Baseline CCSM Specification Version 1.0 - Part 3

Project Specification Draft 01

17 April 2024

This stage:

https://docs.oasis-open.org/ethereum/baseline/baseline-core/v1.0/psd01/baseline-core-v1.0-psd01-part3.md (Authoritative)

https://docs.oasis-open.org/ethereum/baseline/baseline-core/v1.0/psd01/baseline-core-v1.0-psd01-part3.html

https://docs.oasis-open.org/ethereum/baseline/baseline-core/v1.0/psd01/baseline-core-v1.0-psd01-part3.pdf

Previous stage:

N/A

Latest stage:

https://docs.oasis-open.org/ethereum/baseline/baseline-core/v1.0/baseline-core-v1.0-part3.md (Authoritative)

https://docs.oasis-open.org/ethereum/baseline/baseline-core/v1.0/baseline-core-v1.0-part3.html

https://docs.oasis-open.org/ethereum/baseline/baseline-core/v1.0/baseline-core-v1.0-part3.pdf

Open Project:

Baseline, an initiative of Ethereum Community Projects

Project Chair:

Dr. Andreas Freund (a.freundhaskel@gmail.com)

Former Chair:

John Wolpert (john.wolpert@mesh.xyz), ConsenSys Mesh

Editors:

Dr. Andreas Freund (a.freundhaskel@gmail.com)

Kyle Thomas (kyle@provide.services), Provide

Yoav Bittan (yoav.bittan@mesh.xyz), ConsenSys Mesh

Keith Salzman (keith.salzman@mesh.xyz), ConsenSys Mesh

Chaals Nevile (chaals@entethalliance.org), EEA

Related work:

This specification is related to:

[baseline-core-v1.0] *Baseline Core Specification Version 1.0 - Part 1* Edited by Dr. Andreas Freund, Yoav Bittan, Keith Salzman, Chaal Neville, Anais Ofranc and Kyle Thomas. 17 April 2024. Project Specification Draft 01. https://docs.oasis-open.org/ethereum/baseline/baseline-core/v1.0/psd01/baseline-core-v1.0-psd01-part1.md Latest stage: https://docs.oasis-open.org/ethereum/baseline/baseline-core/v1.0/baseline-core-v1-part1.0.md

[baseline-core-v1.0] *Baseline Core Specification Version 1.0 - Part 2* Edited by Dr. Andreas Freund, Yoav Bittan, Keith Salzman, Chaal Neville, Anais Ofranc and Kyle Thomas. 17 April 2024. Project Specification Draft 01. https://docs.oasis-open.org/ethereum/baseline/baseline-api/v1.0/psd01/baseline-api-v1.0-psd01-part2.md Latest stage: https://docs.oasis-open.org/ethereum/baseline/baseline-api/v1.0/baseline-api-v1.0-part2.md

Abstract:

The document describes the minimal set of business and technical prerequisites, functional and non-functional requirements for a Consensus Controlled State Machine (CCSM) network that when utilized ensures that two or more systems of record can synchronize their system state over said CCSM securely and privately.

Status:

This document is a Project Specification Drat and is no longer under active development.

This was last revised or approved by Baseline, part of the Ethereum OASIS Open Project, on the above date. The level of approval is also listed above. Check the “Latest stage” location noted above for possible later revisions of this document.

Comments on this work can be provided by opening issues in the project repository or by sending email to the project’s public comment list baseline@lists.oasis-open-projects.org.

Keywords:

The keywords “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “NOT RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in [RFC2119] when, and only when, they appear in all capitals, as shown here.

Citation format:

When referencing this specification the following citation format should be used:

[baseline-ccsm-v1.0] *Baseline CCSM Requirements Version 1.0 - Part 3* Edited by Dr. Andreas Freund, Yoav Bittan, Keith Salzman, Chaal Neville, Anais Ofranc and Kyle Thomas. 17 April 2024. Project Specification Draft 01. https://docs.oasis-open.org/ethereum/baseline/baseline-core/v1.0/psd01/baseline-core-v1.0-psd01-part3.md . Latest stage: https://docs.oasis-open.org/ethereum/baseline/baseline-core/v1.0/baseline-core-v1.0-part3.md .

Notices

Table of Contents

---

1 Introduction

A Consensus Controlled State Machine (CCSM) is the foundational enabler of a Baseline Protocol Implementation (BPI) with no or limited trust assumptions.

The requirements that a CCSM must satisfy for it to be used in a BPI as defined in the Baseline Protocol Standard fall in the following categories:

1. Security
2. Privacy
3. Scalability
4. Interoperability
5. Network
6. Consensus
7. Virtual State Machine
8. Data Integrity & Transaction Completeness
9. Integration to External Applications

In the requirements below we will refer to “The CCSM” to mean a CCSM chosen by the participants to implement a BPI.

1.1 Glossary

Baseline Protocol:

The Baseline Protocol is a set of methods that enable two or more state machines to achieve and maintain data consistency, and workflow continuity by using a network as a common frame of reference.

Byzantine Fault Tolerant (BFT):

Given a network or system of n components, t of which are dishonest, and assuming only point-to-point channels between all the components, then whenever a component A tries to broadcast a value x such as a block of transactions, the other components are permitted to discuss with each other and verify the consistency of A’s broadcast, and eventually settle on a common value y. The system is then considered to resist Byzantine faults if a component A can broadcast a value x, and then:

- If A is honest, then all honest components agree on the value x.
- If A is dishonest, all honest components agree on the common value y.

“The Byzantine Generals Problem”, Leslie Lamport, Robert E. Shostak, Marshall Pease, ACM Transactions on Programming Languages and Systems, 1982

Consensus Algorithm:

A consensus algorithm achieves agreement among a number of processes (or agents) on a single data value based on candidate values generated by and communicated amongst the processes (agents). Since some of the processes (agents) may fail or be unreliable, a consensus algorithm must be fault tolerant or resilient.

Coulouris, George; Jean Dollimore; Tim Kindberg (2001), Distributed Systems: Concepts and Design (3rd Edition), Addison-Wesley, p. 452, ISBN 978-0201-61918-8

Consensus Controlled State Machine (CCSM):

A Consensus Controlled State Machine is a network of replicated, shared, and synchronized digital data spread across multiple sites connected by a peer-to-peer and utilizing a consensus algorithm. There is no central administrator or centralized data storage.

Interoperability:

The ability of a Party operating Workflows on a baseline-compliant implementation A to instantiate and operate one or more Workflows with one or more Party on a baseline-compliant implementation B without the Party on either implementation A or B having to know anything of the other Party’s implementation.

Liveness:

In concurrent computing, liveness refers to a set of properties of concurrent systems, that require a system to make progress, despite its concurrently executing components (“processes”) may have to “take turns” in critical sections, parts of the program that cannot be simultaneously run by multiple processes. Liveness guarantees are important properties in operating systems and distributed systems.

Alpern B, Schneider FB (1985) Defining liveness. Inf Proc Lett 21:181-185

Party:

A set of Parties participating in the execution of one or more given Workflows. A Workgroup is set up and managed by one Party that invites other Parties to join as workgroup members.

Proof of Correctness:

A Proof of Correctness is a mathematical proof that a computer program or a part thereof will, when executed, yield correct results, i.e. results fulfilling specific requirements. Before proving a program correct, the theorem to be proved must, of course, be formulated. The hypothesis of such a correctness theorem is typically a condition that the relevant program variables must satisfy immediately before the program is executed. This condition is called the precondition. The thesis of the correctness theorem is typically a condition that the relevant program variables must satisfy immediately after execution of the program. This latter condition is called the postcondition. The thesis of a correctness theorem may be a statement that the final values of the program variables are a particular function of their initial values.

“Encyclopedia of Software Engineering”,
Print ISBN: 9780471377375| Online ISBN: 9780471028956| DOI: 10.1002/0471028959,
(2002), John Wiley & Sons, Inc.

Verifiably Secure:

Verifiable computing that can be described as verifiably secure enables a computer to offload the computation of some function to other perhaps untrusted clients, while maintaining verifiable, and thus secure, results. The other clients evaluate the function and return the result with a proof that the computation of the function was carried out correctly. The proof is not absolute but is dependent on the validity of the security assumptions used in the proof. For example, a blockchain consensus algorithm where the proof of computation is the nonce of a block. Someone inspecting the block can assume with virtual certainty that the results are correct because the number of computational nodes that agreed on the outcome of the same computation is defined as sufficient for the consensus outcome to be secure in the consensus algorithm's mathematical proof of security.

Gennaro, Rosario; Gentry, Craig; Parno, Bryan (31 August 2010). Non-Interactive Verifiable Computing: Outsourcing Computation to Untrusted Workers. CRYPTO 2010. doi:10.1007/978-3-642-14623-7_25

1.2 Typographical Conventions

1.2.1 Requirement Ids

A requirement is uniquely identified by a unique ID composed of its requirement level followed by a requirement number, as per convention **[RequirementLevelRequirementNumber]**.
There are four requirement levels that are coded in requirement ids as per below convention:

[R] - The requirement level for requirements which IDs start with the letter*R* is to be interpreted as **MUST** as described in RFC2119.

[D] - The requirement level for requirements which IDs start with the letter*D* is to be interpreted as **SHOULD** as described in RFC2119.

Note that requirements are uniquely numbered in ascending order within each requirement level.

Example: It should be read that [R1] is an absolute requirement of the specification whereas [D1] is a recommendation.

---

2. Security

CCSM security is one of the most, if not the most important characteristic of a CCSM. Therefore, great care has to be taken in its definition, starting with the utilized cryptographic algorithms and their implementations.

[R1]

A CCSM utilized in a BPI MUST support cryptographic algorithms based on commonly used and security-audited libraries.

The usage of cryptographic libraries that successfully passed the National Institute of Standards and Technology (NIST) Cryptographic Module Verification Program (CMVP) is recommended.

[R1] testability: Any security-audited cryptographic library will have test-vectors, and is, therefore, testable.

[R2]

If the CCSM utilized in a BPI utilizes a Peer-to-Peer (P2P) message protocol, the protocol MUST support end-to-end encryption.

[R2] testability: libp2p as a very commonly used P2P message protocol supports end-to-end encryption, seehere for the NOISE specification and relevant tests here. This shows that the requirement is testable.

[R3]

The CCSM utilized in a BPI MUST support CCSM Node Key Management incl. backup and recovery that adheres to established industry security standards such as the US Federal Information Processing Standard [FIPS] or ISO 27001 [ISO27001].

[R3] testability: Given that there exist certification programs for [FIPS] or ISO 27001 [ISO27001], see for example the NIST program, the requirement is testable.

[R4]

The CCSM utilized in a BPI SHOULD support programmatic economic security assurances.

Note, economic security assurances such as those used in Proof-of-Stake consensus algorithms are designed to provide additional security assurances beyond those of cryptography in distributed systems. The security assurances are based on a system of economic incentives and disincentives for distributed system participants with the expressed goal that honest behavior of distributed system participants which enhances system security is in their economic self-interest. This is akin to determining if a cryptographic algorithm is secure or not, and what the level of security of said algorithm is, the security of a system of economic incentives and disincentives must be proven through a game theoretic security analysis.

[R4] testability: Programmatic economic security assurances are typically implemented within the CCSM's consensus protocol. Examples are Proof-of-Work for Bitcoin, and Proof-of-Stake for Ethereum, as common CCSMs. Ethereum's Proof-of-Stake protocol is specified here and tests of a security audited reference implementation can be found here. Therefore, the requirement is testable.

[D1]

The CCSM utilized in a BPI SHOULD be compatible with CCSM protocol execution in Trusted Execution Environments (TEE).

Note, a TEE is a secure area of a main processor. It guarantees code and data loaded inside to be protected with respect to confidentiality and integrity. A TEE as an isolated execution environment provides security features such as isolated execution, integrity of applications executing within the TEE, along with confidentiality of their assets.

[D1] testability: An example of a CCSM utilizing a TEE is Hyperledger Sawtooth with itstest suite. Therefore, the requirement is testable.

[R5]

The CCSM utilized in a BPI MUST provide high network attack resistance and detection capabilities at the protocol level per ISO/IEC 27033ISO-IEC-27033.

[R5] testability: Similar to ISO 27001, there are numerous ISO/IEC 27033 certification programs. Therefore, the requirement is testable.

Network attacks typically take the form of Distributed Denial of Service (DDOS) attacks, attacks from groups of malicious CCSM nodes performing CCSM reorganizations, front running of transactions through transaction injections, and censoring of transactions. This includes game theoretic attacks such as discouragement, extortion, value-extraction, or random oracle attacks.

[R6]

The CCSM utilized in a BPI MUST support a secure consensus algorithm as explained in section8 Virtual State Machine.

Note that secure in this context refers to the security of a consensus algorithm against attacks directed towards its three main characteristics – consistency, availability, and fault tolerance. Therefore, a consensus algorithm is considered secure for a given set of operating assumptions:

- if all nodes produce the same valid output, according to the protocol rules, for the same message broadcast to the network (consistency/safety),
- if all non-faulty participating nodes produce an output indicating the termination, and subsequent restart, of the protocol upon reaching consensus (availability/liveness), and
- if the network exhibits the capability to perform as intended if network nodes fail, either unintentionally or intentionally (fault tolerance).

[R6] testability: Consensus security is proven mathematically, and a reference implementation with tests is then security audited. An example is Ethereum's Proof-of-Stake protocol specified here and tests of a security audited reference implementation can be foundhere. Therefore, the requirement is testable.

[R7]

A CCSM utilized in a BPI MUST have one or more secure-by-construction or Verifiably Secure execution frameworks.

See the glossary for a definition of Verifiably Secure and more details about CCSM supported execution frameworks in section9 Data Integrity and Transaction Completeness.

[R7] testability: CCSM execution frameworks can be verifiably secure if cryptographic proofs-of-correct execution of the execution stack trace can be independently verified to be correct. An example of such an execution framework is the Ethereum Virtual Machine (EVM) with its specification. Proofs-of-correct execution of the EVMs stack trace can be implemented and tested in the context of, for example, a zero-knowledge Ethereum Virtual Machine (zkEVM) that produces such proofs in zero-knowledge with tests showing conformance to the EVM specification. Therefore, the requirement is testable.

3. Privacy

CCSMs range in the level of privacy they support. One approach ensures that the contents of a CCSM transaction or storage are meaningless to parties not participating in an interaction. Another more stringent approach is to use a CCSM that precludes the accessibility of such information to non-participating parties. This standard sets the minimum requirement to the first approach, but the parties can agree to require that the BPI supports the second approach.

[R8]

The CCSM utilized in a BPI MUST support privacy preservation such that the contents of a CCSM transaction or CCSM storage does not carry meaning to parties not participating in a CCSM based interaction.

[R8] testability: The ability to reason about privacy preservation tools such as hashes, Merkle proofs or other, more sophisticated, zero-knowledge proofs requires both a deterministic execution framework and the ability within this framework to reason about privacy-preserving cryptographic primitives. A widely utilized execution framework meeting both criteria is the EVM with its deterministic execution framework and the ability to reason about cryptographic primitives through the usage of EVM precompiles to perform elliptic curve operations. A comprehensive set of tests of the EVM specification itself including precompiles can be found here. Therefore, the requirement is testable.

[R9]

The CCSM utilized in a BPI MUST support privacy preservation of transactions and their execution.

[R9] testability: Reasoning about cryptographically shielded transactions and their execution against a given, yet hidden, state can be performed by, for example, zkEVM proofs as discussed in the [R7] testability statement verified on a CCSM using an EVM execution framework as discussed in the[R8] testability statement. As a consequence of both of those testability statements taken together, the requirement [R9] itself is testable.

[R10]

The CCSM utilized in a BPI MUST support segregation between public and private state/data.

[R10] testability: The testability of [R10] is consequence of the testability statements [R7] - [R9] since a private state can be represented as zero-knowledge proofs (private state data) embedded in the public state of a CCSM.

[D2]

The CCSM utilized in a BPI SHOULD support a privacy-preserving P2P message protocol.

Privacy-preserving in this context means that at least the content of a message, and ideally, also the sender and recipient, is opaque to all participants of the P2P network except sender and recipient.

[D2] testability: Testability of this requirement is a direct consequence of the testability of[R2].

[D3]

The CCSM utilized in a BPI SHOULD support Zero-Knowledge Proof (ZKP) verification (if not generation) at the protocol level.

Zero-Knowledge Proofs (ZKPs) (see reference [ZKP]) are powerful cryptographic methods by which one party (the prover) can prove to another party (the verifier) that they know a value x – the password to an online bank account –, without conveying any information apart from the fact that they know the value x – the password. The essence of zero-knowledge proofs is that it is trivial to prove that one possesses knowledge of certain information by simply revealing it; the challenge is to prove such possession without revealing the information itself or any additional information. When combined with CCSMs, ZKPs allow participants to conduct business and exchange assets in the open without revealing anything about the business itself while any outside party can verify that the way business was conducted was in accordance with all applicable business and legal rules for a commercial transaction.

[D3] testability: This requirement is a stronger requirement than [R9]. The testability statement of [R9] references zkps. Therefore, [D2] is testable.

4. Scalability

To support the required commercial transaction volume between Baseline Protocol counterparties, the CCSM utilized by a BPI should be chosen with these transaction volumes in mind. Especially, since in a public CCSM setting there will be, potentially, a significant volume of transactions competing for scarce CCSM processing and storage resources.

Since forecasting future transaction volumes is difficult and could rapidly change based on adoption, the considered CCSMs should have some form of throughput future-proofing built in. Examples of such techniques include state channels, sidechains, rollup frameworks, state sharding, multiple execution frameworks and parallel process transaction support. This is not mandated in this standard and is considered a question of implementation to be addressed in an agreement between BPI participants.

5. Interoperability

[D4]

The CCSM utilized in a BPI SHOULD support secure data sources.

This requirement means that a CCSM has a mechanism to securely connect its state through, for example, a smart contract with a data source which has certain security assurances in such a way that: a) the security of the data source is not compromised by the CCSM and b) the security assurances of the CCSM are not compromised by the secure data source.

[D4] testability: An example of such a secure data source is a zkEVM operating on a CCSM, and since a zkEVM is testable as shown in the [R7] testability statement, this requirement is testable.

[D5]

When transactions connect one CCSM with another CCSM for the purpose of interoperating assets or data across BPIs, and the CCSMs use the same CCSM Protocol, the CCSM utilized in a BPI SHOULD support asset and data locking techniques to prevent double-spend/usage of assets.

An example of such techniques is a two-phase lock relay bridge. Two-phase locking (2PL) is a concurrency control method that guarantees serializability. The protocol utilizes locks, applied by a CCSM transaction to data/assets, which may block other CCSM transactions from accessing the same data/assets during the lifecycle of a transaction impacting said data/asset(s). This protocol requires support for CCSM transaction receipts signaling CCSM transaction lifecycle completeness. This approach requires a relay server (network) between the two CCSMs which interacts with the locking/unlocking smart contracts on each of the CCSMs. Since both CCSMs operate the same CCSM protocol the relay server can be a node on both networks which does not introduce further security assumptions.

[D5] testability: An example of a protocol with a test suite satisfying the requirement is the GPACT protocol for EVM-compatible CCSMs. Therefore, the requirement is testable.

[D6]

When transactions connect one CCSM with another CCSM for the purpose of interoperating assets or data across BPIs, and the CCSMs use different CCSM Protocols, the CCSM utilized in a BPI SHOULD support asset and data locking techniques to prevent double-spend/usage of assets.

An example of such techniques are Atomic Swap protocols. An atomic swap is a CCSM smart contract technology that enables the exchange of one CCSM asset for another without using centralized intermediaries, such as exchanges.

[D6] testability: Since CCSMs using different protocols can still use the same execution framework such as Avalanche, or Hedera using an EVM-compatible execution client, the GPACT protocol protocol can be utilized for CCSM interoperability. Therefore, and based on the [D5] testability statement, this requirement is testable.

6. Network

Network in this context refers to the nodes of a CCSM that form the CCSM network. A CCSM node has several components that impact the network namely its Peer-to-Peer (P2P) message protocol and its consensus algorithm.

It is important that Peer-to-Peer (P2P) message protocols are used that do not require network nodes which act as message distribution hubs, e.g., leader nodes because network attacks on leader nodes can either cause unwanted network partitions or even network collapse.

[R11]

A CCSM utilized in a BPI MUST support a P2P message protocol that does not require network leader nodes.

[R11] testability: libp2p as a very commonly used P2P message protocol with a test suite here is a leaderless protocol using peer-gossiping. This shows that the requirement is testable.

The network requirements on the consensus algorithms are even more stringent than on the P2P protocol. Additional requirements on the consensus algorithm of the CCSM are discussed in the next section, section 7. Consensus.

[R12]

The CCSM utilized in a BPI MUST be Byzantine Fault Tolerant (BFT) [BFT].

[R12] testability: Ethereum's Proof-of-Stake consensus protocol satisfies the BFT requirements as specifiedhere and tests of a security audited reference implementation can be found here. Therefore, the requirement is testable.

[R13]

The CCSM utilized in a BPI MUST be able to operate under Weak Synchrony.

Weak synchrony in this context means:

1. That all messages will eventually reach their intended recipients
2. That after a certain, yet unknown time, the network will become synchronous again

[R13] testability: For a CCSM to operate under Weak Synchrony, both its messaging implementation and its consensus protocol implementation must satisfy the condition. Both libp2p and Ethereum's Proof-of-Stake consensus protocol implementations operate under weak synchrony per their specifications and as testable by their test suites. This means combining the [R11] and [R12] testability statements shows that this requirement is testable.

7. Consensus

The consensus algorithm is the most important component of a CCSM as it ensures the consistency of the network at any given time. Therefore, the requirements on the consensus algorithms are very stringent.

[R14]

The CCSM utilized in a BPI MUST be able to support more than one BFT consensus algorithm.

This is also known as plugable consensus. Note, that deterministic BFT consensus algorithms lead to strong consistency, and, thus, immediate finality. Probabilistic BFT consensus algorithms lead to eventual consistency, and, thus, eventual finality.

[R14] testability: Any CCSM where the execution client is separate from the consensus client can support multiple BFT consensus algorithms. A well known example is the Hyperledger Besu client for Ethereum with a well-definedtest suite. Therefore, the requirement is testable.

[R15]

Consensus algorithms employed in the CCSM utilized in a BPI MUST have a mathematical proof of security.

A mathematical proof of security is a collection of mathematically provable theorems that make security statements about the three characteristics of a consensus algorithm – consistency/safety, availability/liveness and fault tolerance and is based on specific operating assumptions of the protocol.

[R15] testability: All well-known consensus algorithms have peer-reviewed papers with a mathematical security proof. A peer-reviewed overview of mathematical security proofs of the most popular consensus algorithms can be found here

[D7]

Consensus algorithms employed in the CCSM utilized in a BPI SHOULD include economic security assurances with game theoretic security proofs.

[D7] testability: An example of such a consensus algorithm is Ethereum's Proof-of-Stake protocol specifiedhere including a game theoretic security analysis and tests of a security audited reference implementation can be found here. Therefore, the requirement is testable.

[D8]

Consensus algorithms employed in the CCSM utilized in a BPI SHOULD require not more than order N messages to reach consensus where N is the number of nodes in the network.

Note that the larger the number of nodes, the higher the level of security. Also, note that performance for certain consensus algorithms degrades quickly as the number of nodes increases because of the number of messages required to exchange between them to achieve consensus can grow very quickly. Therefore, algorithms that scale in the number of nodes without significant performance degradation are preferred. Also, note that network performance such as poor network latency can lead to severe issues such as consensus failure if an algorithm requires the exchange of large numbers of messages to reach consensus.

[D8] testability: An example of such a consensus algorithm requiring only O(N) messages is Ethereum's Proof-of-Stake protocol specifiedhere because of its eventual finality feature. Tests of a security audited reference implementation can be found here. Therefore, the requirement is testable.

8. Virtual State Machine

CCSMs most often utilize a virtual state machine (VSM) for CCSM computations of CCSM state transitions; a digital computer running on a physical computer. A VSM requires an architecture and execution rules which together define the Execution Framework.

[R16]

The Execution Framework of the CCSM utilized in a BPI MUST be deterministic.

All CCSM nodes need to arrive at the same result based on the same input and execution instructions, in other words deterministic outcomes. This is only guaranteed if the Execution Framework either does not allow instructions to be executed in parallel, but only strictly sequential, or if the Execution Framework has methods in place that allow the identification and prevention of transactions that would cause CCSM state conflicts, if processed in parallel.

For example, the Buyer, also known as Requester, proposes a commercial state change of the MSA through Order A which is created at time t, and the Seller, also known as the Provider, has just agreed to a suggested discount rate change in the MSA submitted by the Buyer at time t-1 but not yet confirmed by CCSM consensus. This means that if the transaction of the Order A is processed in parallel to the discount change the wrong discount might be applied to Order A depending which transaction is executed first.

[R16] testability: An example of such a deterministic execution framework is the Ethereum Virtual Machine (EVM) with its specification. Proofs-of-correct execution of the EVMs stack trace can be implemented and tested in the context of, for example, a zero-knowledge Ethereum Virtual Machine (zkEVM) that produces such proofs in zero-knowledge with tests showing conformance to the EVM specification. Therefore, the requirement is testable.

[R17]

The Execution Framework of the CCSM utilized in a BPI MUST ensure that state transition computations are either completed or aborted in finite time.

What is deemed to be a suitable finite time is determined by the commercially allowable duration of a commercial transaction. This requirement means that there cannot be infinite computational loops in a distributed computational system with consensus, as this would not allow the CCSM network to reach consensus anymore and bring the CCSM network itself to a halt. Note also, that when a CCSM node is offline, the virtual state machine's Execution Framework does not perform computations; when a CCSM node comes back online, and synchronizes with the state of the CCSM network, it only validates the last available state - either a global state or specific to that node.

[R17] testability: An example of such a non-Turing-complete execution framework is the Ethereum Virtual Machine (EVM) with its specification. Proofs-of-correct execution of the EVMs stack trace with guaranteed completion for every transaction can be implemented and tested in the context of, for example, a zero-knowledge Ethereum Virtual Machine (zkEVM) that produces such proofs in zero-knowledge with tests showing conformance to the EVM specification. Therefore, the requirement is testable.

[R18]

The Execution Framework of the CCSM utilized in a BPI MUST support widely used cryptographic operations natively, e.g., hashing, digital signatures, or zero-knowledge proof verification.

[R18] testability: An example of such an Execution Framework is the EVM as implemented in the Hyperledger Besu client for Ethereum with a well-defined test suite. Therefore, the requirement is testable.

[D9]

The Execution Framework of the CCSM utilized in a BPI SHOULD have a mathematical proof of correctness and security.

[D9] testability: An example of such an execution framework is the Ethereum Virtual Machine (EVM) with its specification with an associated security-audited implementation and test suite in the Hyperledger Besu client. Therefore, the requirement is testable.

[R19]

The Execution Framework of the CCSM utilized in a BPI MUST be Verifiably Secure.

[R19] testability: CCSM execution frameworks can be verifiably secure if cryptographic proofs-of-correct execution of the execution stack trace can be independently verified to be correct. An example of such an execution framework is the Ethereum Virtual Machine (EVM) with its specification. Proofs-of-correct execution of the EVMs stack trace can be implemented and tested in the context of, for example, a zero-knowledge Ethereum Virtual Machine (zkEVM) that produces such proofs in zero-knowledge with tests showing conformance to the EVM specification. Therefore, the requirement is testable.

9. Data Integrity and Transaction Completeness

Data integrity over time, in other words the inability to alter data once it has been committed to the state of the CCSM, is one of the key features of typical CCSMs.

[R20]

If the CCSM utilized in a BPI is strongly consistent (as defined in section 7. Consensus), data committed to the state of the CCSM MUST NOT be alterable after the CCSM state has been finalized (as defined in section 7).

[R20] testability: An example of such a CCSM is the Hyperledger Besu client for Ethereum using the supported QBFT consensus algorithm and with a well-defined test suite. Therefore, the requirement is testable.

[R21]

If the CCSM utilized in a BPI is eventually consistent (as defined in section 7. Consensus), data committed to the state of the CCSM MUST NOT be alterable after the CCSM state has been finalized (as defined in section 7).

Besides data integrity, the notion of censorship-resistance, or the inability of anyone participant in a CCSM to stop any other participant's transaction to be eventually included in the CCSM state, is another key feature of typical CCSMs. It conveys the concept of a network without a central authority that can stop things from happening at will. This can be formalized as follows.

[R21] testability: An example of such a CCSM is Ethereum with its Proof-of-Stake protocol that guarantees eventual finality and is specified here and tests of a security audited reference implementation can be found here. Therefore, the requirement is testable.

[R22]

The CCSM utilized in a BPI MUST guarantee that a transaction compliant with the CCSM protocol rules is eventually included in the state of the CCSM, if the security assumptions of the utilized consensus protocol remain valid during transaction processing (see section 6. Network for details on the security assumptions of consensus algorithms).

The reason why the reference to the consensus algorithm is important is as follows: To guarantee processing of a transaction, one needs only one honest CCSM node in the network. However, this is not sufficient to guarantee consensus. Therefore, and to include a submitted transaction in the CCSM state, there needs to be an honest majority of CCSM nodes to reach consensus on the submitted transaction.

[R22] testability: An example of such a CCSM is Ethereum with its Proof-of-Stake protocol that guarantees eventual finality and is specified here and tests of a security audited reference implementation can be found here. Therefore, the requirement is testable.

10. Integration Capabilities with External Systems

Depending on the CCSM employed in the implementation of a BPI, the security requirements around integration below need to be fulfilled either by the CCSM itself used for the implementation or, alternatively by the CCSM Abstraction Layer. Note that these requirements are distinct from the security requirements for the Baseline Protocol Standard APIs or any custom APIs for that matter because the BPI Processing Layer which is invoking services exposed by the CCSM Abstraction Layer or the CCSM itself should not be assumed to be a trusted service without authentication. This is because the standard does not define the operating model of a CCSM or a BPI or any of the BPI layers, and, therefore, must necessarily prescribe requirements for a 100% adversarial environment.

[R23]

The CCSM utilized in a BPI or the CCSM Abstraction Layer interacting with said CCSM MUST be compatible with widely used external authentication services.

Non-normative examples of such authentication technologies are OAUTH [OAuth-2.0] , SAML [SAML] , OIDC [OIDC], AD/LDAP [ActiveDirectory].

[R23] testability: An example of such a CCSM is the Hyperledger Besu client for Ethereum with a well-defined test suite. Therefore, the requirement is testable.

[R24]

The CCSM utilized in a BPI or the CCSM Abstraction Layer interacting with said CCSM MUST support roles & access management.

[R24] testability: An example of such a CCSM is the Hyperledger Besu client for Ethereum with a well-defined test suite. Therefore, the requirement is testable.

[R25]

The CCSM utilized in a BPI or the CCSM Abstraction Layer interacting with said CCSM MUST support policy management.

[R25] testability: An example of such a CCSM is the Hyperledger Besu client for Ethereum with a well-defined test suite. Therefore, the requirement is testable.

[R26]

The CCSM utilized in a BPI or the CCSM Abstraction Layer interacting with said CCSM MUST support Single-Sign-On (SSO).

See [SSO] also for the recommendations of the National Institute of Standards and Technology (NIST Guide to Secure Web Services).

[R26] testability: SSO can be combined with Oauth2 using for example OpenID Connect certified implementations which have certification test suites. Since Oauth2 is supported by the Hyperledger Besu client, the requirement is testable.

[R27]

The CCSM utilized in a BPI or the CCSM Abstraction Layer interacting with said CCSM MUST support Multi-Factor authentication (MFA).

See the link here for the NIST definition adopted in this document MFA.

[R27] testability: SSO can be combined with MFA suing for example the open-source Authelia framework with its test suit. Therefore, [R27] testability is a testable extension of [R26] testability.

[R28]

The CCSM utilized in a BPI or the CCSM Abstraction Layer interacting with said CCSM MUST support Hardware Security Modules (HSM).

This document adopts the NIST definition and for further information, refer to [HSM].

[R28] testability: HSMs can be readily integrated with OIDC using for example Authelia together with a commercial HSM. Therefore, [R28] testability is a testable extension of [R27] testability.

11 Conformance

This section describes the conformance clauses and tests required to achieve an implementation that is provably conformant with the requirements in this document.

11.1 Conformance Targets

This document does not yet define a standardized set of test-fixtures with test inputs for all MUST, SHOULD, and MAY requirements with conditional MUST or SHOULD requirements.

A standardized set of test-fixtures with test inputs for all MUST, SHOULD, and MAY requirements with conditional MUST or SHOULD requirements is intended to be published with the next version of the standard.

11.2 Conformance Levels

This section specifies the conformance levels of this standard. The conformance levels aim to enable implementers several levels of conformance to establish competitive differentiation.

This document defines the conformance levels of a CCSM as follows:

- **Level 1:** All MUST requirements are fulfilled by a specific implementation as proven by a test report that proves in an easily understandable manner the implementation's conformance with each requirement based on implementation-specific test-fixtures with implementation-specific test-fixture inputs.
- **Level 2:** All MUST and SHOULD requirements are fulfilled by a specific implementation as proven by a test report that proves in an easily understandable manner the implementation's conformance with each requirement based on implementation-specific test-fixtures with implementation-

specific test-fixture inputs.
- **Level 3:** All MUST, SHOULD, and MAY requirements with conditional MUST or SHOULD requirements are fulfilled by a specific implementation as proven by a test report that proves in an easily understandable manner the implementation's conformance with each requirement based on implementation-specific test-fixtures with implementation-specific test-fixture inputs.
- **Level 4:** All MUST requirements are fulfilled by a specific implementation as proven by the test report defined in this document that proves the implementation's conformance with each requirement based on test-fixtures with test-fixture inputs as defined in this document. **This conformance level cannot yet be achieved since there is not yet a defined set of standardized test-fixtures and test-inputs**.
- **Level 5:** All MUST and SHOULD requirements are fulfilled by a specific implementation as proven by the test report defined in this document that proves the implementation's conformance with each requirement based on test-fixtures with test-fixture inputs as defined in this document. **This conformance level cannot yet be achieved since there is not yet a defined set of standardized test-fixtures and test-inputs**.
- **Level 6:** All MUST, SHOULD, and MAY requirements with conditional MUST or SHOULD requirements are fulfilled by a specific implementation as proven by the test report defined in this document that proves the implementation's conformance with each requirement based on test-fixtures with test-fixture inputs as defined in this document. **This conformance level cannot yet be achieved since there is not yet a defined set of standardized test-fixtures and test-inputs**.

[D10]

A claim that a canonical token list implementation conforms to this specification SHOULD describe a testing procedure carried out for each requirement to which conformance is claimed, that justifies the claim with respect to that requirement.

[D10] testability: Since each of the non-conformance-target requirements in this documents is testable, so must be the totality of the requirements in this document. Therefore, conformance tests for all requirements can exist, and can be described as required in [D10].

[R29]

A claim that a canonical token list implementation conforms to this specification at**Level 2** or higher MUST describe the testing procedure carried out for each requirement at **Level 2** or higher, that justifies the claim to that requirement.

[R29] testability: Since each of the non-conformance-target requirements in this documents is testable, so must be the totality of the requirements in this document. Therefore, conformance tests for all requirements can exist, be described, be built and implemented and results can be recorded as required in [R29].

Appendix A - References

This appendix contains the normative and non-normative references that are used in this document.

While any hyperlinks included in this appendix were valid at the time of publication, OASIS cannot guarantee their long-term validity.

A.1 Normative References

The following documents are referenced in such a way that some or all of their content constitute requirements of this document.

[RFC2119]

S. Bradner, Key words for use in RFCs to Indicate Requirement Levels,http://www.ietf.org/rfc/rfc2119.txt, IETF RFC 2119, March 1997.

[ISO-IEC-27033]

ISO/IEC 27033: Information technology — Security techniques — Network security - Parts 1 through 6 published by ISO.

A.2 Non-Normative References

[FIPS]

FIPS, https://www.nist.gov/itl/current-fips.

[ISO27001]

ISO/IEC 27001:2013, https://www.iso.org/standard/54534.html.

[How-to-Explain-Zero-Knowledge-Protocols-to-Your-Children]

Quisquater, Jean-Jacques; Guillou, Louis C.; Berson, Thomas A. (1990). "How to Explain Zero-Knowledge Protocols to Your Children". Advances in Cryptology – CRYPTO '89: Proceedings. Lecture Notes in Computer Science. 435. pp. 628–631. doi:10.1007/0-387-34805-0_60. ISBN 978-0-387-97317-3.

[The-Byzantine-Generals-Problem]

"The Byzantine Generals Problem", Leslie Lamport, Robert E. Shostak, Marshall Pease, ACM Transactions on Programming Languages and Systems, 1982.

[OAuth-2.0]

Aaron Parecki, (2020), "OAuth 2.0 Simplified", ISBN-13: 978-1387751518.

[SAML]

J. Hughes et al. Profiles for the OASIS Security Assertion Markup Language (SAML) V2.0. OASIS Standard, March 2005. Document identifier: saml-profiles-2.0-os.

[OIDC]

OpenID Connect Federation 1.0, https://openid.net/developers/specs/ (2019).

[ActiveDirectory]

“Directory System Agent”. MSDN Library. Microsoft. (2018).

[SSO]

Recommendations of the National Institute of Standards and Technology (NIST Guide to Secure Web Services). NIST SP 800-95, https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-95.pdf.

[HSM]

Ramakrishnan, Vignesh; Venugopal, Prasanth; Mukherjee, Tuhin (2015). Proceedings of the International Conference on Information Engineering, Management and Security 2015: ICIEMS 2015. Association of Scientists, Developers and Faculties (ASDF). p. 9. ISBN 9788192974279.

A.3 Internationalization and Localization Reference

The standard encourages implementers to follow the W3C “Strings on the Web: Language and Direction Metadata” best practices guide for identifying language and base direction for strings used on the Web wherever appropriate.

Appendix B - Acknowledgments

The following individuals have participated in the creation of this specification and are gratefully acknowledged.

Participants:

Dr. Andreas Freund

Anais Ofranc, Consianimis

Gage Mondok, Chainlink

Kyle Thomas, Provide

Daven Jones, Provide

Mehran Shakeri, SAP

Alessandro Gasch, SAP

John Wolpert, ConsenSys

Sam Stokes, ConsenSys

Nick Kritikos, ConsenSys

Yoav Bittan, ConsenSys

Appendix C - Revision History

Revisions made since the initial stage of this numbered Version of this document have been tracked on Github .

Appendix D - Notices