §

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

***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 Requestor's address. The Proxy Smart Contract is called by a ĐApp or an enterprise application from the Requestor's public key or DID.

This function must be called from the same Requestor public key that is included in the request either directly or through the DID.

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 jsonRequest.

`workerId` should match the corresponding field in the jsonRequest.

`requestorId` is a public key or DID. It should match the corresponding field in the jsonRequest.

`workOrderRequest` is a Work Order request data in one of the formats defined in section Registering a New Worker.

If the Work Order payload format for the Worker is set to `JSON-RPC`, the details of the `workOrderRequest` are defined in section Direct Model Invocation.

If the Work Order payload format for the Worker is set to `JSON-RPC-JWT` and to minimize smart contract storage, an `inputDataURI` is provided in the Work Order Request Data, and since we want to provide maximum security, the `workOrderRequest` field is a JWT with Payload set as follows:

Payload:

```
{
    "inputDataURI": <string>
}
```



©2018 Enterprise Ethereum Alliance inc. All Rights Reserved

```
function workOrderSubmit(
    bytes32 workOrderId,
    bytes32 workerId,
    bytes32 requestorId,
    string workOrderRequest) public
```

***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

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

`workOrderId` is the submitted Work Order id

`workerId` – id of the Worker that supposed to process the data (public key or DID)

`senderAddress` is an Ethereum address from which a corresponding `workOrderSubmit` call was done

`workOrderRequest` is the submitted Work Order request data as defined above

```
event workOrderNew (byte4 version,
    bytes32 workOrderId,
    bytes32 workerId,
    address senderAddress,
    string workOrderRequest)
```

***Completing a Work Order***

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

This function can be executed only from the WorkerService address that deployed this contract.

It emits event `workOrderDone`.

Inputs

`workOrderId` is an id of the Work Order, the same as provided during call to workOrderSubmit.

`workOrderResponse` is a Work Order response data in one of the formats defined in section Registering a New Worker.

If the Work Order payload format for the Worker is set to `JSON-RPC`, the details of the `workOrderRequest` are defined in section Direct Model Invocation.

If the Work Order payload format for the Worker is set to `JSON-RPC-JWT` and to minimize smart contract storage, a `workOrderRequestRedirectURI` is provided in the Work Order Request Data, and since we want to provide maximum security, the `workOrderRequest` field is a JWT with Payload set as follows:

Payload:

```
{
    "workOrderRequestRedirectURI": <string>
}
```

```
function workOrderComplete(bytes32 workOrderId, string workOrderResponse)
```

***Work Order Done Event***

This event is emitted by `workOrderComplete` where `workOrderId` is the completed Work Order id.

This event is intended for the Requestor who submitted the Work Order.

Parameters

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

`workOrderId` is the completed Work Order Id

`requestorId` is an Id of the Requestor that submitted the Work Order

`workOrderResponse` is Work Order response data as defined above.

```
event workOrderDone(bytes4 version,
    bytes32 workOrderId,
    bytes32 requestorId,
    string workOrderResponse)
```

***Retrieving Work Order Request Information***

This is an optional function that returns Work Order request info.

It is recommended to allow this function to be called only from the `WorkerService` address that instantiated this contract.

Inputs

`workOrderId` is an id of the Work Order to retrieve request for Outputs

`workOrderRequest` is the Work Order request data submitted during `workOrderSubmit` call.

`senderAddress` is an Ethereum address from which a corresponding `workOrderSubmit` call was made

```
function workOrderGetRequest(bytes32 workOrderId) public view
    returns (string workOrderRequest,
        address senderAddress)
```

***Retrieving Work Order Response Information***

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

Inputs

`workOrderId` is an id of the Work Order to retrieve response for Outputs

`workOrderResponse` is a Work Order response data provided during `workOrderComplete` call.

`requestorId` is the public key or the DID of the Requestor that submitted the Work Order

`workerId` is the public key used by the Worker to sign the Work Order results or the DID

```
function workOrderGetResult(bytes32 workOrderId) public view
    returns(
        string workOrderResponse,
        bytes32 requestorId,
        bytes32 workerId)
```

## 7. Work Order Receipts §

This chapter defines two modes for Work Order verification defined in this specification:

- Smart contract mode when the Work Order Receipts are managed by a Work Order Receipts smart contract. This mode is applicable when workOrderPayloadFormats is set to `JSON-RPC`. Refer to section Register Worker JSON Payload.
- Direct mode when the Work Order Receipts are managed by off-chain using Work Order Receipts JSON RPC API. This mode is applicable when `workOrderPayloadFormats` is set to `JSON-RPC` or `JSON-RPC-JWT`. Refer to section Register Worker JSON Payload.

## Proxy Model Receipt Handling §

This API provides following capabilities:

- Creating Work Order receipt either explicitly by the Requestor or on behalf of the Requestor by the Work Order Invocation Proxy smart contract.
- Updating Work Order by the Worker that executed the Work Order.
- Retrieving a Work Order receipt information.
- Looking up Work Order receipts by Id of either Requestor or Worker or a combination of them. Work Order receipt status also can be used to refine the lookup.

It should be noted that a Receipt includes a signature generated by a Worker. The ĐApp may verify it off chain, but an application smart contract has two options

- Precompiled Receipts Validation smart contract
- Go to an authorized Receipts Validation Worker

Choice and implementation details of Receipt Validation are left to implementations of the API.

***Creating a Work Order Receipt***

This function is implemented by Work Order Receipts smart contract and is called by a Requestor or on behalf of a Requestor 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

`workerServiceId` is the Ethereum public key or DID of the Worker (`WorkerService`) to execute the Work Order

`workerId` is the public key used by the Worker to sign the Work Order results or the DID

`requestorId` is a public key of the requestor

`workOrderId` is an id of the Work Order

```
function workOrderReceiptCreate(bytes32 workerServiceId,
    bytes32 workerId,
    bytes32 requestorId,
    bytes32 workOrderId) public
```

***Completing a Work Order Receipt***

This API is implemented by a Work Order Receipts smart contract and is called by the Worker to update the Receipt upon completing the Work Order.

It must be called from the `WorkerService` public key or DID passed during the receipt creation.

Inputs

`workOrderId` is an id of the Work Order

`requestorId` is the public key or the DID of the Requestor that submitted the Work Order, and must match the public key or the DID of the sender who created the Work Order receipt

`workerId` is an id of the worker that completed the work order

`inputDataHashes*` is an array of hashes of input data

`outputDataHashes*` is an array of hashes of output data

`inputEncryptionKeys*` is an array of inputs encryption keys

`outputEncryptionKeys*` is an array of outputs encryption keys

`workerNonce*` is the `requestorGeneratedNonce`

`workerSignature` is a signature generated by the worker. All parameters above are included in the signature. Refer to the section Direct Mode Invocation for details.

`status` is a number indicating the Work Order execution status. It is not included in the signature. One of

- `0` – successfully completed
- `1` – the Work Order was rejected before the execution started
- `2` – the Work Order execution failed
- optional additional status values indicating a failure

Refer to Work Orders for more details

```
function workOrderReceiptComplete(bytes32 workOrderId,
    bytes32 requestorId,
    bytes32 workerId,
    bytes32[] inputDataHashes,
    bytes32[] outputDataHashes,
    bytes32[] inputEncryptionKeys,
    bytes32[] outputEncryptionKeys,
    bytes32 workerNonce,
    bytes workerSignature,
    uint8 status) public
```

***Retrieving a Work Order Receipt***

Authorized entities can retrieve a Work Order receipt using this function

Inputs

`workOrderId` is an id of the Work Order to retrieve

Outputs

`receiptCreatorAddress` is an address used to make a corresponding `workOrderReceiptCreate` call

For other parameter definitions refer to `workOrderReceiptComplete` function above.

```
function workOrderReceiptRetreive(bytes32 workOrderId) public view
    returns(address receiptCreatorAddress,
        bytes32 requestorId,
        bytes32[] inputDataHashes,
        bytes32[] outputDataHashes,
        bytes32[] inputEncryptionKeys,
        bytes32[] outputEncryptionKeys,
        bytes32 workerNonce,
        bytes workerSignature,
        uint8 status)
```

***Work Order Receipt Completed Event***

In order to minimize storage utilization the smart contract implementation should maintain a relatively small number of the receipts in the storage, e.g. only receipts for Work Orders that have not completed.

The smart contract is should use Ethereum log for completed receipts.

An event below is emitted by function `workOrderReceiptComplete`.

```
event workOrderReceiptCompleted(
    bytes32 workOrderId,
    byte32 workerId,
    byte32 requestorId,
    byte32[] inputDataHashes,
    byte32[] outputDataHashes,
    byte32 workerNonce,
    bytes workerSignature,
    uint8 status)
```

***Work Order Receipt Lookup***

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

Inputs

`workerServiceId` is the Ethereum public key or DID of the Worker (`WorkerService`) to execute the Work Order

`workerId` is the public key used by the Worker to sign the Work Order results or the DID

`requestorId` is the public key or the DID of the Requestor that submitted the Work Order, and must match the public key or the DID of the sender who created the Work Order receipt

`status` is a string indicating the Work Order execution status. It is not included in the signature. Refer to `workOrderReceiptComplete` for the list of supported values.

Outputs

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

`lookupTag` is an optional parameter. If it is returned, it means that there are more matching receipts that can be retrieved by calling `workOrderReceiptLookUpNext` and 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 requestorId,
    uint8 status) public view
    returns(
        int totalCount,
        string lookUpTag, //OPTIONAL
        bytes32[] ids)
```

***Work Order Receipt Lookup Next***

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

Inputs

`workerServiceId` is the Ethereum public key or DID of the Worker (`WorkerService`) to execute the Work Order

`workerId` is the public key used by the Worker to sign the Work Order results or the DID

`requestorId` is the public key or the DID of the Requestor that submitted the Work Order, and must match the public key or the DID of the sender who created the Work Order receipt

`status` is a string indicating the Work Order execution status. It is not included in the signature. Refer to `workOrderReceiptComplete` for the list of supported values.

`lastLookupTag` is returned by a previous call to either this function or to `workOrderReceiptLookUp`.

Outputs

`totalCount` is a 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 and with this tag as an input parameter.

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

```
function workOrderReceiptLookUpNext(byte32 workerServiceId,
    bytes32 workerId,
    bytes32 requestorId,
    uint8 status,
    string lastLookUpTag) public view
    returns(
        int totalCount,
        string lookUpTag,
        bytes32[] ids)
```

## 7.1 Direct Model Receipt Handling §

Work Order Receipt management can be done off-chain by a dedicated service or by a workerService itself. This is expected to help scalability.

This section defines Work Order Receipts JSON RPC API to support this case. Requestors and Workers invoke this API without relying on blockchain smart contract, 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. Asynchronous mode is not defined in this revision, but may be added later.

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 doesn't assume any return values, e.g. creating a Work Order receipt.

***New Work Order Receipt Request Payload***

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

```
{
    "jsonrpc": "2.0",
    "method": "WorkOrderReceiptCreate",
    "id": <integer>,
    "params": {
        "workOrderId": <hexadecimal string>
        "workerServiceId": <hexadecimal string>,
        "workerId": <hexadecimal string>,
        "requestorId":< <hexadecimal string>,
        "requestorGeneratedNonce": <hexadecimal string>, //OPTIONAL
        "requestorSignature":< <hexadecimal string>, //OPTIONAL

        "inputData": [
            <ether base64 string or an array element>
        ]
    }
}
```

`method` must be `WorkOrderReceiptCreate`,

`params` is a collection of the request parameters below

`workOrderId` is an id assigned to the Work Order by the Requestor

`workerServiceId` is an Ethereum public key or DID of the Worker (`WorkerService`) to which a Work Order submitted

`workerId` is either the worker public key or its DID used during registration. In either case, worker id will be used to sign the work order results

`requestorId` is either the Requestor's public key or its DID

`requestorSignature` is an optional parameter.

If the requestor decides to not sign this message, both `requestorGeneratedNonce` and `requestorSignature` are blank.

If the requestor decides to sign the message and `workOrderPayloadFormats` is set to `JSON-RPC-JWT` (refer to section Register Worker JSON Payload), the signature is in JSON Web Tokens (JWT) format and provided in `requestorGeneratedNonce` parameter below. `requestorSignature` is blank in this case.

If the requestor decides to sign the message and `workOrderPayloadFormats` is set to `JSON-RPC`, the digital signature is generated by the Requestor signing a SHA256 hash of the message containing `requestorGeneratedNonce`, `workOrderId`, `workerId`, `requestorID`, and `dataHash` and `encryptedDataEncryptionKey` values for all input data. The signature has the following array format

`[signed_message, SHA256_message, curve_type]`

where `curve_type` is either `RSA` or `secp256k1`. The service will verify this signature.

`requestorGeneratedNonce` is the hash of a random number generated by the Requestor as a part of signature above. It is an optional parameter, if the `requestorSignature` is blank. It is submitted either directly as an insecure string or as a JWT with Payload set as follows:

Payload:

```
{
    "workOrderId": <string>
    "workerId": <string>
    "requestorID": <string>
    "dataHash": <hex string>
}
```

`Data` 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.

If `workOrderPayloadFormats` is set to `JSON-RPC-JWT` (refer to section Register Worker JSON Payload), it is a JWT with Payload set as follows:

Payload:

```
{
    "dataHash": <hex string>,
    "encryptedDataEncryptionKey": <hex string>
}
```

If `workOrderPayloadFormats` is set to `JSON-RPC` (refer to section Register Worker JSON Payload), it is an array:

```
{
    "dataHash": <hex string>,
    "encryptedDataEncryptionKey": <hex string>
}
```

`dataHash` is a SHA256 hash of the data

`encryptedDataEncryptionKey` represents a symmetric key used to encrypt this item input and/or output data. The key itself is sent in the encrypted form. It is encrypted using the Worker encryption key

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

***Work Order Receipt Update Request Payload***

This request is sent by a Worker that completed the Work Order to update its Receipt. This request doesn't have a specific corresponding response payload - a generic error response is returned.

```
{
    "jsonrpc": "2.0",
    "method": " WorkOrderReceiptUpdate",
    "id": <integer>,
    "params": {
        "workOrderId": <hexadecimal string>,
        "requestorId": <hexadecimal string>,
        "inputDataHashes":[  <hexadecimal strings> ],
        "outputDataHashes":[  <hexadecimal strings> ],
        "workerNonce": <hexadecimal string>,
        "workerSignature": <hexadecimal string>,
        "status": <number>
    }
}
```

`method` must be `WorkOrderReceiptUpdate`,

`params` is a collection of the request parameters below

`workOrderId` is an id of the Work Order

`requestorId` is either the Requestor's public key or its DID

`inputDataHashes` is an array of hashes of input data

`outputDataHashes*` is an array of hashes of output data

`workerNonce*` is the `requestorGeneratedNonce`

`workerSignature` is a signature generated by the worker. All parameters above are included in the signature. Refer to the section Direct Mode Invocation for details.

`status` is a number indicating the Work Order execution status. It is not included in the signature. Refer to section Completing a Work Order Receipt for details.

Refer to Work Orders for more details

***Work Order Receipt Retrieval Request Payload***

This request is sent by a Requestor to retrieve a Work Order Receipt id. The response to this request is defined in section Work Order Receipt Retrieval Response Payload.

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

`method` must be `WorkOrderReceiptRetrieve`,

`params` is a collection of the request parameters below

`workOrderId` is an id of the Work Order to retrieve

***Work Order Receipt Retrieval Response Payload***

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

```
{
    "jsonrpc": "2.0",
    "id": <integer>,
    "result": {
        "workOrderId": <hexadecimal string>,
        "workerServiceId": <hexadecimal string>,
        "workerId": <hexadecimal string>,
        "requestorNonce": <hexadecimal string>,
        "requestorId":< <hexadecimal string>,
        "requestorSignature":< <hexadecimal string>,
        "inputDataHashes":[  <hexadecimal strings> ],
        "outputDataHashes":[  <hexadecimal strings> ],
        "inputEncryptionKeys":[  <hexadecimal strings> ],
        "outputEncryptionKeys":[  <hexadecimal strings> ],
        "workerNonce": <hexadecimal string>,
        "workerSignature": <hexadecimal string>,
        "status": <string>
    }
}
```

`result` is a collection of the response specific parameters

`workOrderId` is an id of the Work Order

`workerServiceId` is the public key or the DID of the Worker (Worker Service) to execute the Work Order, hexadecimal string

`workerId` is the public key or DID used by the Worker to sign the Work Order results, hexadecimal string `requestorGeneratedNonce` is a nonce generated by the Requestor. Refer to section New Work Order Receipt Request Payload for details.

`requestorSignature` is generated by the Requestor. Refer to section New Work Order Receipt Request Payload for details.

`requestorId` is the public key or DID of Requestor that submitted the request Its value must match the address of the sender who created the Work Order receipt

`inputDataHashes*` is an array of hashes of input data

`outputDataHashes*` is an array of hashes of output data

`inputEncryptionKeys*` is an array of input encryption keys

`outputEncryptionKeys*` is an array of output encryption keys

`workerNonce*` is the requestorGeneratedNonce

`workerSignature` is a signature generated by the worker. All parameters above are included in the signature. Refer to the section Direct Mode Invocation for details.

`status` is a string indicating the Work Order execution status. It is not included in the signature. Refer to `workOrderReceiptComplete` for the list of supported values.

Refer to Work Orders for more details

***Work Order Receipt Lookup Request Payload***

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

The response to this request is defined in section Work Order Receipt Lookup Response Payload. Note that the response may not provide a complete list of matching ids, and the Requestor 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>,
        "requestorId": <hex string>,
        "status": <hex string>
    }
}
```

`method` must be `WorkOrderReceiptLookUp`,

`params` is a collection of the request parameters below

`workerServiceId` is the public key or the DID of the Worker or Worker Service to execute the Work Order, expressed as a hexadecimal string

`workerId` is the public key or DID used by the Worker to sign the Work Order results, hexadecimal string

`requestorId` is the public key or DID of Requestor that submitted the request. Its value must match the address of the sender who created the Work Order receipt

`status` is a string indicating the Work Order execution status. It is not included in the signature. Refer to `workOrderReceiptComplete` for the list of supported values.

***Work Order Receipt Lookup Response Payload***

This payload is sent back to a Requestor in response to the request defined in sections Work Order Receipt Lookup Request Payload and Work Order 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 response specific parameters

`totalCount` is a total number of receipts matching the lookup criteria. If this number is bigger than size of ids array, the caller should send Work Order Receipt Lookup Next Request Payload to retrieve the rest of the ids.

`lookupTag` is an optional parameter. If it is returned, it means that there are more matching receipts that can be retrieved by sending Work Order Receipt Lookup Next Request Payload with this tag as an input parameter.

`ids` is an array of the Work Order receipt ids that match the input parameters

***Work Order 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 Work Order Receipt Lookup Request Payload. Since the call may not return the complete list if ids, more than one Work Order Receipt Lookup Next Request Payload call 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>,
        "requestorId": <hexadecimal string>,
        "status": < hexadecimal string>,
        "lastLookUpTag": <hexadecimal string>
    }
}
```

`method` must be `WorkOrderReceiptLookUpNext`,

`params` is a collection of the request parameters below workerServiceId is the public key or the DID of the Worker (Worker Service) to execute the Work Order, hexadecimal string

`workerId` is the public key or DID used by the Worker to sign the Work Order results, hexadecimal string

`requestorId` is public key or DID of Requestor that submitted the request, must match the address of the sender who created the Work Order receipt

`status` is a string indicating the Work Order execution status. It is not included in the signature. Refer to `workOrderReceiptComplete` for the list of supported values.

`lastLookupTag` is returned by the previous response to either this Work Order Receipt Lookup Next Request Payload or the original lookup request. Refer to section Work Order Receipt Lookup Response Payload.

# 8. Appendix A: `workerTypeData` URI JSON RPC API §

This appendix defines a JSON RPC API for registering Worker type specific information such that it can be retrieved via a workerTypeDataURI. All messages follow a request-response pattern and are completed synchronously during the same session.

***Register Worker JSON Request Payload***

This message registers a Worker. It doesn't 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>,
        "workOrderInvocationAddress": <hex string>,
        "workOrderInvocationUri": <hex string>,
        "receiptInvocationAddress": <hex string>,
        "receiptInvocationUri": <hex string>,
        "fromAddress": <hex string>,
        "workOrderPayloadFormats": [<hex string>],
        "workerTypeData" : { …}
    }
}
```

`method` must be `WorkerRegister`,

`params` is an object containing the request parameters below

`workerId` is a unique Worker Id as a hex string or DID. It must match the id in the Worker Registry.

`workOrderInvocationURI` is a URI that should be used to submit Work Orders to a Worker in the direct mode. Multiple Workers belonging to a 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 mode. Multiple Workers such as Workers belonging to a same organization can share the same URI.

`receiptInvocationAddress` is an address of the Work Order Receipt smart contract that should be used to manage Work Order receipts processed by a Worker in the proxy mode. Multiple Workers can share the same address.

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

`fromAddress` is an Ethereum address that is used by or on behalf of this Worker to submit transactions.

`workOrderPayloadFormats` defines in what formats for Work Order requests and responses. A Worker may support multiple formats. This specification defines following payload formats and the list 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 specific details in the format defined in `workOrderPayloadFormats`

***Set Worker Status JSON Request Payload***

This message sets a Worker's status. It doesn't 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>,
        "status": <hex string>
    }
}
```

`method` must be `WorkerSetStatus`,

`params` is an object containing the request parameters below

`workerId` is a unique Worker Id as a hex string or DID

`status` can be one of `active`, `offline`, `decommissioned`, or `compromised`

***Retrieve Worker Type Info JSON Request Payload***

This message retrieves information about the type of a Worker, and type-specific data, based on its *Workerid*. The response is defined in section Retrieve Worker Type Info JSON Response Payload.

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

`method` must be `WorkerRetrieveTypeInfo`,

`params` is an object containing the request parameters below

`workerId` is a unique Worker Id as a hex string or DID

***Retrieve Worker Type Info JSON Response Payload***

This payload is sent back to a Requestor in response to the request defined in section Retrieve Worker Type Info JSON Request Payload.

```
{
    "jsonrpc": "2.0",
    "id": <integer>,
    "result": {
        "workerId":<hex string>,
        "workOrderInvocationAddress": <hex string>,
        "workOrderInvocationUri": <hex string>,
        "receiptInvocationAddress": <hex string>,
        "receiptInvocationUri": <hex string>,
        "fromAddress": <hex string>,
        "workOrderPayloadFormats": <hex string>,
        "workerTypeData" : {<JSON object>},
        "status": <string>
    }
}
```

`result` is a collection of the response specific parameters. Refer to the output parameters in sections Register Worker JSON Request Payload and Set Worker Status JSON Request Payload for the description of the elements in this object.

`workerId` is a unique Workerid as a hex string or DID. It must match the id in the Worker Registry.

`workOrderInvocationURI` is a URI that should be used to submit Work Orders to a Worker in the direct mode. Multiple Workers belonging to a 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 mode. Multiple Workers such as Workers belonging to a same organization can share the same URI.

`receiptInvocationAddress` is an address of the Work Order Receipt smart contract that should be used to manage Work Order receipts processed by a Worker in the proxy mode. Multiple Workers can share the same address.

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

`fromAddress` is an Ethereum address that is used by or on behalf of this Worker to submit transactions.

`workOrderPayloadFormats` defines in what format Work Order requests and responses are provided for and by this Worker. This specification currently defines the following payload formats:

- `JSON-RPC` – the payloads are provided in JSON RPC format as defined in section Direct Model Invocation.
- Custom Work Order payload formats should start with tilde "~".

`workerTypeData` contains the Worker specific details, using the format defined in `workOrderPayloadFormats`

`status` is a current status of the Worker

***TEE workerTypeData***

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

Intel SGX workerTypeData

Proof Data

`TEE-SGX-IAS` indicates that an Intel Attestation Server (IAS) issued verification report is used as the source attestation. It usually represents a unique Intel SGX TEE instance.

The proof data contains a verification report issued by SGX IAS.

`TEE-TRUST-CHAIN` indicates that an Enclave was attested by another Enclave creating an attestation or trust chain. In this case, the attesting Enclave provides the RSA and ECDSA/SECP256K1 public-private key pairs sets for the Enclaves it is attesting. Requestors are required to verify the trust chain to verify such a Enclave.

The proof data contains the signature generated by the attested Enclave. The signed message is a concatenation of verification key, subscription key, and all Enclave measurements of the attested Enclave.

Identification and attestation payload

```
{
    "workerTypeData": {
        "workerType":<hex string>,     //"TEE-" prefix
        "verificationKey": <hex string>,
        "encryptionKey": <hex string>,
        "enclaveMeasurements": [<one or more hex strings>],
        "proofDataType": <hex string>,
        "proofData": <hex string>,
    }
}
```

`workerType` is defined in section Worker Registry Smart Contract API.

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

`encryptionKey` is a hex string representing an RSA public key used to encrypt data sent to the Enclave. This field must be included in the `proofData`.

`enclaveMeasurements` is an array of hexadecimal strings representing one or more measurements. All values in this array must be included and attested in the `proofData`.

Requestors are expected to match Enclave measurements to required software off-chain unless the blockchain client contains a pre-compiled contract that enables on-chain matching. For details on the matching mechanism refer to https://software.intel.com/en-us/sgx.

`proofDataType` is one of the "TEE-" prefixed data types. For TEE type Intel SGX these are "TEE-SGX-IAS", or "TEE-TRUST-CHAIN".

`proofData` is proof data corresponding to the proofDataType. See Proof data Implementation Note.

***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 provisioned to multiple Enclaves. This specification supports "TEE-TRUST-CHAIN" proof data type to support Enclave pools, e.g.

- A master Enclave provides "SGX-IAS" proof data for its verification
- The master Enclave generates verification and subscription keys for use by the pool Enclaves
- The master Enclave securely provisions the keys to other Enclaves in the pool
- Pool Enclaves are registered in the TEE Enclave Registry. Their verification and encryption is signed by the master Enclave and the proof data type is set "TEE-TRUST-CHAIN". Since all Enclaves in the pool share the same keys, it is recommended to represent all of them by a single (master) Enclave record in the Worker Registry

To ***validate Enclave pool keys*** a Requestor should perform the following steps

1. Retrieve the Enclave pool record and check that its proof data type is "TEE-TRUST-CHAIN"
2. Retrieve master TEE Enclave registry info (specified in the Enclave pool proof data)
3. Check that the master proof data type is "SGX-IAS" and verify its proof data
4. Verify the signature generated by the master Enclave for the Enclave pool

The exact Enclave pool provisioning mechanism is outside the scope for this specification, but Requestors can inspect enclaveMeasurements of the master Enclave to learn what provisioning mechanism is used.

MPC workerTypeData

Identification and attestation payload

```
{
    "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 future.

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

ZK workerTypeData

Identification and attestation payload

```
{
    "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 Note" chapter for more details.

# 9. Implementation notes §

*This section is non-normative*.

## 9.1 Note 1: Receipts §

Work Order Receipts are created upon each successful or unsuccessful Work Order execution. They are created (and signed) by the Requestor 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.

## 9.2 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 off-chain registry.
- A Worker Service can support a single or a group of Workers. Workers may be registered individually, enabling Requestors to select a particular Worker for Work Order execution, or multiple Workers may be registered and managed collectively as a Worker Pool. When a Requestor selects a Worker Pool, the Worker Service chooses which Worker(s) in the pool will be used to execute a particular Work Order. Requestors should be able to lookup 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 an URI and a RSA and ECDSA/SECP256K1 public key set by its Worker Service on the main blockchain. An 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 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 public key that is used to create an Ethereum transaction that contains the updated Work Order Receipt generated by a Worker).
- The same "from" Ethereum public key (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 public key 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 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 Worker Service and 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.

### 9.3 Note 3: Proof Data §

This specification support 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 requestor 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.
- Custom Proofs must start with a tilde "~"

## A. Additional Information §

### A.1 Terminology §

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

A ***Worker*** is a computational resource for Work Order execution. A Worker may be identified by an Ethereum public key 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 HW 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 enforces 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 terms TEE and Enclave are used interchangeably.

A ***Worker Service*** is an implementation dependent middleware entity that acts as 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 Requestor for execution to a Worker. 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 a remote storage locations. Work Order inputs and outputs are normally sent encrypted.

The ***Direct Model*** is a Work Order execution model in which a Requestor 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-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