Key Event Receipt Infrastructure (KERI)

Design and Build

Samuel M. Smith Ph.D.[1]

v1.62 2020/02/11

v1.60 2019/07/03

Abstract—A decentralized key management infrastructure (DKMI) that uses the design principle of minimally sufficient means is presented. The primary key management operation is key rotation via a novel key pre-rotation scheme. Two trust modes are presented, the online or pair-wise mode and the offline or any-wise mode. The offline mode depends on witnessed key event receipt logs (KERL) as an additional trust basis for validating events. This gives rise to the acronym KERI for key event receipt infrastructure. The KERI approach may be much more performant and scalable than more complex approaches that depend on a total ordered distributed consensus ledger. KERI may be augmented with distributed consensus ledgers but does not require them. The KERI approach allows more granular architecture in a DKMI. Moreover, because KERI is event streamed it enables DKMI that operates in-stride with data events streaming applications such as web 3.0, IoT, and others where performance and scalability are more important. KERI is designed to support DIDs but its core services are identifier independent (this includes DID method independence). This makes KERI a candidate for a universal portable DKMI.

Index Terms—Decentralized, Key, Management, Infrastructure, Key, Event, Receipts, Pre-rotation, Rotation, Event, Streaming, DKMI, KERI, KERL, KEL.

1 Introduction

Herein we define an *identity* for an *entity* simply as an *identifier* and *attributes* that may be used to describe the *entity*. Entities are not limited to natural persons but may include groups, organizations, software agents, things, and even data items. With a centralized identity system one entity controls all the identifiers. With a decentralized identity system, in contrast, disparate entities each control some of the identifiers but in an interoperable way. To restate, each entity may have a set of identifiers that it controls but are still recognized by the other entities. Each entity may control or be sovereign over a set of identifiers. In the case where an identifier refers to its controlling entity then that entity is self-sovereign over its own identifier and hence the identity associated with that identifier. This property of decentralized identity systems has given rise to the term self-sovereign identity (SSI) [1; 20; 21; 28].

The typical approach to establishing or proving control over an identifier is to use a universally unique cryptographic digital signature based a key pair comprised of a public key and a private key. The identifier includes the public key or a unique fingerprint of the public key. The private key is used to digitally sign attestations that may be cryptographically verified using the public key. The holder of the private key is in *control* of the identifier because only that holder can sign attestations that may be cryptographically verified with the public key bound to that identifier. This makes the identifier *self-certifying* [11; 15–17].

1. https://arxiv.org/abs/1907.02143

1.1 Source of Truth

The fundamental assumption is that the (public, private) key pair is universally unique and may be established using a random seed of sufficient size such that the likelihood of duplication (collision) is negligible. A typical method for generating the random seed is to use a cryptographic strength pseudo-random number generator with sufficient entropy. Currently 128 bits of entropy is considered sufficient. Other self-certifying identifier systems may use different means to generate and prove control over a universally unique identifier but this work is limited to systems where control is proven via cryptographic (public, private) key pair based digital signatures. In this sense there is one primary source of truth for the identifier and hence any key management operations applied to the associated (public, private) key pair. This source of truth is the set of signed attestations made with the private key and verifiable with the public key.

Although this work may be generally applied to any decentralized identity system that uses self-certifying identifiers, its primary focus is on decentralized identity systems that are interoperable with the emerging DID (Decentralized ID) standard [7; 23; 26; 27].

1.2 Minimally Sufficient Means

For decentralized identity systems based on self-certifying identifiers, management of the associated private keys is essential. Because the controlling entity holds their own private key(s) the primary burden of management falls on that entity or its assigns. The security of the identity is a function of the security of the management infrastructure. As mentioned above, unlike a centralized or administrative identity system where a central administrative entity controls all the identifiers, a decentralized identity system may have a multitude of controlling entities each controlling one or more identifiers. Some of these entities may not have the resources or the expertise to design, build, and maintain secure key management infrastructure. Consequently there is a need for open interoperable decentralized key management infrastructure (DKMI). Moreover, some applications of decentralized identity may benefit from DKMI that is scalable and performant. Example applications include data streaming, supply chain, IoT (internet of things), and other applications where data provenance among multiple controlling entities is important and data processing is demanding. One design approach to composing scalable and performant infrastructure is to find minimally sufficient means for each of the key management tasks. This is a primary motivation for this work, that is, to identify the minimally sufficient means for essential key management tasks. This does not imply that other means might not be beneficial or best for a given application but that by first understanding minimally sufficient means an implementor might have at hand more design options that might be customized to better fit a broader class of applications.

1.3 Distributed Ledgers

The primary alternative for decentralized key management infrastructure is a distributed ledger based on a distributed consensus algorithm that provides an additional source of truth for key management operations. There are many types of distributed consensus algorithms with various properties. One useful property of many distributed consensus algorithms is a total (global) ordering of events from multiple sources. This allows all transactions on the associated ledger to have a unique ordering with respect to one another. In the case of key management, for example, the total ordering property makes it easy to establish the ordering of key inception and rotation events. In addition to the aforementioned signed attestations with the private key, a distributed consensus ledger may be used as a source of truth in a DKMI. This may be as a primary (essential) source of truth this coupled to signed attestations or may be merely a secondary (non-essential) source of truth that provides complementary security. A distributed consensus ledger, however, may require a significant amount of infrastructure that must be setup, operated, and

maintained. Typically infrastructure that depends on distributed consensus ledgers must tradeoff cost, throughput, and latency and as a result may not be as scalable or performant as infrastructure that does not depend on a distributed consensus ledger or that minimizes that dependency such as minimizing the number of operations that must be performed on the distributed consensus ledger.

This work explores more scalable and performant DKMI that either minimizes the number of primary operations required on a distributed consensus ledger or merely uses a distributed consensus ledger as a secondary source of truth if at all. One way to avoid or minimize the use of a distributed consensus ledger is to leverage the fact that only the holder of the private key may create events that produce verifiable operations to the keys including the ordering of associated events. Thus a secondary source of truth merely needs to witness events and their ordering not provide the ordering.

1.4 Key Management

The three main key management tasks are key reproduction, key recovery, and key rotation. We call these the three Rs of key management [23]. The focus of this work is key rotation which may be the most difficult.

1.4.1 Reproduction

Key reproduction includes the creation and derivation of (public, private) key pairs and the associated tracking and storage of the private keys. A discussion of key reproduction is provided elsewhere. But in summary one method that simplifies key reproduction tasks is the use of hierarchically deterministic key derivation algorithms that produce ((HD keys) or keychains. An HD key pair is usually derived from a root private key and some deterministic key path. The key path may be public. This means that there is no need to store the derived HD private key which is a security risk because the private key may be rederived on demand from the root key and the public derivation path.

1.4.2 Recovery

Key recovery involves methods for securely backing up or distributing private keys such that they may be recovered in the event that the device holding the private key is lost or damaged. Key recovery approaches are also discussed elsewhere.

1.4.3 Rotation

Key rotation involves methods for securely revoking a key pair and replacing it with a new key pair. Revoke without replace may be accomplished by merely rotating to a null key. Thus rotation may be implemented as a primary operation and revocation (without replace) may be implemented as a special case of rotation. The primary motivation for key rotation is to prevent, mitigate, or recover from an exploit of the private key due to exposure. The primary risk of exposure comes from use of the private key to sign attestations. Creating a signature typically requires loading the private onto a computing device in order to create the signature. Should an attacker exploit or capture the computing device they may gain access to the private key. In addition continued use of the private key to sign attestations may over time reduce the effort to cryptographically break the key. Finally over time cryptographic exploits of a given (public, private) key crypto-system may be discovered thereby rendering the key insecure. Best practice, therefore, is to enable the rotation of a given (public, private) key pair to a new (public, private) key pair either to the same crypto-system or to a stronger crypto-system. In decentralized identity systems, key rotation is useful when the controller of a self-certifying identifier needs to maintain persistent control over that identifier indefinitely despite exploits of the private key(s).

Otherwise in the event of exploit, the controller could just abandon the exploited identifier and create a new identifier with a new (public, private) key pair.

Periodically rotating the key bounds the risk of compromise resulting from exposure over time. This can be used proactively to upgrade the digital signature crypto-system to keep up with advances in computing. The more difficult problem to solve is rotation after a specific exploit may have already occurred. In this case, the exploiter may create a valid signed rotation operation to a key pair under the exploiter's control prior to the original controller detecting the exploit and they rotating to a new key pair under its own control. The exploiter could thereby either "capture" the identifier or create a conflict or race condition where two inconsistent but verifiable rotation events have been created. This work provides a scalable performant protocol using pre-rotation or equivalently one-time rotation keys that solves the problem of secure rotation after an exploit may have occurred.

With self-certifying identifiers special semantics are applied to rotation. The public key associated with the identifier is not changed, but merely the private key that is authoritative for signing attestations is changed. Otherwise the identifier loses its value as an identifier. Consequently in order to verify an attestation belonging to a self-certifying identifier the verifier must know the key rotation history for that identifier. To clarify, the original public key from the initial (public, private) key pair is used to create the identifier. The identifier includes a reference to the public key. The original private key is used to cryptographically sign attestations that prove control over the identifier. The original public key is used to cryptographically verify the signatures. Each rotation operation creates a new (public, private) key pair. A valid rotation operation must be signed at the very least with the original private key. Rotation does not change the identifier. It still references the original public key. After rotation, however, attestations are now signed with the new private key and verified with the new public key. The original private key has been revoked and replaced with the new private key specified in the rotation operation. The new public key is included in the identifier's key rotation history. Validation of an attestation first requires lookup and validation of the key rotation history. The final rotation entry provides the current key pair used to sign and verify attestations.

The key rotation history of digital signing keys used to control an identifier, provides the basis for managing any other data affiliated with the identifier. In general, changes to the value of attributes associated with or under the control of a digital signing key pair may be managed by verifiable signed assertions using the signing key. Thus management of the signing key pair enables management of affiliated data including other keys such as encryption keys.

This work describes a protocol that provides secure verifiable rotation that solves the problem of successful exploit of a given private key due to exposure when that exploit happens sometime after creation and use of the key. It is assumed, however, that the private key remains private for some meaningful time after its creation. This protocol does not address the cryptographic security of a private key in the face of side channel attacks that capture the private key at the time of creation and/or first use nor does it address brute force or other attacks that may break a private key given only the public key and signed attestations. Side channel attacks, however, may be prevented or mitigated in other ways.

2 Nomenclature

2.1 Definitions

A digital *signature* is a string of characters that is produced by a cryptographic operation on a given string of characters (signed text) using the private or signing key from a (public, private) key pair. The public or verifying key may be used in a related cryptographic operation on the

given signed text to verify the validity of the signature. A digital signature has two important properties. The signature is unique to the key pair for the given signed text and the signature is non-repudiable by the signer, that is, only the holder of the private key can create the signature.

A *digest* is a string of characters that is the output of cryptographic hash function on another (usually much longer) string of characters. The important property of a cryptographic hash function is that the digest is highly unique to the longer string of characters on which it was computed. This is called collision resistance. A digest is a space efficient but unique fingerprint of the content of some message. A signed digest is a commitment to the content. Because digital signatures schemes include hash functions, a signature is also a type of digest.

A *self-certifying identifier* is bound to at least one cryptographic digital signature (public, private) key pair. This protocol description uses that the simpler term *identifier* to mean *self-certifying identifier* unless that shorter expression would be ambiguous.

A *controller* is a controlling entity of an *identifier*. An *identifier* has at least one but may have more than one *controller*. Let L be the number of *controllers*. All proper key management *events* on the *identifier* must include a *signature* from the *controller* when there is only one or a *signature* from at least one of the *controllers* when there is more than one. Typically, when there is more than one *controller*, control is established via L signatures, one from each *controller*. Alternatively, with a K of L threshold control scheme, where $K \leq L$, control is established via any set of at least K signatures each one from a subset of at least size K of the L *controllers*. A more sophisticated scheme is a fractional weighted multi-signature.

A *message* as used by the protocol includes a serialized data structure. The data structure must be serialized in order to be digitally signed. Typically, a message includes a serialized data structure and one or more appended digital *signatures*.

An *event* may be conveyed by a *message*. The protocol is primarily concerned with creating, transmitting, validating, and logging *events* as conveyed by *messages*.

A *key management event* or *key event* for short is a special type of *event* that represents a key management operation on the key(s) associated with an *identifier*. These operations are typically related to key rotation. A proper key management event includes a *signature(s)* by the *controller(s)* of the associated *identifier*.

An *inception event* is a *key event* that represents the creation operation of an *identifier* and its initial set of controlling keys. There may be one and only one inception operation performed on an identifier.

A *rotation event* is a *key event* that represents a rotation operation on an identifier that rotates the set of controlling keys to new set. There may be any number of rotation operations in a unique ordered sequence. Any and all rotation operations may only occur after an inception operation.

A *verifier* is an *entity* that cryptographically verifies the signature(s) on an event message.

A *witness* is a somewhat trusted entity that may receive, verify, and store a key management *event* on an *identifier* for which the *witness* is not a *controller*. In addition each *witness* controls its own self-referential *identifier* via an associated (public, private) key pair or pairs. A *witness* may thereby create digital *signatures* on *key event messages* it has received but are associated with other *identifiers* not under its control. Any entity that controls its own self-referential identifier may act as a witness by signing with its own identifier key(s) a copy of a *key event message* for an *identifier* not under its control. Depending on the circumstances of the protocol a set of entities may be designated as trusted or authoritative witnesses. The primary role of a witness is to verify controller signatures attached to an event as well as the event's integrity. In this sense a

witness is an event verifier. The witness also establishes priority to the first version of an event it receives (first seen). The witness signifies this by only signing the first successfully verified version of an event it receives. To restate a witness will never sign any other conflicting version of the same event in a key event sequence.

A *receipt* is a special type of *event* conveyed by a *message* and may include of a copy of a *key management event message* or else a reference to the message with one or more attached *signatures*. This may be referred to as *key event receipt*. A valid key event receipt must include a signature of the associated key event message. Typically each signature comes from one or more *witnesses* or entities acting as witnesses. Indeed the primary purpose of a witness is to generate, store, and disseminate an event receipt for the first verified version of an event the witness receives. A simple witness key event receipt message might include the identifier of the witness, a label for the event that that it is receipting, and a signature by the witness of the receipted event.

A *key event log* (KEL) is an ordered record of all the key event messages created by the controller(s) of an identifier.

A *key event receipt log* (KERL) is an ordered record of all the key event receipt messages for a given identifier created by a given set of witnesses. Typically witnesses keep a KERL for all the identifiers for which they create receipts.

Because a proper key event message includes the signature(s) of the controller(s) it may be thought of as a special type of self-signed receipt. In this same way a KEL may be thought of as a special type of KERL.

A *validator* is an entity that must determine which set of keys are the current controlling set for an identifier that has been subject to rotation operations subject to key operation events by the controller. A validator may do this by replaying the events in a valid KEL.

A *judge* is an entity that examines the entries in a KEL or KERL for a given identifier to determine if the associated events are valid and ultimately to determine which set of keys are the current controlling set given the rotation operations on that given identifier as represented by the events in the log. The primary role of a judge is to validate that a given event sequence has been sufficiently witnessed such that it may be trusted. In this sense a judge is a validator of witnessed events. Each *judge* controls its own self-referential *identifier* via an associated (public, private) key-pair or pairs. A *judge* may thereby create digital *signatures* on attestations about validations it has performed on KELs or KERLS to determine the authoritative key-pair(s) of an identity. A *judge* may be a second party involved in a transaction with a first party controller or a *judge* may be a trusted third party in a multi-party transaction that includes a controller and other parties.

A given entity may act in the roles of both *witness* and *judge*. A *validator* might perform its function by serving as both a witness and judge or by trusting other *witnesses* and *judges*.

This terminology gives rise to the acronym KERI for Key Event Receipt Infrastructure that describes the associated infrastructure consisting of controllers, witnesses, judges, validators and key event (receipt) logs [22].

2.2 Pre-rotation

A described above, one use of key rotation it to recover from a successful exploit of a private key. Given that the private key may be compromised the rotation operation must be authorized using a different private key. One common approach it to use another key-pair just for rotation operations. This has the disadvantage that the rotation key-pair will over time also become exposed due to use and may therefore eventually be at risk of exploit and likewise need to be rotated. To mitigate this yet another key-pair may be used to authorize the rotation of the rotation

key-pair and so on. This comprises a more complex hierarchical key management infrastructure. In contrast, the approach presented here, called pre-rotation, is much simpler [23]. In its simplest form, whenever an identifier is created two key-pairs are created. The first is the original key-pair bound to the identifier and the second is a pre-rotated key-pair that will be activated as a result of the first rotation or the original key-pair. This pre-rotation is declared at initialization. The associated key event is called an *inception* event. Each later rotation operation in turn creates and activates a new pre-rotated key-pair. In essence pre-rotation creates and uses one-time rotation keys. Thus later exploit may not capture the key rotation authority. A rotation operation must be signed by pre-rotated key-pair as declared in the previous rotation or inception operation. Furthermore for enhanced protection against forged key rotation histories each rotation operation must also be signed by the current signing key-pair at the time of rotation. Each rotation operation provides a signed commitment to the next pre-rotated key-pair but using the current controlling key-pair and the previous (erstwhile) controlling key-pair . Because each pre-rotated key is not used to sign anything at the time of its creation nor anything else prior to the next rotation event, it is not thereby exposed to exploit from usage. A subsequent exploit of the current key-pair cannot change that prior commitment to the specific pre-rotated key-pair. This prevents the exploiter from capturing control of the identifier by using the exploited key to rotate to some other key under its control. In order to verify the provenance of the current valid key-pair a validator needs to be able to replay the history of rotation operations.

The pre-rotation approach has some useful features. For many exploits, the likelihood of success is a function of exposure to continued monitoring or probing. Narrowly restricting the opportunity for exploit in terms of time, place, and method, especially if the time and place happen only once, makes exploit extremely difficult. The exploiter has to either predict the one time and place of the event or has to have continuous universal monitoring of all events. By declaring the very first pre-rotation at the inception event of the associated identifier and controlling key-pair, the window for exploit is as narrow as possible. Likewise, each subsequent rotation is a one time and place event as well a one-time usage of the (pre-rotated) rotation key. Pre-rotation does not require any additional key infrastructure for rotation. This makes the approach self-contained. Because each rotation-operation event requires two signatures, one using the current key and the other using the pre-rotated key, an exploiter would have to exploit both keys. This is extremely difficult because the only prior times where the private side of the pre-rotated key-pair is used are (1) at its creation in order to make the associated public key, and (2) at the later signing of the rotation operation event. This minimizes the times and places to a very constrained set.

The caveat with pre-rotation is that the validating party be able to replay the rotation events to ensure that it did not miss a rotation. This replay allows the validator to verify the provenance of the chain of rotations. Requiring both the current and previous signing keys in the rotation operation means that a forger must exploit both keys or sets of keys (when multi-sig).

There are two use cases for providing this replay capability. The first is online one-to-one or pair-wise interactions and the other is offline one-to-one (pair-wise) or equivalently offline one-to-many (any-wise) interactions.

2.2.1 Online One-to-One Replay

In the online one-to-one (pair-wise) case, the first party is a *controller* of a given identifier and wishes to interact with a second party, the *validator*, using that given identifier. The second party must validate that the given identifier is in the control of the first party. To restate, with respect to the given identifier the first party is the *controller* and the second party is the *validator*. Because the identifier is bound to the public key of a key-pair the controller may establish control by sending an *inception* operation event message signed with the corresponding private key.

This *inception* event establishes the controller's control over the identifier from the perspective of the validator. The inception event also includes a commitment to a future pre-rotated key pair. This forward chains the inception event to the next event albeit not to the content but to a future verifiable signature made with the pre-rotated key. Likewise the first *rotation* operation event sent by the controller after the inception event includes a commitment to the next pre-rotated key-pair. Each subsequent rotation operation includes a commitment to the next pre-rotated key. This establishes a forward chain of future commitments to unexposed key-pairs. The private key in a pre-rotated key-pair is not used until a subsequent rotation operation. This prevents an exploiter from capturing control of a future rotation operation with an exposure based attack.

The controller must also establish a commitment to the order of events. Because only the controller can create verifiable events, the controller may uniquely establish the order of events. No other source of truth with respect to ordering is needed. As long as the validator maintains a copy of the original event sequence it will be able to detect a later exploit that attempts to change any of the events in that sequence

There are two ways a controller may establish an event sequence. The first is to use a monotonically increasing sequence number such as an integer counter. The second is to include a backward cryptographic commitment in each event to the contents of the previous event (excepting the inception event). The inclusion of a commitment to the content of the previous event content effectively backward chains the events together in an immutable sequence (a self contained backward event chain). Although a sequence number is simpler and results in smaller messages. the backward chain may be more robust because with backward chaining, possession of only the latest event allows the detection of tampering of any earlier event that is later presented to the validator without it having to keep an original copy of all the events. This prevents an exploit where an alternative sequence of events are created but that terminate with the same key-pair(s) as the original final event. This means the validator does not have to maintain strict custody over its copy of the events but merely the final event in order to detect tampering in any other copy of the event sequence. This relaxes the requirement from maintaining secure custody of the complete key event sequence to merely maintaining secure custody of the last message. Backward chaining comes at the cost of making each event message bigger to include a backward commitment to the previous message.

A backward commitment is usually a collision resistant digest of the serialized content of the previous event. The output of a hash function or a digital signature may be used as a collision resistant digest. Because each event is already signed, using a hash requires extra computation to create. Verification of a hash, however, may take somewhat less time than the verification of a signature. Furthermore, a signature may be longer than a comparable hash. The choice of signature or hash function is a computation space tradeoff [6]. In this work the output of a cryptographic hash function is used for the digest [5].

The validator needs to maintain a log of the key event messages to verify the provenance of the current controlling key-pair for the identifier. This is a *key event log* (KEL). As long as the validator maintains a copy of this log, an exploiter may not establish control of the identifier due to an exposure exploit. For example, a later exploit of the original key-pair could be used to forge a different inception event. As long as the validator has a copy of the original inception event it could detect the forged inception event and ignore it. Likewise later exploits of any of the keys except the last pre-rotated key could be used to forge different rotation events. As long as the validator has a copy of the original chained key rotation event sequence starting with the original inception event it could detect the exploited rotation events and ignore them. Because the last pre-rotated key is never used until a new rotation event is created, the risk of exploit is minimized (exploit must come from via a side-channel attack not exposure). When the key event

sequence is backward chained the validator need merely keep a secure copy of the latest event in order to detect tampering of any copy of the event log. The validator still needs to be provided a copy of all the events to validate the event sequence but should the validator lose custody of the full event long and only maintain custody of the final event it may re-establish the validity of some other copy by backwards validation from its preserved final event.

Absent any other infrastructure, in order that the validator obtain a complete event log, the controller must ensure that the validator has received each and every rotation event in sequence. This requires an acknowledged transfer of each new rotation event. In order to ensure that this occurs the controller and validator must both be online (hence the online case) at the time of the transfer. If either party becomes unavailable the interaction pauses and the other party must wait until both parties are online to resume the interaction. Consequently this case is only useful for interactions where pausing and resuming is acceptable behavior.

The acknowledgment message includes a signature by the validator of the associated key event message, in other words, an event receipt. The controller now has a signed receipt that attests that the validator received and verified the key event message. The validator is thereby in this narrow sense also a *witness* of the event. The controller can keep the receipt in a key event receipt log (KERL). This provides a trust basis for transactions with the validator. The validator may not later repudiate its signed receipts nor use a different key event history in interactions with the controller without detection by the controller.

Each party could establish its own identifier for use with the other in the pair-wise interaction. Each party would thereby in turn be the controller for its own identifier and the validator and witness for the other's identifier. Each could maintain a log of the key events for the other's identifier and key event receipts from the other for its own identifier thereby allowing each to be invulnerable to subsequent exploit of the associated keys with respect to the pair-wise interaction. A log that includes both the signed key events (signed by the controller) and signed key event receipts (signed by the validator) is a log of doubly signed receipts of key events. Any transactions conducted with the associated keys within the time-frame maintained by the logged key event histories may thereby be verified with respect to the keys without the need for other infrastructure such as a distributed consensus ledger. A discussion of how to verify associated transaction events is provided later.

Of particular concern with this approach is the original exchange of the inception event. In a pair-wise interaction, however, the controller may create a unique identifier for use with the associated validator and thereby a unique inception event for that identifier. This inception event is therefore a one time, place, and use event. Consequently, as long as the validator retains a copy of the original inception event, (or when backward chained a copy of any later event), the inception event itself is not subject to later exploit due to exposure from subsequent usage of the originating key-pair. Another way of looking at this approach is that each pair-wise relationship gets a unique set of identifiers and associated key-pairs for each party.

The exchange of the inception event message must also be made invulnerable to man-in-the-middle attacks (for example by using multi-factor authentication) otherwise an imposter (man-in-the-middle) could create a different identifier under its control and confuse the validator about the correct identifier to use in interactions with the genuine controller. A diagram of the pair-wise infrastructure is shown below.

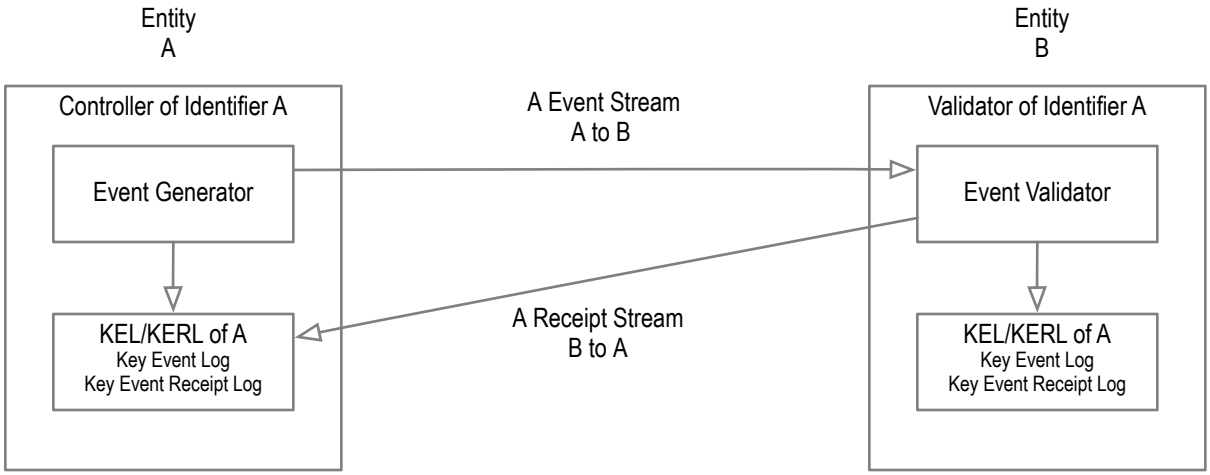

Figure 2.1. Online Pair-wise A to B

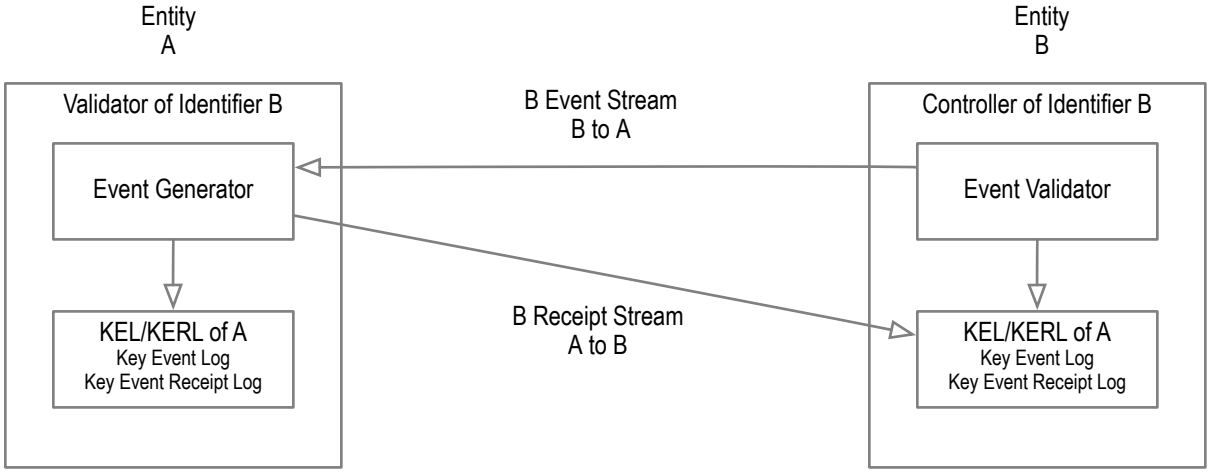

Figure 2.2. Online Pair-wise B to A

2.2.2 Offline One-to-One or Offline One-to-Many Replay

In the offline case a validator may not be available at the time of creation to receive and acknowledge an inception event. Consequently a later exploit of the originating key-pair might allow an exploiter to establish an alternative inception event and provide that instead to a validator thereby preempting the original unexploited inception event. The exploiter could thereby capture control of the identifier from the perspective of the validator. Likewise in the offline case a validator might not be available at the time to receive and acknowledge a rotation event. Consequently a later exploit of the associated key-pairs might allow an exploiter to establish an alternative rotation event and provide that instead to a validator thereby preempting the original unexploited rotation event. The exploiter could thereby capture subsequent control of the identifier from the perspective of the validator. The offline one-to-one case generalizes to the one-to-many case where one controller uses the same identifier for interactions with a set of validators. this might occur when the controller is using a public identifier.

In the offline one-to-one (pair-wise) or one-to-many (any-wise) case, the key rotation history for a given identifier is maintained by a trustworthy service. While a decentralized distributed consensus ledger could provide this service it may not be the minimally sufficient means. This work describes an alternative approach that uses a redundant immutable event log of inception and rotation key event receipts to provide a minimally sufficient means. The service is provided by a set of *N* designated *witnesses* and one or more *judges*. A *validator* may act as either a *wit-*

ness or a *judge* but in this case a *validator* is not one of the *N* designated *witnesses* that provide the service. The *validator* may either be its own *judge* or rely on some other *judge* or *judges*. The objective of the service is to provide at a copy of a *correct* or at least a *complete* key event receipt log to any validator that needs it (*correct* and *complete* are defined below). A redundant immutable (deletion proof) event log may be parallelizable and hence highly scalable.

In this approach, the controller of a given identifier creates and disseminates associated key operation event messages to the set of *N* witnesses. Each witness verifies the signatures and content of each key event it receives. When a verified key event is also the first version of that event the witness has received then it signs the event message and creates a receipt with its signature. The witness then stores the event receipt in its log. The witness sends its receipt as an acknowledgement to the controller. Depending on the dissemination protocol a witness may also send its receipt to other witnesses.

By event *version* it is meant the contents of the event at a given location in the key event sequence. If the event contents are different but at the same location in the sequence then it is a conflicting *version* of the event. The policy is that the first one wins (event version) and is stored in the log all other versions are discarded. Later messages or receipts may not change an existing entry in the log. Each witness adds to its log any verified signatures attached to receipts it receives from any other witness for the same version of the event in its log. Receipts for conflicting event versions are also discarded (not kept in the log) but may be used to proof duplicitous behavior by a witness. A witness' event receipt log is therefore by construction an immutable log. This log includes the events with attached verified signatures; from the controller, the witness itself, and other witnesses, that are the receipts.

Distribution of receipts to the *N* witnesses may be implemented extremely efficiently with respect to network bandwidth by the controller by acting in round-robin fashion. Each time the controller connects to a witness to send new events it also sends the receipts it has received so far from other witnesses. The controller must wait for a receipt from each witness before it can complete the dissemination which may require a second pass around the witnesses. This means, however, that at most *2N* two-way exchanges are needed for each event in order to create a fully witnessed key event receipt log at each and every witness and the controller.

When network bandwidth is not too constrained then a gossip protocol might be a faster but still efficient mechanism for disseminating the events and receipts. A directed acyclic graph or other data structure can be used to determine what needs to be gossiped.

The purpose of having *N* designated witnesses is to better ensure trustworthiness of the service in the event of faults. In addition to faults such as unresponsiveness and network interruptions, allowed faults include what are commonly known as Byzantine faults such as malicious or duplicitous behavior (dishonesty) by the witnesses. Dishonest witnesses may neglect to forward or store event receipts. Nonetheless, as long as the controller is honest and at any time any *M* of the *N* witnesses are non-faulty then for any given event the controller will eventually have a signed receipt with at least *M*+1 signatures in its event log (one from each of *M* witnesses plus one from itself). Likewise given the same conditions, for any given event, eventually each of the non-faulty witnesses will have a signed receipt with at least *M*+1 signatures in its event log. The service is trustworthy because access by any validator or judge to a receipt from a witness on a given event enables the detection of later collusion or duplicitous behavior by that witness in an exploit on that same event. Events from an altered key event history produced by an exploiter will not be accepted by any witness who had previously logged a different version of an event. Consequently a successful exploit must exploit not merely one or more of the controller's keys but must also exploit *M* witnesses. This significantly increases the difficulty of exploit. To minimize

the risk of exploit a controller should take care to select *N* witnesses that are independent and have a small likelihood of colluding to defraud.

We define a *complete* key event receipt log as one where each key event receipt in the sequence has at least *M+1* signatures (one from each of *M* witnesses plus one from the controller). With an honest controller where *M* of the *N* witnesses are non-faulty at any time, a judge or validator is guaranteed to be able to eventually obtain from the service a *complete* key event receipt log from any one of the non-faulty witnesses. A complete key event receipt log is also immutable by construction.

The important result is that at any time the service with provide access by a validator via one of the non-faulty witnesses to a complete key event receipt log that will enable it to validate the controlling keys of the associated identifier over the time-frame of the events in the complete log. A complete key event receipt log may also be provided by any entity that has retained a copy of it not merely the witnesses. Subsequent exploit of the keys or exploit of any or all of the witnesses or the controller may not exploit any of the events in the complete key event receipt log. The following diagram shows the offline key event infrastructure.

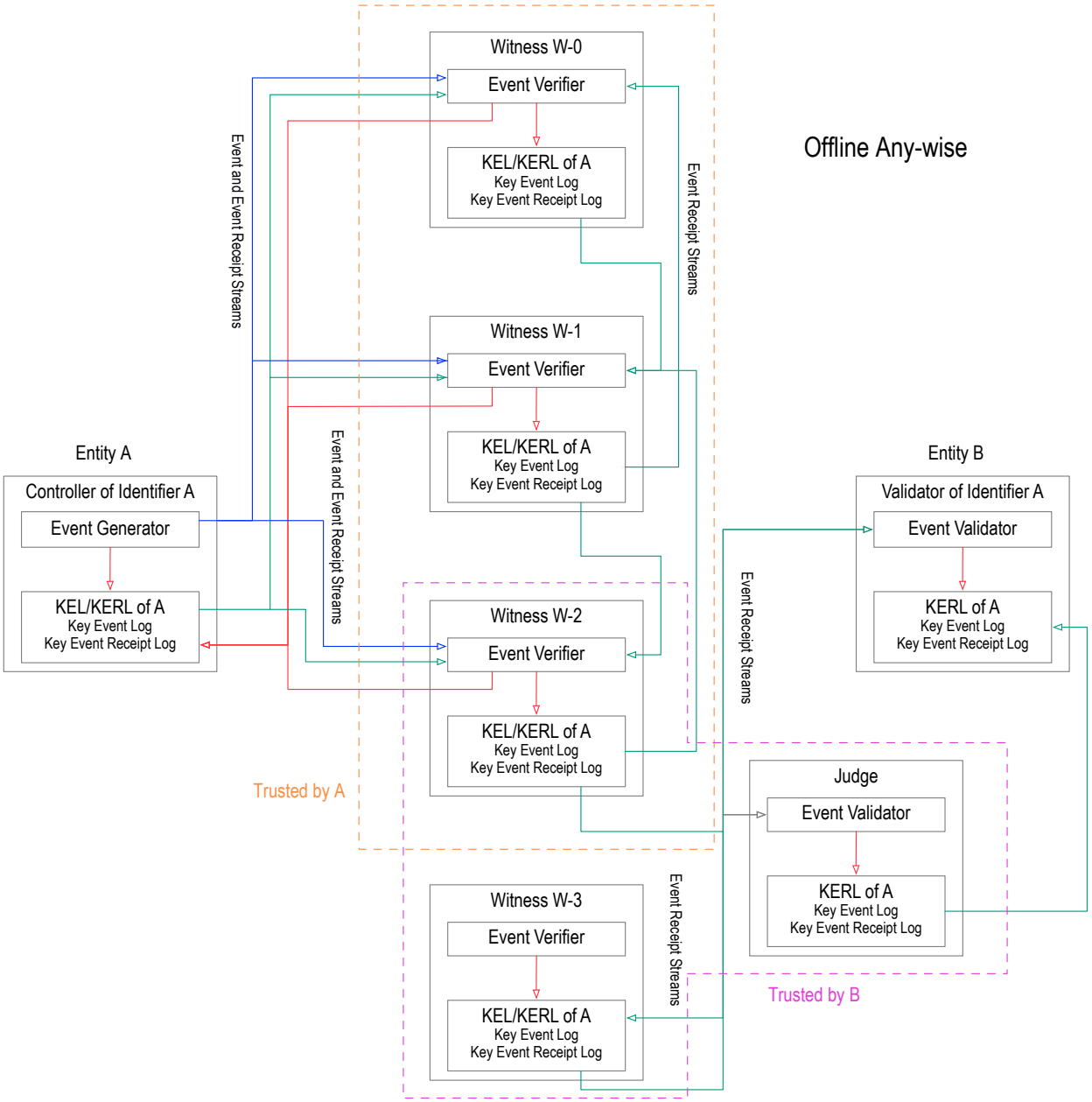

Figure 2.3. Offline Any-wise A to Whoever

To summarize, in order to fool a validator or judge into accepting an erroneous or exploited key event history, a successful exploiter must forge a complete key event receipt log but with a different sequence of key events. To do this the exploiter must not only exploit one or more of the controller's signing keys (assuming an honest controller) but also exploit *M* of the *N* designated witnesses. Typically a successful exploit of a witness would involve gaining access to the witness' signing key. In rare cases, a simpler exploit could succeed if the exploit of the controller's key(s) happened before a witness received a verified copy of the true event. Then even an unexploited (honest) witness would assume the exploited event was valid and unwittingly collude with a set of dishonest witnesses. In order to create a complete event receipt log the exploiter would still need to induce another *M*-1 witnesses into supplying receipts of the exploited event. This exploit requires a race condition where some witnesses are unresponsive long enough for the key exploit due to exposure to occur. This may take some time. This race condi-

tion would be easy for the controller to prevent by replacing unresponsive witnesses. This mitigation is discussed later.

The assumption of an honest controller means the the validator must trust the controller. This is a reasonable assumption in many cases. It it usually in the best interests of the controller to act in a trustworthy manner with respect to its own identifiers. Furthermore because each key event must be signed by the controller any inconsistent key events would be detectable by any validator that eventually had access to both events. This would destroy both the trustworthiness of the controller with respect to that identifier and any value it may have built up in the identifier. Because the main purpose of key rotation is to allow a controller to maintain persistent control over an identifier, detectably inconsistent behavior removes any benefit of persistent control of a given identifier.

Nevertheless in cases where duplicity provides some temporary benefit, a controller might be tempted to act dishonestly (or equivalently an exploited might be tempted to fully exploit a controller). One way to mitigate the risk to a validator of a dishonest controller is for the validator to require that the witness pool include witnesses that the validator itself trusts. A validator might provide to the controller the identifiers of the set of trusted witnesses. The validator would then only accept an identifier from the controller if the controller designates one or more of the validator's trusted witnesses as witnesses to the key event receipt history of the controller's identifier.

Nevertheless, when M is less than a majority of N, a dishonest controller or fully exploited controller might be able to induce different disjoint subsets of M witnesses into each eventually producing a divergent (inconsistent) but complete (and immutable) key event receipt log for the same identifier. This might be used to temporarily create a race condition that the dishonest controller may be able to exploit. Any validator, judge, or witness, however, merely needs to anonymously query more than M witnesses to detect the inconsistency. This limits the exploit potential of a dishonest controller albeit at some cost to the participants.

Requiring that the number of non-faulty witnesses M be a majority of N at any time adds an additional property that will prevent a dishonest controller from ever inducing the creation of more than one version of a complete key event receipt log for a given identifier. When M is a majority of N then in order for two inconsistent key event receipt logs (different sequence of key events) to be created an honest witness would have to create a receipt for two conflicting versions of the same event which an honest witness would never do. Consequently, no other *complete* log may be created with a different sequence of events despite a dishonest or fully exploited controller because at least one of the M witnesses must be honest. To see how this is so, consider that if M is a majority of N and M witnesses are non-faulty then the number of faulty witnesses F must be less than M, that is,

$$F = N - M < M \ . \tag{2.1}$$

A *complete* event log will also be *correct* whenever there may not be any other complete event log with a different sequence of key events. This will be true when M is a majority of N. To restate, a *complete* event receipt log is also a *correct* event receipt log when completeness requires that each event receipt is signed by a majority of the witnesses.

When M is a majority of N then even despite a dishonest or fully exploited controller, the service may either only produce a correct event receipt log or only produce incomplete event receipts logs but may not produce multiple complete but divergent event receipt logs. In other words, should the service ever produce a complete event receipt log then that event receipt log will also be correct. In the case of an honest controller then the service will produce at least one *correct* event receipt log which may be provided to a validator by any of the latest set of M non-

faulty witnesses. When the controller is responsive it will also produce a correct event receipt log which it may provide to a validator.

The most important result is that when *M* is a majority of *N* where *M* witnesses are non-faulty at any time, then the service will either provide to a validator or judge a correct and complete key event receipt log (when the controller is honest) or it may not provide a complete log at all (much less correct). This protects the validator from a dishonest controller. A correct log enables validation of the controlling keys of the associated identifier over the time-frame of the events in the log. Once produced a correct key event receipt log may be provided by any entity that has retained a copy of it not merely the witnesses. Subsequent exploit of the controller's keys and/or exploit of less than *M* of witnesses may not invalidate any of the events in a correct key event receipt log. Moreover, even in the extreme case that a full exploit of the controller and all the witnesses has occurred and given that they produce a correct but divergent event receipt log, a prior copy of a correct key event receipt log will enable detection and proof that an exploit has occurred. In this extreme case some other mechanism may be used to resolve the production of more than one complete but divergent event receipt log. This might be through an anchor transaction to the event log on a distributed consensus ledger to establish priority of the original event sequence. This approach minimizes the use of a distributed consensus ledger to only that of resolving the most extreme and unlikely cases.

Moreover, distributed consensus algorithms employed by blockchain ledgers, typically, require multiple rounds of exchanges between replicants (members of the consensus pool) in order to come to consensus. In some protocols each round requires a number of exchanges that scales with N^2. In contrast, the witnessed key event receipt log algorithm presented here is much more scalable because the algorithm only needs two rounds of receipt exchanges between the controller and witnesses for each new event. The total number of exchanges scale with as little *2N*. Using a gossip protocol and/of DAG for this algorithm may allow very efficient tradeoffs between throughput and network bandwidth. In general the requirements for this witnessed protocol are much less than that of a distributed consensus ledger and hence a much more minimally sufficient means for decentralized key management infrastructure (KERI for DKMI) even in cases of extreme exploit

2.2.3 Judge

Validation by a *judge* of an event receipt log adds additional security. Like a *witness*, the *judge* first verifies the controller signatures and event content for each event. The *judge* also verifies all the witness signatures for each event and validates that there are at least *M* verified witness signatures. The operation associated with the event is now in force or active until it is superseded by a later event. The result of the operation is now applied when verifying subsequent events in the log. If an event is not complete (i.e. does not have at least *M* verified witness signatures then the validation stops and the associated key event receipt log is deemed invalid by the judge. A given implementation of a judge may apply additional rules to resolve the rare case of full exploit of the controller and *M* witnesses that may thereby produce multiple valid complete but divergent key event receipt logs. The judge may use the first received or resort to some other trust source including other judges. This could include a median time stamp of the astronomical time at the time of reception of the inception event or some other consensus algorithm with respect to other judges such a distributed consensus ledger.

2.3 Representations

2.3.1 Key-pair

Each entity uses its own identifier when participating in the protocol. That identifier is represented with an upper case letter symbol such as *A*, *B*, *C* etc. Each identifier has a a controlling (public, private) key pair. Without loss of generality that key pair is represented with the same letter symbol as the entity's identifier but where the public key is uppercase and the private key is lower case. For example, entity *A* has a key pair (*A*, *a*) where *A* denotes the public key and *a* denotes the private key. Successive controlling key pairs are indicated with a superscript. For example *(*A^0, a^0) for the initial key pair and (A^1, a^1) for the next pair. When it is not important to distinguish which key pair is used but merely to indicate that the latest valid or authoritative key pair is to be used whichever index it may be then the superscripts are left off. The key pair then becomes (*A*, *a*). Usually the context determines which member of the key pair, the public key for verification or private key for signing, is to be used. In those cases, without loss of clarity only the uppercase may be used to represent the appropriate member of the key pair.

2.3.2 Entity

Individual entities that are members of a class of similar entities may be represented by a subscripted uppercase letter symbol where the letter symbol denotes the class and the subscript denotes the member. Suppose for example that *C* denotes the class of controllers. Then C_0 and C_1 denote two different controllers. This convention may be combined with superscripts to denote successive controlling key pairs for each controller, such as, $\left(C_0^0, c_0^0\right)$ and $\left(C_0^1, c_0^1\right)$ for the first two key pairs of C_0, and $\left(C_1^0, c_1^0\right)$ and $\left(C_1^1, c_1^1\right)$ for the first two key pairs of C_1. Likewise $\left(C_0, c_0\right)$ for the latest key pair of C_0 and $\left(C_1, c_1\right)$ for the latest key pair of C_1.

2.3.3 Serialized Data

A data structure with ordered fields may is represented by a tuple where the tuple is denoted with parentheses about a comma separated list of field names, such as, $\left(t, A, C\right)$. A serialized version of that data structure is denoted with angle brackets about a comma separated list of the field names, such as, $\langle t, A, C \rangle$. In general a tuple may be converted to a data structure with labeled key value pairs. Ordering is important when serializing.

2.3.4 Signature

A digital signature operation is represented with the lowercase greek sigma, σ where the symbol of the signing key pair is provided as a subscript to the sigma and the serialized data or text is provided as the argument to the signing operation. For example, the digital signature for entity *A* may be denoted as follows:

$$\sigma_{A^0}\left(\langle t, A, C \rangle\right), \tag{2.2}$$

where the signing key is the private key from the key pair, $\left(A^0, a^0\right)$ controlled by *A* and the signed text is the serialized data structure, $\langle t, A, C \rangle$. When appropriate an alternative form would use the lowercase symbol for the private key instead, as follows;

$$\sigma_{a^0}\left(\langle t, A, C \rangle\right). \tag{2.3}$$

But as stated above when there is no doubt as to the context, the uppercase symbol may be used to represent the appropriate member from the key pair. A message that includes both a serialized data structure and an attached signature may be denoted as follows:

$$\langle t, A, C \rangle \sigma_{A^0} \left(\langle t, A, C \rangle \right). \tag{2.4}$$

A message with two attached signatures, one each from entities *A* and *B* may be denoted as follows:

$$\langle t, A, C \rangle \sigma_{A^0} \left(\langle t, A, C \rangle \right) \sigma_{B^0} \left(\langle t, A, C \rangle \right), \tag{2.5}$$

where the signing key for B is the private key from the key pair, $\left(B^0, b^0 \right)$ and the other parts are as above. Without confusion, in the expression above the arguments to the signing operations may be redundant. Therefore a more compact form of the preceding expression may be denoted as follows:

$$\langle t, A, C \rangle \sigma_{A^0} \sigma_{B^0}. \tag{2.6}$$

Expressions eq. 2.5 and eq. 2.6 are equivalent. When the attached signatures use the latest key pair from both *A* and and *B* then an even more compact form of eq. 2.6 may be denoted as follows:

$$\langle t, A, C \rangle \sigma_A \sigma_B. \tag{2.7}$$

2.3.5 Event

When the message is a key event message for an identifier then the greek epsilon, ε may be used to label the event with a subscript representing its position in the event sequence, such as, ε_0. When the particular identifier the event belongs too is not provided by the context then the ε_0 is used as a subscript on the symbol representing the identifier, such as, A_{ε_0}. For example a key event message for identifier *A* might be labeled as follows:

$$\langle t, A, A^0, A^1 \rangle \sigma_{A^0} = A_{\varepsilon_0} = \varepsilon_0. \tag{2.8}$$

2.3.6 Receipt

A receipt of a key event message consists of that key event message together with an attached signature made by a witness of that event message. A key event receipt may include other information such as the identifier of the witness and/or the witnesses public key needed to verify its signature. In general a witness may use a specific identifier for a given KERL (key event receipt log). Should the keys for the identifier become compromised the witness may recover from the compromise by either rotating the compromised key and resuming operation as a witness under the same identifier or cease acting as a witness using that identifier. An exploiter may not change previously created receipts and inconsistent receipts from the same witness will invalidate the witness. In the later approach the witness may resume acting as a witness but under a new identifier and must therefore recreate all its receipts using the new identifier. Although either approach may be made to work, the later approach avoids recursive rotation validation of KERL entries (where the KERL entries of witnesses to a controller are validated and then the KERL entries of witnesses the witnesses are validated and so on). Recursive validation complicates the implementation. Consequently, without loss of generality, in this work witnesses may not rotate keys associated with their identifiers but must create a new identifier when exploited. Let W_0 represent a witness that uses its only signing key pair. A minimal receipt created by W_0 of event A_{ε_0} may be denoted in long form as follows:

$$\left\langle \left\langle t, A, A^0, A^1 \right\rangle \sigma_{A^0} \left(\left\langle t, A, A^0, A^1 \right\rangle \right) \right\rangle \sigma_{W_0} \left(\left\langle t, A, A^0, A^1 \right\rangle \sigma_{A^0} \left(\left\langle t, A, A^0, A^1 \right\rangle \right) \right) \tag{2.9}$$

where $\sigma_{W_0} \left(\left\langle t, A, A^0, A^1 \right\rangle \sigma_{A^0} \left(\left\langle t, A, A^0, A^1 \right\rangle \right) \right)$ is the the digital signature created by W_0 using its key-pair $\left(W_0, w_0 \right)$ of the key event $\left\langle t, A, A^0, A^1 \right\rangle \sigma_{A^0} \left(\left\langle t, A, A^0, A^1 \right\rangle \right)$. The second set of enclosing brackets in expression eq. 2.9 indicates what text is signed by σ_{W_0}. The equivalent short form of this receipt may be expressed as follows:

$$\left\langle \left\langle t, A, A^0, A^1 \right\rangle \sigma_{A^0} \right\rangle \sigma_{W_0} \tag{2.10}$$

where the second set of enclosing brackets once again indicates what text is signed by σ_{W_0}. An even shorter form would use the event label $A_{\varepsilon_0} = \varepsilon_0$ (when A is given by context) as follows:

$$\left\langle A_{\varepsilon_0} \right\rangle \sigma_{W_0} = \left\langle \varepsilon_0 \right\rangle \sigma_{W_0} \tag{2.11}$$

A receipt of the same event signed by two witnesses, W_0 and W_1 may be represented in short form as follows:

$$\left\langle \left\langle t, A, A^0, A^1 \right\rangle \sigma_{A^0} \right\rangle \sigma_{W_0} \sigma_{W_1} = \left\langle A_{\varepsilon_0} \right\rangle \sigma_{W_0} \sigma_{W_1} = \left\langle \varepsilon_0 \right\rangle \sigma_{W_0} \sigma_{W_1} . \tag{2.12}$$

A receipt may be labeled with the lowercase greek rho, ρ with subscripts representing the witnesses who created the attached signatures and the argument of the receipt being either the associated key event or its label. This gives the compact receipt label form as follows:

$$\rho_{W_0 W_1} \left(\left\langle t, A, A^0, A^1 \right\rangle \sigma_{A^0} \right) = \left\langle \left\langle t, A, A^0, A^1 \right\rangle \sigma_{A^0} \right\rangle \sigma_{W_0} \sigma_{W_1} . \tag{2.13}$$

or the extremely compact receipt and event label form as follows:

$$\rho_{W_0 W_1} \left(A_{\varepsilon_0} \right) = \left\langle A_{\varepsilon_0} \right\rangle \sigma_{W_0} \sigma_{W_1} = \left\langle \left\langle t, A, A^0, A^1 \right\rangle \sigma_{A^0} \right\rangle \sigma_{W_0} \sigma_{W_1} , \tag{2.14}$$

or when the context gives us A then

$$\rho_{W_0 W_1} \left(\varepsilon_0 \right) = \left\langle \varepsilon_0 \right\rangle \sigma_{W_0} \sigma_{W_1} . \tag{2.15}$$

2.3.7 Receipt Message

A key event receipt message conveys information about the receipt. Typically the recipient of a key event receipt message already has a copy of the associated key event message. Consequently there is no need to retransmit a copy of original key event in the receipt message. Likewise the recipient of the receipt may know the identity of the sending witness and thereby its public key. The only required element is the witness' signature of the key event. Nonetheless, in addition to the signature it may be convenient to include in the receipt message the identifier and/or public key of the witness and a label of the event. This makes the message bigger but may be easier to track and manage especially with an asynchronous communications channel. For a witness identifier W_0 that is bound to $\left(W_0^0, w_0^0 \right)$ its key event receipt message may be expressed as follows:

$$\left\langle t_k, W_0, \sigma_{W_0} \right\rangle \tag{2.16}$$

where t_k is the label of the associated key event, W_0 is the witness' identifier, and σ_{W_0} is the signature of the associated key event, The message itself does not need to be signed by the witness because the only important information the message conveys is that the witness

"witnessed" the associated key event. The witness' signature is non-repudiable so a verified copy of the signature is sufficient to convey that information. In order to verify the signature the recipient of the message needs to have a copy of the original key event message and know the signature's verification (public) key. The event label and witness identifier and in the receipt message enable convenient lookup of that information.

2.3.8 Digest

The digest of a key event message is string of characters produced by a cryptographic hash function that is a space efficient but unique fingerprint of the content of the message. A signed digest is a commitment to the message content. The serialized data or text is provided as the argument to the hash function. A digest may be labeled with the lowercase greek eta, η. A digest of serialized data structure $\langle t, A, A^0, A^1 \rangle$ may be denoted as:

$$\eta\left(\langle t, A, A^0, A^1 \rangle\right). \tag{2.17}$$

A digest of a full event message comprised of a serialized data structure and an attached signature may be denoted as follows:

$$\eta\left(\langle t, A, A^0, A^1 \rangle \sigma_{A^0}\right) = \eta\left(A_{\varepsilon_0}\right) = \eta\left(\varepsilon_0\right), \tag{2.18}$$

where $\langle t, A, A^0, A^1 \rangle \sigma_{A^0} = A_{\varepsilon_0} = \varepsilon_0$ are various denotations of the event message.

2.3.9 Single and Multiple Signature

The descriptions so far have assumed that the current control authority over the identifier at any point in time is based on a single key-pair. One way to increase security is to use a multi-signature approval schemes where multiple simultaneous key-pairs jointly control the identifier. This would require multiple signers to authorize or approve key management events and verifiable attestations associated with the identifier. Multi-signature approval may add an extra layer of security thereby making exploit more difficult. They may also lessen the risk of loss of a single private key and make key recovery more robust. Multiple signature rotation, however, can be somewhat complex. To simplify and clarify the descriptions of both the single and multiple signature versions of the protocols some terminology is introduced in this section.

The sequence of key-pairs used to control an identifier may be denoted with integer index j, such as, $\left(C^j, c^j\right)$. The zeroth index, where $j = 0$, indexes the originating key-pair, $\left(C^0, c^0\right)$, whose public key, C^0 is bound to the identifier C and controlled by private key c^0. A sequence of indexed public keys may be used to represent the sequence of key-pairs as follows:

$$C^0, C^1, C^2, \ldots, \tag{2.19}$$

where C^j is the j^{th} public key in the sequence for a controller labeled C.

The sequence of rotation operations may be denoted with integer index l. Each rotation operation changes the controlling key-pair(s) for the identifier. When the controlling key-pair is singular, then each rotation consumes one new key-pair from the sequence. In this case, the indices of the key-pair sequence and rotation sequence will have the same value, that is, $j = l$. In this notation the key-pair $\left(C^l, c^l\right)$ is also the l^{th} key-pair in the sequence of key-pairs controlled by C. When the controlling key-pair is not singular, however, (i.e. multi-signature) then each rotation operation may consume more than one key-pair from the sequence of controlling key-pairs. In this case then the indices j and l may not always be equal, i.e. $j \neq l$. Let the number of control-

ling key-pairs for the l^{th} rotation be denoted L_l. The l^{th} rotation thereby consumes L_l key-pairs from the sequence of controlling key-pairs. The key-pairs consumed by the l^{th} rotation form a subsequence of length L_l. The subsequence of indices *j* for the subsequence of key-pairs for the rotation *l* of length L_l may be denoted as follows.

$$\left[r_l, r_l+1, r_l+2, \ldots, r_l+L_l-1\right] \tag{2.20}$$

where r_l is the value of index *j* for the first key-pair in the subsequence. This means that

$$r_{l+1} = r_l + L_l . \tag{2.21}$$

The originating key-pair $\left(C^0, c^0\right)$ is declared by an inception operation that creates the identifier *C*. The inception operation may be thought of as a special case of rotation that is the zeroth rotation operation, that is, $l=0$. Also let $r_0 = 0$ be the value of the first index from the inception or zeroth rotation. Given that $r_0 = 0$ then the value of the index r_l for $l>0$ may be computed by summing the lengths, L_l, of subsequences of key-pairs consumed by all the prior rotations. This may be expressed as follows:

$$r_l = \sum_{i=0}^{l-1} L_i \Big|_{l>0} . \tag{2.22}$$

where $r_0 = 0$.

The subsequence of public keys for the l^{th} rotation may be denoted as follows:

$$\left[C^{r_l}, \ldots, C^{r_l+L_l-1}\right]_l , \tag{2.23}$$

with

$$\left[C^{r_0}\right]_0 = \left[C^0\right]. \tag{2.24}$$

Suppose for example that the inception uses one key-pair, that is, $L_0 = 1$ and the following three rotations use three, three, and four key-pairs respectively, that is, $L_1 = 3$, $L_2 = 3$, and $L_3 = 4$. The resultant starting indices for each rotation subsequence are as follows:

$$\begin{aligned} r^0 &= 0 \\ r^1 &= \sum_{i=0}^{0} L_i = L_0 = 1 \\ r^2 &= \sum_{i=0}^{1} L_i = L_0 + L_1 = 4 \\ r^3 &= \sum_{i=0}^{2} L_i = L_0 + L_1 + L_2 = 7 \\ r^4 &= \sum_{i=0}^{3} L_i = L_0 + L_1 + L_2 + L_3 = 11 \end{aligned} \quad . \tag{2.25}$$

Furthermore the resultant subsequences of public-keys for each rotation in order are as follows:

$$\begin{aligned} \left[C^{r_0}\right]_0 &= \left[C^0\right] \\ \left[C^{r_1}, C^{r_1+1}, C^{r_2+2}\right]_1 &= \left[C^1, C^2, C^3\right] \\ \left[C^{r_2}, C^{r_2+1}, C^{r_2+3}\right]_2 &= \left[C^4, C^5, C^6\right] \\ \left[C^{r_3}, C^{r_3+1}, C^{r_3+2}, C^{r_3+4}\right]_3 &= \left[C^7, C^8, C^9, C^{10}\right] \end{aligned} \tag{2.26}$$

With the nomenclature presented above we can now more efficiently describe the protocol.

2.4 Miscellaneous

2.4.1 Sufficient Majority

Some parts of KERI are dependent on what is called a *sufficient majority*. Given an allowed number of faulty members *F*, the number of *sufficient majority* of members *M* less than the total *N* may be pre-determined. The smallest possible majority is called a *simple majority*. The *simple majority* $\hat{M}$ For *N* members is given by,

$$\hat{M} = \text{int}\left(\frac{N}{2}\right) + 1, \tag{2.27}$$

where the value of the int() function is the nearest integer less than or equal to its argument. In some cases a *sufficient majority* is a *super majority* that is greater than a *simple majority*.

The set of entities that form a sufficient majority is called a *quorum*. If *N* is the number of entities and *M* is the number of a sufficient majority of *N* then the number of possible *quorums* is given by the number of distinct combinations of *N* taken *M* at a time. This is represented as,

$$C(N,M) = \frac{N!}{M!(N-M)!}. \tag{2.28}$$

Typically at each stage of processing of any given event, an algorithm may require that a *quorum* of members participate in making a decision. The members of the quorum at one stage may be different from the members of the quorum at another stage. Quorum membership at any stage, however, may be a universally verifiable item of knowledge. Changes in quorum membership are provided by rotation events that also change quorum membership. These are therefore *quorum membership events*.

2.4.2 Sequence Numbers

Each event has a unique monotonically increasing sequence number. This may be used to order sort, index, and organize events. In the following messages the field label *t* is used to represent the sequence number with a numeric subscript to indicate the order of sequence number such as $t_0, t_1, \ldots, t_n$. A simple format for *t* is a counter with integer values starting at 0 that increment by 1 for each new event. When the sequence number is a counter then the subscript value of *t* will equal the value of the sequence number, that is,

$$t_i = i, \tag{2.29}$$

and

$$t_0 = 0, t_1 = 1, \ldots, t_n = n. \tag{2.30}$$

2.4.3 Date-time Stamps

It may be convenient to include absolute (real, astronomical) date-time stamps in messages or log entries. One well known date-time format is the ISO-8601 standard [12; 13]. An ISO-8601 time zone aware UTC date time stamp with up to nanosecond resolution has the following form:

YYYY-MM-DDTHH:MM:SS.mmmmmmmmm+00:00 (2.31)

An example is

`2000-01-01T00:00:00.000000000+00:00` (2.32)

2.4.4 Cypher Suite

Best practices cryptography limits the options that user may choose from for the various cryptographic operations, such as signing, encrypting, and hashing to a suite of balanced and tuned set of protocols, one for each operation. Each member of the set should be the *one and only one* best suited to that operation. This prevents the user from making bad choices. Many key-representation schemes allow the user the freedom to specify the features of the operation independently. This is usually a very bad idea [4; 24; 25]. Users should not be custom combining different features that are not part of a best practices cypher suite. Each custom configuration may be vulnerable to potential attack vectors. The suggested approach is to narrowly specify a single cypher suite family and version for each operation. If an exploit is discovered for a member of a suite and then fixed, the suite is updated wholly to a new version. The number of allowed cypher suites should be minimized to those essential for compatibility but no more. This approach increases expressive power because only one syntactic element is needed to specify a suite instead of a different element per feature.

In the following event expressions a particular cryptographic system or cypher suit for creating the signatures is either assumed known or else is explicitly specified. A suggested specification format is that described in the W3C DID (Decentralized Identifier) report for the *type* field in the authentication section of a DID Document [26]. This is single string that includes a cypher suite family, operation, and version. For example the family = Ed25519, operation = verification, and version = 2018 may be expressed as follows:

```
Ed25519VerificationKey2018                    (2.33)
```

One way to specify the cypher suite is to include an additional *kind* field in each message with a value (cypher suite type) as given above. The problem with that approach is that all the public keys must use the same *kind*. This does not allow support for rotations that change or upgrade the cypher suite nor does it support a mixture of cypher suites for multi-signature schemes. In general, the approach used herein is to treat each occurrence of a public key field in a message as a data structure that may optionally include the cypher suite type in a *kind* field when not using the default cypher suite kind if any. An example data structure in JSON follows:

```
{
    key: "Xq5YqaL6L48pf0fu7IUhL0JRaU2_RxFP0AL43wYn148=",
    kind: Ed25519VerificationKey2018
}
```

2.4.5 Encoding

Public keys, signatures, and digests are large integers. They must be encoded in order to be consistently represented when stored, transmitted over networks, or included in serialized data structures that are signed or hashed. A suggested approach is to use the RFC-4648 Base64 URL-File Safe encoding standard. as per RFC-4648 [14] to encode these items. The contents of the messages in this work are intended to be processed by computer algorithms and not transcribed by humans. Consequently Base64 is more appropriate and more compact than other representations such as Base58-check. The main advantage of Base58-check is that it is less prone to error when manually transcribing. Because this behavior should be discouraged in general Base58-check is more cumbersome, Base64 is preferred.

2.4.6 Serialization

A suggested approach for serializing data structures or tuples is to use the JSON standard. One of the limitations of JSON is that the field order of a serialized JavaScript object is not norma-

tive, that is, a valid JSON serialization does not guarantee the order of appearance of the fields within a JavaScript object (or Python dict) that results from the deserialization. Likewise whitespace in a JSON serialization is not normative. Consequently round trip serializations and deserializations may not be identical and therefore would not verify against the same cryptographic signature. This is the so called canonicalization problem. One simple solution to this problem when using JSON is that the data associated with a signature may only be serialized once by the signer. Users of the data may deserialize but never re-serialize unless they also re-sign. Any compliant JSON deserialization will produce an equivalent Javascript object (same field names and values but order and whitespace are ignored).

2.4.7 Signatures

In many protocols the signatures are attached to the serialized data in one data string. This encounters another limitation of JSON, that is, many JSON implementations raise an error if a deserialization attempt on a string does not consume all the characters in the string. Thus a hybrid data string that consists of a serialized JSON object followed by a signature string might be difficult to deserialize with some JSON implementations. A portable approach, however, is to concatenate the signature but separate it from the JSON serialization with a unique string of characters that will not be produced by a JSON serializer. A parser first searches for the separator string and then separately extracts both the JSON serialization and the signature. One such human friendly separator string is the 4 character sequence of whitespace characters, CR LF CR LF (in ascii notation), where CR represents the CarriageReturn character (ASCII 13) and LF represents the LineFeed character (ASCII 10). In escaped notation this string is "\r\n\r\n". If these separator characters were to appear within a quoted string they would be doubled escaped by the JSON serializer. Furthermore, this separator string does not appear in a Base64 serialization. Consequently this same separator can be used to attach multiple signatures. This approach makes it easy for a parser to separate, extract, and verify the serialized data with the attached signature(s) without deserializing the JSON. In this work serializations consist of the JSON followed by one or more sets of the separator "\r\n\r\n' and a JSON string delimited (double quoted) Base64 serialization of a signature. An example follows:

```
{
  "id": "Xq5YqaL6L48pf0fu7IUhL0JRaU2_RxFP0AL43wYn148=",
  "sn": 1,
  "kind": "ek",
  "signer": "Xq5YqaL6L48pf0fu7IUhL0JRaU2_RxFP0AL43wYn148=",
  "ensuer": "Qt27fThWoNZsa88VrTkep6H-4HA8tr54sHON1vWl6FE="
}
\r\n\r\n
"AeYbsHot0pmdWAcgTo5sD8iAuSQAfnH5U6wiIGpVNJQQoYKBYrPPxAoIc1i5SHC
IDS8KFFgf8i0tDq8XGizaCg=="
```

When using HTTP as the transport it may be more convenient to attach the signatures in an HTTP Header. One way to do this is with a custom header. The format of the custom *Signature* header follows the conventions of RFC7230 [10]. The header format is as follows:

```
Signature: headervalue

Headervalue:
  tag = "signature"
```

```
or
  tag = "signature"; tag = "signature"  ...
```

where the name of each tag is given a unique string to identify each signature.

Each signature tag value is a doubly quoted string "" that contains the actual signature in Base64 url safe format.

An optional tag named type may be present of the form:

```
  kind = "kind"
```

where the value of the kind tag is a string that specifies the cypher suit type used to create the signature(s) within that header. All signatures within a header must be of the same cypher suite type.

Multiple *Signature* headers may be present and the set of signatures are the union of all the signatures from the *Signature* headers.

If the same signature tag appears multiple times then only the last occurrence is used.

An example is shown below with a kind tag, one signature tag named “signer” and the other signature tag named “ensuer”.

```
Signature:
kind="Ed25519VerificationKey2018";signer="Y5xTb0_jTzZYrf5SSEK2f3
LSLwIwhOX7GEj6YfRWmGViKAesa08UkNWukUkPGuKuu-EAH5U-
sdFPPboBAsjRBw==";ensuer="Xhh6WWGJGgjU5V-
e57gj4HcJ87LLOhQr2Sqg5VToTSg-
SI1W3A8lgISxOjAI5pa2qnonyz3tpGvC2cmf1VTpBg=="
```

The convention is that each signature tag name must equal a field name in the JSON body. The associated JSON field value is the public key used to verify the signature provided in the corresponding header value.

2.4.8 Event Ilk

It may be convenient to include in key events a key event *ilk* field that specifies what type of event the message contains. This may aid in parsing the event message. The suggested field name is *ilk*. Suggested *ilk* values and events are as follows: `icp` for *inception* event, `rot` for *rotation* event, `rct` for event *receipt* message, `itc` for generic *interaction* event, `dip` for delegated *inception* event, `drt` for delegated *rotation* event, `dwd` for *domestic witness designation* event, and `fwd` for *foreign witness designation* event. Specific applications may use other event ilks.

3 Online One-to-One Protocol

In this version of the protocol there are no declared witnesses. The defined events reference entities with identifiers that are bound to key-pairs. A controller has an identifier labeled *C* that is bound to a key-pair $\left(C^0, c^0\right)$. Likewise a validator has an identifier labeled *V* that is bound to a key-pair $\left(V^0, v^0\right)$. Because the likelihood of collision of public keys generated with cryptographic strength digital key systems is remote, therefore, with loss of generality, in the event de-

scriptions below, the identifier bound to a key-pair may be equal to the public key. For example, $C = C^0$ and $V = V^0$. Using only the public key enables the core parts of a KERI implementation to be independent of the syntax for specific types of self-certifying identifiers. In the case of DIDs this means that the KERI core may be DID method independent. Indeed the KERI core may be made independent of any identifier syntax other than that the identifier be uniquely bound to a cryptographic strength signing (public, private) key-pair. The universally unique identifier for the core may be set equal to the public key and identifier specific syntax for other sections of the identifier may be handled by an identifier specific client.

It is also assumed that the cypher suite *type* used to generate signatures is either known or else each appearance of a public key (when not the base identifier) in the events is actually a data structure that includes the public key and an optional cypher suite *type* when it is not the default *type* (see the discussion in Section 2.4). This caveat applies to the public keys referenced in the events below.

In the following event descriptions, the index *k* indexes all events of any ilk and index *l* indexes only rotation events. In general the k^{th} event may not be the same as the l^{th} rotation event. Only when the all events are rotation events will $k = l$. The inception event is special it always has $k = l = 0$. The inception event is considered a special case of a rotation event. Other events may be interspersed between the inception event and the first rotation event as well as between subsequent rotation events.

There are two types of events, these are chained and unchained. Chained events include a digest of the previous event (except the inception event). To avoid redundancy, because, the only difference between chained and unchained events is the inclusion of the digest field in chained events, the event definitions will provide a full description of the chained version but only provide the denotation of the unchained version. The inception event has no digest because it is the first or zeroth event in the sequence. The digest backward chains the events and reinforces the sequence ordering provided by the sequence number (see Section 2.8). Some applications where economy of expression is more important than security may choose to use the non-chained versions of the protocols. To restate, when chained the order of events is provided by a sequence number but reinforced with a backward chained digest. The sequence of digests and the sequence number must correspond.

3.1 Single Signature

In this scheme only a single key-pair controls the identifier at any time. A multi-signature scheme is provided later.

3.1.1 Inception Event

Given a controller with identifier labeled *C* the serialized key *inception* event message labeled C_{ε_0} is denoted as follows:

$$\varepsilon_0 = C_{\varepsilon_0} = \left\langle C, t_0, \texttt{icp}, C^0, C^1 \right\rangle \sigma_{C^0}, \tag{3.1}$$

where the identifier *C* is uniquely bound to the key-pair $\left(C^0, c^0\right)$, $t_0 = 0$ is the unique monotonically increasing sequence number for this event , `icp` is the event *ilk* representing an inception event, the controlling signing key-pair is indicated by C^0, which is the public key from the key-pair $\left(C^0, c^0\right)$ that is controlled by the private key, c^0, the ensuing pre-rotated key-pair is indicated by C^1 which is the public key from the key-pair $\left(C^1, c^1\right)$ that is controlled by the private key,

c^1, and σ_{C^0} is the digital signature of the contents of the brackets, $\langle\ \rangle$, made with private key c^0. The inception event demonstrates control over the identifier public key C via the signature σ_{C^0}. It implicitly declares that $\left(C^0, c^0\right)$ is the current controlling key-pair and that $\left(C^1, c^1\right)$ will be the ensuing controlling key pair after rotation.

A tuple of the labels for the fields in the event may be denoted as follows:

$$\left(id, sn, ilk, signer, ensuer\right) \tag{3.2}$$

These labels may be used as the field names in a JSON serialized version of the key inception event as follows:

```
{
  "id": "Xq5YqaL6L48pf0fu7IUhL0JRaU2_RxFP0AL43wYn148=",
  "sn": 0,
  "ilk": "icp",
  "signer": "Xq5YqaL6L48pf0fu7IUhL0JRaU2_RxFP0AL43wYn148=",
  "ensuer": "Qt27fThWoNZsa88VrTkep6H-4HA8tr54sHON1vWl6FE="
}
\r\n\r\n
"AeYbsHot0pmdWAcgTo5sD8iAuSQAfnH5U6wiIGpVNJQQoYKBYrPPxAoIc1i5SHC
IDS8KFFgf8i0tDq8XGizaCg=="
```

As mention above, when the cypher suite is not know then the appearance of a public key (not the "id" field) may be replaced with a data structure that include both the public key and the cypher suit kind. This may be denoted as follows:

$$\varepsilon_0 = C_{\varepsilon_0} = \left\langle C, t_0, \texttt{icp}, \left(C^0, kind\right), \left(C^1, kind\right)\right\rangle \sigma_{C^0}. \tag{3.3}$$

where *kind* is a string defining the cypher suite. The value of each appearance of *kind* may be different.

A serialized version of this representation follows:

```
{
  "id": "Xq5YqaL6L48pf0fu7IUhL0JRaU2_RxFP0AL43wYn148=",
  "sn": 0,
  "ilk": "icp",
  "signer":
  {
     "key":"Xq5YqaL6L48pf0fu7IUhL0JRaU2_RxFP0AL43wYn148=",
     "kind":"Ed25519VerificationKey2018"
  },
  "ensuer":
  {
     "key":"Qt27fThWoNZsa88VrTkep6H-4HA8tr54sHON1vWl6FE=",
     "kind":"EcdsaSecP256VerificationKey2018"
  },

}
\r\n\r\n
```

```
"AeYbsHot0pmdWAcgTo5sD8iAuSQAfnH5U6wiIGpVNJQQoYKBYrPPxAoIc1i5SHC
IDS8KFFgf8i0tDq8XGizaCg=="
```

Although not provided in other sections this example may be used as a reference for how to modify the JSON serializations to accommodate varied *kind* values for the various public keys.

3.1.2 Inception Event Receipt

A validator, *V*, may create an inception event receipt by signing with its associated key-pair. Its signature of the inception event may be denoted as follows:

$$\sigma_V\left(\left\langle C_{\varepsilon_0}\right\rangle\right)=\sigma_V\left(\left\langle C,t_0,\mathtt{icp},C^0,C^1\right\rangle\sigma_{C^0}\right). \tag{3.4}$$

The associated event receipt may then be denoted using compact notation as follows:

$$\rho_V\left(C_{\varepsilon_0}\right)=\left\langle C_{\varepsilon_0}\right\rangle\sigma_V=\left\langle\left\langle C,t_0,\mathtt{icp},C^0,C^1\right\rangle\sigma_{C^0}\right\rangle\sigma_V. \tag{3.5}$$

All events share the same key event receipt message format. This is described in Section 5.3.

3.1.3 Chained Rotation Event

The l^{th} rotation event operation occurring as the k^{th} event in the sequence for a controller labeled *C* that is provided via a serialized key *rotation* event message labeled, C_{ε_k}, may be denoted as follows:

$$\varepsilon_k=C_{\varepsilon_k}=\left\langle C,t_k,\eta\left(\varepsilon_{k-1}\right),\mathtt{rot},C^{l+1}\right\rangle\sigma_{C^{l-1}}\sigma_{C^l} \tag{3.6}$$

where the identifier *C* is uniquely bound to the key-pair $\left(C^0,c^0\right)$, with public key, C^0, which is controlled by the private key, c^0, $t_k=k$ is the unique monotonically increasing sequence number for this event , $\eta\left(\varepsilon_{k-1}\right)$ is the digest of the preceding, $(k-1)^{th}$ event, `rot` is the event *ilk* representing a rotation event, C^{l+1} is the public key from the ensuing key-pair, $\left(C^{l+1},c^{l+1}\right)$ which is controlled by the private key, c^{l+1}, and the pair of digital signatures $\sigma_{C^{l-1}}$ (erster) and σ_{C^l} (signer) made with private keys c^{l-1} and c^l from key-pairs $\left(C^{l-1},c^{l-1}\right)$ and $\left(C^l,c^l\right)$ sign the contents of the brackets, $\langle\ \rangle$. The inclusion of the erster signature chains the signing authority of successive rotation events which provides additional protection against forgery of key event rotation operations in a key event history as two keys must be exploited not only one. In this notation the key-pair $\left(C^l,c^l\right)$ is the l^{th} key-pair in the pre-rotated sequence of keys controlled by *C*. This means that the previous rotation event declared $\left(C^{l-1},c^{l-1}\right)$ as the signing key-pair and $\left(C^l,c^l\right)$ as the ensuing key-pair but now this rotation event implicitly declares $\left(C^l,c^l\right)$as the current signing key-pair and explicitly declares $\left(C^{l+1},c^{l+1}\right)$as the ensuing pre-rotated key-pair in the sequence. This makes $\left(C^{l-1},c^{l-1}\right)$ the erstwhile (erster) signing key-pair or erster.

As a clarifying example, when the first rotation event is also the next event after the inception event then $k=l=1$ and may be denoted as follows:

$$\varepsilon_1=C_{\varepsilon_1}=\left\langle C,t_1,\eta\left(\varepsilon_0\right),\mathtt{rot},C^2\right\rangle\sigma_{C^0}\sigma_{C^1}, \tag{3.7}$$

where the identifier C is uniquely bound to the key-pair $\left(C^0, c^0\right)$, with public key, C^0, which is controlled by the private key, c^0, $t_1 = 1$ is the sequence number for this event , `rot` is the event *ilk* for rotation, C^2 is the public key from the ensuing key-pair $\left(C^2, c^2\right)$ which is controlled by private key, c^2, and the digital signatures σ_{C^0} and σ_{C^1} made with private keys c^0 and c^1 from key-pairs $\left(C^0, c^0\right)$, $\left(C^1, c^1\right)$ sign the contents of the brackets, $\langle\ \rangle$. The inception event declared $\left(C^1, c^1\right)$ as the ensuing pre-rotated key pair which via this rotation event becomes the current signing key-pair. This rotation event also declares $\left(C^2, c^2\right)$ as the (next) second pre-rotated key-pair.

A tuple of the field labels may be denoted as follows:

$$\left(id, sn, digest, ilk, ensuer\right) \tag{3.8}$$

These labels may be used as the field names in a JSON serialized version of the key rotation event message as follows:

```
{
  "id": "Xq5YqaL6L48pf0fu7IUhL0JRaU2_RxFP0AL43wYn148=",
  "sn": 1,
  "digest": "GEj6YfRWmGViKAesa08UkNWukUkPGsdFPPboBAsjRBw=",
  "ilk": "rot",
  "ensuer": "UhL0JRaU2_RxFP0AL43wYn148Xq5YqaL6L48pf0fu7I="
}
\r\n\r\n
"AeYbsHot0pmdWAcgTo5sD8iAuSQAfnH5U6wiIGpVNJQQoYKBYrPPxAoIc1i5SHC
IDS8KFFgf8i0tDq8XGizaCg=="
\r\n\r\n
"KFFgf8i0tDq8XGizaCgAeYbsHot0pmdWAcgTo5sD8iAuSQAfnH5U6wiIGpVNJQQ
oYKBYrPPxAoIc1i5SHCIDS8=="
```

3.1.4 Chained Rotation Event Receipt

A validator, V, may create a rotation event receipt by signing with its associated key-pair. Its signature of the l^{th} rotation event operation occurring as the k^{th} event in the sequence may be denoted as follows:

$$\sigma_V\left(\left\langle C_{\varepsilon_k}\right\rangle\right) = \sigma_V\left(\left\langle C, t_k, \eta\left(\varepsilon_{k-1}\right), \mathtt{rot}, C^{l+1}\right\rangle \sigma_{C^{l-1}} \sigma_{C^l}\right). \tag{3.9}$$

The associated event receipt may then be denoted using compact notation as follows:

$$\rho_V\left(C_{\varepsilon_k}\right) = \left\langle C_{\varepsilon_k}\right\rangle \sigma_V = \left\langle \left\langle C, t_k, \eta\left(\varepsilon_{k-1}\right), \mathtt{rot}, C^{l+1}\right\rangle \sigma_{C^{l-1}} \sigma_{C^l}\right\rangle \sigma_V . \tag{3.10}$$

All events share the same key event receipt message format. This is described in Section 5.3.

3.1.5 Unchained Rotation Event

When unchained the order of events is provided solely by the sequence number. In this case, the l^{th} rotation event operation occurring as the k^{th} event in the sequence for a controller labeled

C that is provided via a serialized key *rotation* event message labeled, C_{ε_k}, may be denoted as follows:

$$\varepsilon_k = C_{\varepsilon_k} = \left\langle C, t_k, \mathtt{rot}, C^{l+1} \right\rangle \sigma_{C^{l-1}} \sigma_{C^l} \tag{3.11}$$

where the field definition are the same as provided in the definition of eq. 3.6. The associated rotation event receipt and message are modified appropriately with the content above.

3.2 Multiple Signature

In this section the single signature scheme is extended to a multiple signature scheme. Review Section 2.9 for a more in-depth definition of some of the associated terminology.

3.2.1 Inception Event

Given a controller with identifier labeled C the serialized multi-signature key *inception* event message labeled C_{ε_0} is denoted as follows:

$$\varepsilon_0 = C_{\varepsilon_0} = \left\langle C, t_0, \mathtt{icp}, C^0, K_1, \widehat{C}_1 \right\rangle \sigma_{C^0}, \tag{3.12}$$

where the identifier C is uniquely bound to the key-pair $\left(C^0, c^0\right)$, with public key, C^0, which is controlled by the private key, c^0, $t_0 = 0$ is the unique monotonically increasing sequence number for this event , `icp` is the event *ilk* representing an inception event, the controlling signing key-pair is indicated by C^0, which is the public key from the key-pair $\left(C^0, c^0\right)$ that is controlled by the private key c^0, K_1, is the threshold number of required signatures out of L_1 signers after the ensuing rotation, $\widehat{C}_1 = \left[C^1, \ldots, C^{L_1}\right]_1$ is the list of ensuing public keys that form a set of L_1 potential signers for the ensuing rotation to a multi-signature, and σ_{C^0} is the digital signature of the contents of the brackets, $\langle\ \rangle$, made with private key c^0. The inception event demonstrates control over the identifier public key C via the signature σ_{C^0}. It explicitly declares that $\left(C^0, c^0\right)$ is the current controlling key-pair and that K_1 of the L_1 signers in $\widehat{C}_1 = \left[C^1, \ldots, C^{L_1}\right]_1$ will be the ensuing controlling key-pairs after rotation (K of L multi-signature). In general the inception may be considered as a special case of rotation, that is, the zeroth rotation that implies that $K_0 = L_0 = 1$. and $C = C^0$. The inception pre-rotates the subsequence $\left[C^1, \ldots, C^{L_1}\right]$ as the ensuing set of key-pairs. The inception event demonstrates control of the identifier but is not multi-signature itself. Consequently, when using a multiple signature scheme the inception event must be immediately followed by a rotation event to activate the first set of multiple signatures declared in the inception event. This rotation will immediately turn control over to the K of L multi-signature scheme previously specified in the inception event.

As a clarifying example, suppose the ensuing multi-signature is 2 of 3, that is, $K_1 = 2$ and $L_1 = 3$. The corresponding inception event may be denoted as follows:

$$C_{\varepsilon_0} = \left\langle C, t_0, \mathtt{icp}, C^0, 2, \left[C^1, C^2, C^3\right] \right\rangle \sigma_{C^0}, \tag{3.13}$$

where C^0, C^1, C^2, and C^3 are public keys from their associated key-pairs.

A tuple of the field labels may be denoted as follows:

$$\left(id, sn, ilk, signer, threshold, ensuers\right) \tag{3.14}$$

These labels may be used as the field names in a JSON serialized version of the key inception event as follows:

```
{
  "id": "Xq5YqaL6L48pf0fu7IUhL0JRaU2_RxFP0AL43wYn148=",
  "sn": 0,
  "ilk": "icp",
  "signer": "Xq5YqaL6L48pf0fu7IUhL0JRaU2_RxFP0AL43wYn148=",
  "threshold": 2,
  "ensuers":
  [
    "Qt27fThWoNZsa88VrTkep6H-4HA8tr54sHON1vWl6FE=",
    "VrTkep6H-4HA8tr54sHON1vWl6FEQt27fThWoNZsa88=",
    "HA8tr54sHON1Qt27fThWoNZsa88VrTkep6H-4vWl6FE="
  ]
}
\r\n\r\n
"AeYbsHot0pmdWAcgTo5sD8iAuSQAfnH5U6wiIGpVNJQQoYKBYrPPxAoIc1i5SHC
IDS8KFFgf8i0tDq8XGizaCg=="
```

3.2.2 Inception Event Receipt

A validator, V, may create an inception event receipt by signing with its associated key-pair. Its signature of the inception event may be denoted as follows:

$$\sigma_V\left(\left\langle C_{\varepsilon_0}\right\rangle\right)=\sigma_V\left(\left\langle C,t_0,\mathtt{icp},C^0,K_1,\widehat{C}_1\right\rangle\sigma_{C^0}\right), \tag{3.15}$$

The associated event receipt may then be denoted using compact notation as follows:

$$\rho_V\left(C_{\varepsilon_0}\right)=\left\langle C_{\varepsilon_0}\right\rangle\sigma_V=\left\langle\left\langle C,t_0,\mathtt{icp},C^0,K_1,\widehat{C}_1\right\rangle\sigma_{C^0}\right\rangle\sigma_V. \tag{3.16}$$

All events share the same key event receipt message format. This is described in Section 5.3.

3.2.3 Chained Rotation Event

The rotation event format enables changes to both the threshold and number of signatories with each rotation. The l^{th} multi-signature key *rotation* event event operation occurring as the k^{th} event in the sequence for a controller labeled *C* that is provided via a serialized key *rotation* event message labeled, C_{ε_k}, may be denoted as follows:

$$\varepsilon_k=C_{\varepsilon_k}=\left\langle C,t_k,\eta\left(\varepsilon_{k-1}\right),\mathtt{rot},K_{l+1},\widehat{C}_{l+1},\widehat{e}_l,\widehat{s}_l\right\rangle\widehat{\sigma}_l, \tag{3.17}$$

where the identifier *C* is uniquely bound to the key-pair $\left(C^0,c^0\right)$, with public key, C^0, which is controlled by the private key, c^0, $t_k=k$ is the unique monotonically increasing sequence number for this event , $\eta\left(\varepsilon_{k-1}\right)$ is the digest of the preceding, $(k-1)^{th}$, event, `rot` is the event *ilk* representing a rotation event, K_{l+1}, is the threshold number of required signatures out of L_{l+1} signers after the ensuing rotation, $\widehat{C}_{l+1}=\left[C^{r_{l+1}},\ldots,C^{r_{l+1}+L_{l+1}-1}\right]_{l+1}$ is the list of ensuing public keys that form a set of L_{l+1} potential signers for the ensuing rotation to a multi-signature, r_{l+1} is the starting index of the subsequence of controlling key-pairs for the $(l+1)^{th}$ rotation (see Section

2.9), $\hat{e}_l = \left[e_0, \ldots, e_{E_l-1} \right]_l$ is a list of length E_l of integer indices into the list, $\left[C^{r_{l-1}}, \ldots, C^{r_{l-1}+L_{l-1}-1} \right]$, of ersters (erstwhile signers) as indicated by an earlier rotation event, r_{l-1} is the starting index of the subsequence of controlling key-pairs for the $(l-1)^{th}$ rotation (see Section 2.9), $\hat{s}_l = \left[s_0, \ldots, s_{S_l-1} \right]_l$ is a list of length S_l of integer indices into the list, $\left[C^{r_l}, \ldots, C^{r_l+L_l-1} \right]$, of signers for this the l^{th} rotation event but as declared in the $(l-1)^{th}$ rotation event, r_l is the starting index of the subsequence of controlling key-pairs for the l^{th} rotation (see Section 2.9), and $\hat{\sigma}_l$ is the set of digital signatures that sign the contents of the brackets, $\langle\ \rangle$. The inclusion of the ersters signatures chains the signing authority of successive rotation events which provides additional protection against forgery of key event rotation operations in a key event history as two sets of keys must be exploited not only one. The number of signatures in the set $\hat{\sigma}_l$ is equal to the sum of the lengths of the ersters and signers lists, that is, $E_l + S_l$. Each element of the ersters and signers lists produce one signature. The appropriate index into the sequence of controlling key-pairs is computed from the offset given by each element of associated ersters and signers lists added to the starting index of the respective subsequence of key-pairs. This may be denoted as follows:

$$. \hat{\sigma}_l = \sigma_{C^{e_0+r_{l-1}}} \ldots \sigma_{C^{e_{E_l-1}+r_{l-1}}} \sigma_{C^{s_0+r_l}} \ldots \sigma_{C^{s_{S_l-1}+r_l}} \tag{3.18}$$

In this derivation, the key-pair $\left(C^j, c^j \right)$ is the j^{th} key-pair in a pre-rotated sequence of key-pairs controlled by *C*. Each rotation indexed by *l* consumes one or more of these key-pairs. Furthermore, the list of zero based integer indices of ersters $\hat{e}_l = \left[e_0, \ldots, e_{E_l-1} \right]_l$ is of length E_l and satisfies $K_{l-1} \le E_l \le L_{l-1}$. Each index refers to a key-pair from the ordered list of public keys $\left[C^{r_{l-1}}, \ldots, C^{r_{l-1}+L_{l-1}-1} \right]$ taken as a subsequence of the set of all $\left(C^j, c^j \right)$. The indices may be used to generate a list of signing ersters with the notation $\left[C^{e_0+r_{l-1}}, \ldots, C^{e_{E_l-1}+r_{l-1}} \right]$ (see Section 2.9). From this list the associated E_l signatures $\sigma_{C^{e_0+r_{l-1}}} \ldots \sigma_{C^{e_{E_l-1}+r_{l-1}}}$ are produced. Likewise, the list of zero based integer indices of signers $\hat{s}_l = \left[s_0, \ldots, s_{S_l-1} \right]_l$ is of length S_l and satisfies $K_l \le S_l \le L_l$. Each index refers to a key-pair from the list of public keys $\left[C^{r_l}, \ldots, C^{r_l+L_l-1} \right]$ taken as a subsequence of the set of all $\left(C^j, c^j \right)$. The indices may be used to generate a list of signing signers with the notation $\left[C^{s_0+r_l}, \ldots, C^{s_{S_l-1}+r_l} \right]$ (see Section 2.9). From this list the associated S_l signatures $\sigma_{C^{s_0+r_l}} \ldots \sigma_{C^{s_{S_l-1}+r_l}}$ are produced. Together there are $E_l + S_l$ signatures attached to the event. The combination comprises $\hat{\sigma}_l$ defined in eq. 3.18.

This rotation event declares that the previously declared set of key-pairs in the subsequence $\left[C^{r_l}, \ldots, C^{r_l+L_l-1} \right]$ as the new controlling set that replaces the set of key-pairs in the subsequence $\left[C^{r_{l-1}}, \ldots, C^{r_{l-1}+L_{l-1}-1} \right]$. Furthermore this rotation event declares that the set of key-pairs in the subsequence $\hat{C}_{l+1} = \left[C^{r_{l+1}}, \ldots, C^{r_{l+1}+L_{l+1}-1} \right]_{l+1}$ will be the ensuing (pre-rotated) set.

As a clarifying example, let the sequence of inception event plus 3 rotation be as follows:

$$
\begin{aligned}
\varepsilon_0 &= \left\langle C, t_0, \texttt{icp}, C^0, 2, \left[C^1, C^2, C^3\right]\right\rangle \sigma_{C^0} \\
\varepsilon_1 &= \left\langle C, t_1, \eta(\varepsilon_0), \texttt{rot}, 2, \left[C^4, C^5, C^6\right], [0]_1, [0,2]_1\right\rangle \sigma_{C^0}\sigma_{C^1}\sigma_{C^3} \\
\varepsilon_2 &= \left\langle C, t_2, \eta(\varepsilon_1), \texttt{rot}, 3, \left[C^7, C^8, C^9, C^{10}\right], [1,2]_2, [0,1]_2\right\rangle \sigma_{C^2}\sigma_{C^3}\sigma_{C^4}\sigma_{C^5} \\
\varepsilon_3 &= \left\langle C, t_3, \eta(\varepsilon_2), \texttt{rot}, 2, \left[C^{11}, C^{12}, C^{13}\right], [0,1,2]_3, [1,2,3]_3\right\rangle \sigma_{C^4}\sigma_{C^5}\sigma_{C^6}\sigma_{C^8}\sigma_{C^9}\sigma_{C^{10}}
\end{aligned}
\quad . \quad (3.19)
$$

The inception event, ε_0, always implies that $K_0 = 1$, $L_0 = 1$, and $r_0 = 0$. From this we can compute $r_1 = 1$. The inception event also declares that $K_1 = 2$ and by the given length of the ensuers list that $L_1 = 3$. From this we can compute $r_2 = 4$. These declarations mean that the ensuing multi-signature scheme will be 2 of 3. The signature is σ_{C^0}.

The first rotation event, ε_1, declares that $K_2 = 2$ and by the given length of the ensuers list that $L_2 = 3$. From this we can compute $r_3 = 7$. The ersters list in this first rotation event is of length $E_1 = 1$. Because $K_0 = 1$ and $L_0 = 1$, it may have only one entry. From its entry, $\left[e_0 = 0\right]_1$ we may compute the index $j = e_0 + r_0 = 0 + 0 = 0$ of the associated key-pair. This gives the derived list of ersters as $\left[C^0\right]$. This corresponds to the original key-pair for the identifier. The signers list in this rotation must have at least $K_1 = 2$ but may have up to $L_1 = 3$ entries. In this case it has $S_1 = 2$ entries, $\left[s_0 = 0, s_1 = 2\right]_1$. From these entries we may compute the indices of the signing key-pairs. These indices are $j = s_0 + r_1 = 0 + 1 = 1$ and $j = s_1 + r_1 = 2 + 1 = 3$. This gives the derived list of signers as $\left[C^1, C^3\right]$. The associated signatures are $\sigma_{C^0}\sigma_{C^1}\sigma_{C^3}$.

The second rotation event, ε_2, declares that $K_3 = 3$ and by the given length of the ensuers list that $L_3 = 4$. From this we can compute $r_4 = 11$. The ersters list in this rotation must have at least $K_1 = 2$ but may have up to $L_1 = 3$ entries. In this case it has 2 entries. From its entries, $\left[e_0 = 1, e_1 = 2\right]_2$ and $r_1 = 1$ we may derive the list of ersters as $\left[C^2, C^3\right]$. The signers list in this rotation must have at least $K_2 = 2$ but may have up to $L_2 = 3$ entries. In this case is has 2 entries. From its entries $\left[s_0 = 0, s_1 = 1\right]_2$ and $r_2 = 4$. we may derive the list of signers as $\left[C^4, C^5\right]$. The associated signatures are $\sigma_{C^2}\sigma_{C^3}\sigma_{C^4}\sigma_{C^5}$.

The third rotation event, ε_3, declares that $K_4 = 2$ and by the given length of the ensuers list that $L_4 = 3$. From this we can compute $r_5 = 14$. The ersters list in this rotation must have at least $K_2 = 2$ but may have up to $L_2 = 3$ entries. In this case it has 3 entries. From its entries, $\left[e_0 = 0, e_1 = 1, e_1 = 2\right]_3$ and $r_2 = 4$ we may derive the list of ersters as $\left[C^4, C^5, C^6\right]$. The signers list in this rotation must have at least $K_3 = 3$ but may have up to $L_3 = 4$ entries. In this case is has 3 entries. From its entries $\left[s_0 = 1, s_1 = 2, s_1 = 3\right]_3$ and $r_3 = 7$. we may derive the list of signers as $\left[C^8, C^9, C^{10}\right]$. The associated signatures are $\sigma_{C^4}\sigma_{C^5}\sigma_{C^6}\sigma_{C^8}\sigma_{C^9}\sigma_{C^{10}}$.

A tuple of the field labels may be denoted as follows:

$$
\left(id, sn, digest, ilk, threshold, ensuers, ersters, signers\right) \quad (3.20)
$$

These labels may be used as the field names in a JSON serialized version of the key rotation event message, ε_1, as follows:

```
{
  "id": "Xq5YqaL6L48pf0fu7IUhL0JRaU2_RxFP0AL43wYn148=",
  "sn": 1,
  "digest": "GEj6YfRWmGViKAesa08UkNWukUkPGsdFPPboBAsjRBw=",
  "ilk": "rot",
  "threshold": 2,
  "ensuers":
  [
    "WoNZsa88VrTkep6HQt27fTh-4HA8tr54sHON1vWl6FE=",
    "8tr54sHON1vWVrTkep6H-4HAl6FEQt27fThWoNZsa88=",
    "VrTkep6HHA8tr54sHON1Qt27fThWoNZsa88-4vWl6FE="
  ],
  "ersters": [0],
  "signers": [0,2]
}
\r\n\r\n
"AeYbsHot0pmdWAcgTo5sD8iAuSQAfnH5U6wiIGpVNJQQoYKBYrPPxAoIc1i5SHC
IDS8KFFgf8i0tDq8XGizaCg=="
\r\n\r\n
"KFFgf8i0tDq8XGizaCgAeYbsHot0pmdWAcgTo5sD8iAuSQAfnH5U6wiIGpVNJQQ
oYKBYrPPxAoIc1i5SHCIDS8=="
\r\n\r\n
"Hot0pmdWAcgTo5sKFFgf8i0tDq8XGizaCgAeYbsD8iAuSQAfnH5U6wiIGpVNJQQ
oYKBYrPPxAoIc1i5SHCIDS8=="
```

3.2.4 Chained Rotation Event Receipt

A validator, V, may create a rotation event receipt by signing with its associated key-pair. Its signature of the l^{th} rotation event operation occurring as the k^{th} event in the sequence may be denoted as follows:

$$\sigma_V\left(\left\langle C_{\varepsilon_k}\right\rangle\right). \tag{3.21}$$

In longer form this may be denoted as follows:

$$\sigma_V\left(\left\langle C, t_k, \eta\left(\varepsilon_{k-1}\right), \texttt{rot}, K_{l+1}, \widehat{C}_{l+1}, \widehat{e}_l, \widehat{s}_l\right\rangle\widehat{\sigma}_l\right). \tag{3.22}$$

The associated event receipt may then be denoted using compact notation as follows:

$$\rho_V\left(C_{\varepsilon_k}\right) = \left\langle C_{\varepsilon_k}\right\rangle\sigma_V. \tag{3.23}$$

In longer form as follows:

$$\left\langle\left\langle C, t_k, \eta\left(\varepsilon_{k-1}\right), \texttt{rot}, K_{l+1}, \widehat{C}_{l+1}, \widehat{e}_l, \widehat{s}_l\right\rangle\widehat{\sigma}_l\right\rangle\sigma_V. \tag{3.24}$$

All events share the same key event receipt message format. This is described in Section 5.3.

3.2.5 Unchained Rotation Event

The rotation event format enables changes to both the threshold and number of signatories with each rotation. When unchained the order of events is provided solely by the sequence number. In this case, the l^{th} multi-signature key *rotation* event event operation occurring as the

k^{th} event in the sequence for a controller labeled *C* that is provided via a serialized key *rotation* event message labeled, C_{ε_k}, may be denoted as follows:

$$\varepsilon_k = C_{\varepsilon_k} = \left\langle C, t_k, \mathtt{rot}, K_{l+1}, \widehat{C}_{l+1}, \widehat{e}_l, \widehat{s}_l \right\rangle \widehat{\sigma}_l \tag{3.25}$$

where the field definition are the same as provided in the definition of eq. 3.17. The associated rotation event receipt and message are modified appropriately with the content above.

4 Offline One-to-One or One-to-Many Protocol

In this version of the protocol there are declared witnesses and declared changes or "rotations" to the witness list. The following events reference entities with identifiers that are bound to key-pairs. A controller has an identifier labeled *C* that is bound to a key-pair $\left(C^0, c^0\right)$. Likewise a validator has an identifier labeled *V* that is bound to a key-pair $\left(V^0, v^0\right)$. A set of witnesses indexed by subscript *i* each have an identifier $W_i\big|_{i=0,1,2,\ldots}$ that is bound to a key-pair $\left(W_i^0, w_i^0\right)$. For example witness identifier W_0 is bound to $\left(W_0^0, w_0^0\right)$. Because the likelihood of collision of public keys generated with cryptographic strength digital key systems is remote, therefore, with loss of generality in the event descriptions below, the identifier bound to a key-pair may be equal to the public key. For example, $C = C^0$, $V = V^0$, and $W_i = W_i^0$. Using only the public key enables the core parts of a KERI implementation to be independent of the syntax for specific types of self-certifying identifiers. In the case of DIDs this means that the KERI core may be DID method independent. Indeed the KERI core may be made independent of any identifier syntax other than that the identifier be uniquely bound to a cryptographic strength signing (public, private) key-pair. The public key becomes the universally unique identifier for the core. Likewise identifier specific syntax for other sections of the identifier may be handled by an identifier specific client.

It is also assumed that the cypher suite *kind* used to generate signatures is either known or else each appearance of a public key in the events is actually a data structure that includes the public key and an optional cypher suite *kind* when it is not the default *kind* (see the discussion in Section 2.4). This caveat applies to the public keys (not the *id* field) referenced in the events below.

In the following event descriptions, the index *k* indexes all events of any *ilk* and index *l* indexes only rotation events. In general the k^{th} event may not be the same as the l^{th} rotation event. Only when the all events are rotation events will $k = l$. The inception event is special it always has $k = l = 0$. The inception event is considered a special case of a rotation event. Other events may be interspersed between the inception event and the first rotation event as well as between subsequent rotation events.

There are two types of events, these are chained and unchained. Chained events include a digest of the previous event (except the inception event). To avoid redundancy, because, the only difference between chained and unchained events is the inclusion of the digest field in chained events, the event definitions will provide a full description of the chained version but only provide the denotation of the unchained version. The inception event has no digest because it is the first or zeroth event in the sequence. The digest backward chains the events and reinforces the sequence ordering provided by the sequence number (see Section 2.8). To restate, when chained the order of events is provided by a sequence number but reinforced with a backward chained digest. The sequence of digests and the sequence number must correspond.

4.1 Single Signature

In this scheme only a single key-pair controls the identifier at any time. A multi-signature scheme is provided later.

4.1.1 Inception Event

Given a controller with identifier labeled *C* the serialized key *inception* event message labeled C_{ε_0} is denoted as follows:

$$\varepsilon_0 = C_{\varepsilon_0} = \left\langle C, t_0, \texttt{icp}, C^0, C^1, M_0, \hat{W}_0 \right\rangle \sigma_{C^0}, \tag{4.1}$$

where the identifier *C* is uniquely bound to the key-pair $\left(C^0, c^0\right)$, with public key, C^0, which is controlled by the private key, c^0, $t_0 = 0$ is the unique monotonically increasing sequence number for this event , `icp` is the event *ilk* representing an inception event, the controlling signing key-pair is indicated by C^0, which is the public key from the key-pair $\left(C^0, c^0\right)$ that is controlled by the private key, c^0, the ensuing pre-rotated key-pair is indicated by C^1 which is the public key from the key-pair $\left(C^1, c^1\right)$ that is controlled by the private key, c^1, the tally, M_0, is the number of witnesses sufficient for the event to be deemed valid by a judge (validator), witness list $\hat{W}_0 = \left[W_0, \ldots, W_{N-1}\right]_0$ contains a list of the identifiers (public keys) of *N* designated witnesses, and σ_{C^0} is the digital signature of the contents of the brackets, $\langle\ \rangle$, made with private key c^0. The inception event demonstrates control over the identifier public key C via the signature σ_{C^0}. It declares that $\left(C^0, c^0\right)$ is the current controlling key-pair and that $\left(C^1, c^1\right)$ will be the ensuing controlling key pair after rotation.

This off-line inception event designates witnesses to provide an additional layer of trust. The tally M_0, indicates the number of confirming witness receipts from the witness list that a judge or validator must obtain before validating the event. The tally represents a quorum size for validation and may be a sufficient majority. Each non-faulty member of the set of designated witnesses will have a copy of the inception event and thereby as long as one witness remains no-faulty that witness' copy will enable detection by a validator of a later exploit that creates an alternative inception event.

As a clarifying example, with a tally of 2 out of a total of 3 witnesses the inception event may be denoted as follows:

$$\varepsilon_0 = C_{\varepsilon_0} = \left\langle C, t_0, \texttt{icp}, C^0, C^1, 2, \left[W_0, W_1, W_2\right] \right\rangle \sigma_{C^0}, \tag{4.2}$$

where C^0, C^1, W_0, W_1, and W_2 are public keys from their associated key-pairs.

A tuple of the field labels may be denoted as follows:

$$\left(id, sn, ilk, signer, ensuer, tally, witnesses\right) \tag{4.3}$$

These labels may be used as the field names in a JSON serialized version of the key inception event as follows:

```
{
  "id": "Xq5YqaL6L48pf0fu7IUhL0JRaU2_RxFP0AL43wYn148=",
  "sn": 0,
```

```
  "ilk": "icp",
  "signer": "Xq5YqaL6L48pf0fu7IUhL0JRaU2_RxFP0AL43wYn148=",
  "ensuer": "Qt27fThWoNZsa88VrTkep6H-4HA8tr54sHON1vWl6FE=",
  "tally": 2,
  "witnesses":
  [
    "VrTkep6H-Qt27fThWoNZsa884HA8tr54sHON1vWl6FE=",
    "HON1vWl6FEQt27fThWoNZsa88VrTkep6H-4HA8tr54s=",
    "ThWoNZsa88VrTkeQt27fp6H-4HA8tr54sHON1vWl6FE=",
  ]
}
\r\n\r\n
"AeYbsHot0pmdWAcgTo5sD8iAuSQAfnH5U6wiIGpVNJQQoYKBYrPPxAoIc1i5SHC
IDS8KFFgf8i0tDq8XGizaCg=="
```

4.1.2 Inception Event Receipt

A witness, W_i (or validator), may create an inception event receipt by signing with its associated key-pair. Its signature of the inception event may be denoted as follows:

$$.\sigma_{W_i}\left(\left\langle C_{\varepsilon_0}\right\rangle\right)=\sigma_{W_i}\left(\left\langle C,t_0,\mathtt{icp},C^0,C^1,M_0,\widehat{W}_0\right\rangle\sigma_{C^0}\right) \tag{4.4}$$

The associated event receipt may then be denoted using compact notation as follows:

$$\rho_{W_i}\left(C_{\varepsilon_0}\right)=\left\langle C_{\varepsilon_0}\right\rangle\sigma_{W_i}=\left\langle\left\langle C,t_0,\mathtt{icp},C^0,C^1,M_0,\widehat{W}_0\right\rangle\sigma_{C^0}\right\rangle\sigma_{W_i}. \tag{4.5}$$

All events share the same key event receipt message format. This is described in Section 5.3.

4.1.3 Chained Rotation Event

The l^{th} rotation event operation occurring as the k^{th} event in the sequence for a controller labeled *C* that is provided via a serialized key *rotation* event message labeled, C_{ε_k}, may be denoted as follows:

$$\varepsilon_0=C_{\varepsilon_k}=\left\langle C,t_k,\eta\left(\varepsilon_{k-1}\right),\mathtt{rot},C^{l+1},M_l,\widehat{X}_l,\widehat{Y}_l\right\rangle\sigma_{C^{l-1}}\sigma_{C^l} \tag{4.6}$$

where the identifier *C* is uniquely bound to the key-pair $\left(C^0,c^0\right)$, with public key, C^0, which is controlled by the private key, c^0, $t_k=k$ is the unique monotonically increasing sequence number for this event, $\eta\left(\varepsilon_{k-1}\right)$ is the digest of the preceding, $(k-1)^{th}$, `rot` is the event *ilk* representing a rotation event, C^{l+1} is the public key from the ensuing key-pair, $\left(C^{l+1},c^{l+1}\right)$ which is controlled by the private key, c^{l+1}, the l^{th} tally, M_l, is the number of witnesses sufficient for the event to be deemed valid by a judge (validator), the l^{th} exclude (omit) witness list $\widehat{X}_l=\left[X_0,\ldots,X_{O_l-1}\right]_l$ contains a list of the identifiers (public keys) of O_l designated witnesses that are to be removed from the list of witnesses, the l^{th} include (append) witness list $\widehat{Y}_l=\left[Y_0,\ldots,Y_{P_l-1}\right]_l$ contains a list of the identifiers (public keys) of P_l designated witnesses that are to be added to the list of witnesses, event, and the digital signatures $\sigma_{C^{l-1}}$ (erster) and σ_{C^l} (signer) made with private keys c^{l-1} and c^l from key-pairs $\left(C^{l-1},c^{l-1}\right)$ and $\left(C^l,c^l\right)$ sign the

contents of the brackets, $\langle\ \rangle$ .The inclusion of the erster signature chains the signing authority of successive rotation events which provides additional protection against forgery of key event rotation operations in a key event history as two keys must be exploited not only one. In this notation the key-pair $\left(C^l, c^l\right)$ is the l^{th} key-pair in the pre-rotated sequence of keys controlled by *C*. This means that the previous rotation event declared $\left(C^{l-1}, c^{l-1}\right)$ as the signing key-pair and $\left(C^l, c^l\right)$ as the ensuing key-pair but now this rotation event implicitly declares $\left(C^l, c^l\right)$as the current signing key-pair and explicitly declares $\left(C^{l+1}, c^{l+1}\right)$as the ensuing pre-rotated key-pair in the sequence. This makes $\left(C^{l-1}, c^{l-1}\right)$ the erstwhile (erster) signing key-pair or erster.

As a clarifying example, suppose the inception event is as follows:

$$\varepsilon_0 = C_{\varepsilon_0} = \left\langle C, t_0, \texttt{icp}, C^0, C^1, 2, \left[W_0, W_1, W_2\right]\right\rangle \sigma_{C^0}, \tag{4.7}$$

where C^0 and C^1 are the public keys of the originating and first pre-rotated key-pair, the incepting tally is 2 out of 3 witnesses, W_0 , W_1 , and W_2 are the public keys of the witnesses. Suppose also that the first rotation event is also the next event after the inception event. In this case $k = l = 1$. Further suppose that the new tally is 2 out of 3 witnesses with one pruned witness from and one grafted witness onto the incepting set of witnesses. This event may be denoted as follows:

$$C_{\varepsilon_1} = \left\langle C, t_1, \eta\left(\varepsilon_0\right), \texttt{rot}, C^2, 2, \left[X_0\right]_1, \left[Y_0\right]_1\right\rangle \sigma_{C^0} \sigma_{C^1}, \tag{4.8}$$

where C^2 is the public key of the ensuing pre-rotated key-pair, X_0 is the public key of the pruned witness and Y_0 is the public key of the grafted witness.

A tuple of the field labels may be denoted as follows:

$$\left(id, sn, digest, ilk, ensuer, tally, prune, graft\right) \tag{4.9}$$

These labels may be used as the field names in a JSON serialized version of the key rotation event message as follows:

```
{
  "id": "Xq5YqaL6L48pf0fu7IUhL0JRaU2_RxFP0AL43wYn148=",
  "sn": 1,
  "digest": "GEj6YfRWmGViKAesa08UkNWukUkPGsdFPPboBAsjRBw=",
  "ilk": "rot",
  "ensuer": "UhL0JRaU2_RxFP0AL43wYn148Xq5YqaL6L48pf0fu7I=",
  "tally": 2,
  "prune":
  [
    "VrTkep6H-Qt27fThWoNZsa884HA8tr54sHON1vWl6FE=",
  ],
  "graft":
  [
    "HA8tr54sHON1vWl6FEVrTkep6H-Qt27fThWoNZsa884=",
  ]
}
\r\n\r\n
"AeYbsHot0pmdWAcgTo5sD8iAuSQAfnH5U6wiIGpVNJQQoYKBYrPPxAoIc1i5SHC
```

```
IDS8KFFgf8i0tDq8XGizaCg=="
\r\n\r\n
"KFFgf8i0tDq8XGizaCgAeYbsHot0pmdWAcgTo5sD8iAuSQAfnH5U6wiIGpVNJQQ
oYKBYrPPxAoIc1i5SHCIDS8=="
```

4.1.4 Chained Rotation Event Receipt

A witness, W_i (or validator), may create an inception event receipt by signing with its associated key-pair. Its signature of the l^{th} rotation event operation occurring as the k^{th} event in the sequence may be denoted as follows:

$$\sigma_{W_i}\left(\left\langle C_{\varepsilon_k}\right\rangle\right), \tag{4.10}$$

which in longer form is as follows:

$$\sigma_{W_i}\left(\left\langle C, t_k, \eta\left(\varepsilon_{k-1}\right), \mathtt{rot}, C^{l+1}, M_l, \widehat{X}_l, \widehat{Y}_l\right\rangle \sigma_{C^{l-1}} \sigma_{C^l}\right). \tag{4.11}$$

The associated event receipt may then be denoted using compact notation as follows:

$$\rho_{W_i}\left(C_{\varepsilon_k}\right) = \left\langle C_{\varepsilon_k}\right\rangle \sigma_{W_i}, \tag{4.12}$$

which in longer form as follows:

$$\left\langle\left\langle C, t_k, \eta\left(\varepsilon_{k-1}\right), \mathtt{rot}, C^{l+1}, M_l, \widehat{X}_l, \widehat{Y}_l\right\rangle \sigma_{C^{l-1}} \sigma_{C^l}\right\rangle \sigma_{W_i}. \tag{4.13}$$

All events share the same key event receipt message format. This is described in Section 5.3.

4.1.5 Unchained Rotation Event

When unchained the order of events is provided solely by the sequence number. In this case, the l^{th} rotation event operation occurring as the k^{th} event in the sequence for a controller labeled *C* that is provided via a serialized key *rotation* event message labeled, C_{ε_k}, may be denoted as follows:

$$\varepsilon_k = C_{\varepsilon_k} = \left\langle C, t_k, \mathtt{rot}, C^{l+1}, M_l, \widehat{X}_l, \widehat{Y}_l\right\rangle \sigma_{C^{l-1}} \sigma_{C^l} \tag{4.14}$$

where the field definition are the same as provided in the definition of eq. 4.6. The associated rotation event receipt and message are modified appropriately with the content above.

4.2 Witness Rotations

In both the chained and unchained rotation events the set of designated witnesses may be changed. This may be thought of as rotating the witness list. This facility gives the controller the ability to not only rotate keys that may have become exposed but also witnesses that may be unresponsive or compromised. Both the members and the total number of designated witnesses may be changed by the rotation event. This allows the controller to prune faulty witnesses and graft on new ones.

The resultant set of designated witnesses may be denoted as follows:

$$\boldsymbol{W}_l = \left(\boldsymbol{W}_{l-1} - \boldsymbol{X}_l\right) \cap \boldsymbol{Y}_l \tag{4.15}$$

where $\boldsymbol{W}_l$ is the set of witnesses after the current, l^{th}, rotation, $\boldsymbol{W}_{l-1}$ is the set of witnesses after the previous, $(l-1)^{th}$, rotation, $\boldsymbol{X}_l$ is the set of newly excluded witnesses from $\boldsymbol{W}_{l-1}$, and $\boldsymbol{Y}_l$ is the set of newly included witness in $\boldsymbol{W}_l$. The following set value properties apply:

$$\boldsymbol{X}_l \subseteq \boldsymbol{W}_{l-1},\ \boldsymbol{Y}_1 \not\subset \boldsymbol{W}_{l-1}, \text{ and } \boldsymbol{X}_l \not\subset \boldsymbol{W}_l. \tag{4.16}$$

The total, N_l, of witnesses after the l^{th} rotation may be computed as follows:

$$N_l = N_{l-1} - O_l + P_l \tag{4.17}$$

where N_{l-1} is the total after the $(l-1)^{th}$ rotation, O_l is the number of witnesses newly removed by the l^{th} rotation, and P_l is the number of witnesses newly added by the by the l^{th} rotation. The the l^{th} tally, M_l must satisfy:

$$M_l \leq N_l. \tag{4.18}$$

Let $|\ |$ represent the cardinality (number of elements) of a set. Then we have the following:

$$|\boldsymbol{X}_l| = O_l,\ |\boldsymbol{Y}_l| = P_l, \text{ and } |\boldsymbol{W}_l| = N_l. \tag{4.19}$$

Witness designation adds a layer of trust. The tally M_l, indicates the number of confirming witness receipts from the witness list that a judge or validator must obtain before validating the event. The tally represents a quorum size for validation and may be a sufficient majority. Each non-faulty member of the set of designated witnesses will have a copy of the rotation event and thereby as long as one witness remains no-faulty that witness' copy will enable detection by a validator of a later exploit that creates an alternative version of that rotation event.

In addition, a judge or validator may also require confirmation via M_{l-1} witness receipts from the previous, $(l-1)^{th}$, set of witnesses. In this case there are two confirmation sets. For the first confirmation, let $\boldsymbol{U}_{l-1}$ be the set of confirming witness receipts but of the l^{th} rotation event. There must be at least M_{l-1} receipts from the $(l-1)^{th}$ (prior) witness list. Let $|\ |$ represent the cardinality of set. These requirements mean that,

$$\boldsymbol{U}_{l-1} \subseteq \boldsymbol{W}_{l-1}, \text{ and } |\boldsymbol{U}_{l-1}| \geq M_{l-1}. \tag{4.20}$$

Likewise for the second confirmation, let $\boldsymbol{U}_l$ be the set of confirming witness receipts of the l^{th} rotation event. There must be at least M_l receipts from the l^{th} (current) witness list. These requirements mean that,

$$\boldsymbol{U}_l \subseteq \boldsymbol{W}_l, \text{ and } |\boldsymbol{U}_l| \geq M_l. \tag{4.21}$$

The two confirmations together satisfy the following:

$$|\boldsymbol{U}_{l-1} \cup \boldsymbol{U}_l| \leq M_{l-1} + M_l \tag{4.22}$$

Requiring both confirmation sets means that an exploiter must not only exploit the prior private keys but also a quorum of the prior set of witnesses. It also means that an exploited alternative event may not avoid detection by merely replacing all the witnesses from prior designations with exploited witnesses. A judge or validator will not validate the rotation event unless there is quorum of confirmation receipts from both sets of witnesses.

4.3 Multiple Signature

In this section the single signature scheme is extended to a multiple signature scheme. Review Section 2.9 for a more in-depth definition of some of the associated terminology.

4.3.1 Inception Event

Given a controller with identifier labeled *C* the serialized multi-signature key *inception* event message labeled C_{ε_0} is denoted as follows:

$$\varepsilon_0 = C_{\varepsilon_0} = \left\langle C, t_0, \texttt{icp}, C^0, K_1, \widehat{C}_1, M_0, \widehat{W}_0 \right\rangle \sigma_{C^0}, \tag{4.23}$$

where the identifier *C* is uniquely bound to the key-pair $\left(C^0, c^0\right)$, with public key, C^0, which is controlled by the private key, c^0, $t_0 = 0$ is the unique monotonically increasing sequence number for this event , `icp` is the event *ilk* representing an inception event, the controlling signing key-pair is indicated by C^0, which is the public key from the key-pair $\left(C^0, c^0\right)$ that is controlled by the private key c^0, K_1, is the threshold number of required signatures out of L_1 signers after the ensuing rotation, $\widehat{C}_1 = \left[C^1, \ldots, C^{L_1}\right]_1$ is the list of ensuing public keys that form a set of L_1 potential signers for the ensuing rotation to a multi-signature, the tally, M_0, is the number of witnesses sufficient for the event to be deemed valid by a judge (validator), witness list $\widehat{W}_0 = \left[W_0, \ldots, W_{N-1}\right]_0$ contains a list of the identifiers (public keys) of *N* designated witnesses, and σ_{C^0} is the digital signature of the contents of the brackets, $\langle\ \rangle$, made with private key c^0. In general the inception may be considered as a special case of rotation, that is, the zeroth rotation that implies that $K_0 = L_0 = 1$. and $C = C^0$. The inception pre-rotates (declares) the subsequence $\widehat{C}_1 = \left[C^1, \ldots, C^{L_1}\right]_1$ to be the ensuing set of key-pairs. The inception event demonstrates control over the identifier public key C via the signature σ_{C^0} but is not multi-signature itself. Consequently, when using a multiple signature scheme the inception event must be immediately followed by a rotation event to activate the first set of multiple signatures declared in the inception event. This rotation will immediately turn control over to the *K* of *L* multi-signature scheme previously specified in the inception event. The inception event explicitly declares that $\left(C^0, c^0\right)$ is the current controlling key-pair and that K_1 of the L_1 signers in $\widehat{C}_1 = \left[C^1, \ldots, C^{L_1}\right]_1$ will be the ensuing controlling key-pairs after rotation (*K* of *L* multi-signature).

This off-line inception event also designates witnesses to provide an additional layer of trust. The tally M_0, indicates the number of confirming witness receipts from the witness list that a judge or validator must obtain before validating the event. The tally represents a quorum size for validation and may be a sufficient majority. Each non-faulty member of the set of designated witnesses will have a copy of the inception event and thereby as long as one witness remains non-faulty that witness' copy will enable detection by a validator of a later exploit that creates an alternative inception event.

As a clarifying example, suppose the ensuing multi-signature is 2 of 3, that is, $K_1 = 2$ and $L_1 = 3$. Suppose also that the tally is 2 out of a total of 3 witnesses. The corresponding inception event may be denoted as follows:

$$C_{\varepsilon_0} = \left\langle C, t_0, \texttt{icp}, C^0, 2, \left[C^1, C^2, C^3\right], 2, \left[W_0, W_1, W_2\right] \right\rangle \sigma_{C^0}, \tag{4.24}$$

where C^0, C^1, C^2, C^3, W_0, W_1, and W_2 are public keys from their associated key-pairs.

A tuple of the field labels may be denoted as follows:

$$\left(id, sn, ilk, signer, threshold, ensuers, tally, witnesses\right) \tag{4.25}$$

These labels may be used as the field names in a JSON serialized version of the key inception event as follows:

```
{
  "id": "Xq5YqaL6L48pf0fu7IUhL0JRaU2_RxFP0AL43wYn148=",
  "sn": 0,
  "ilk": "icp",
  "signer": "Xq5YqaL6L48pf0fu7IUhL0JRaU2_RxFP0AL43wYn148=",
  "threshold": 2,
  "ensuers":
  [
    "Qt27fThWoNZsa88VrTkep6H-4HA8tr54sHON1vWl6FE=",
    "VrTkep6H-4HA8tr54sHON1vWl6FEQt27fThWoNZsa88=",
    "HA8tr54sHON1Qt27fThWoNZsa88VrTkep6H-4vWl6FE="
  ],
  "tally": 2,
  "witnesses":
  [
    "VrTkep6H-Qt27fThWoNZsa884HA8tr54sHON1vWl6FE=",
    "HON1vWl6FEQt27fThWoNZsa88VrTkep6H-4HA8tr54s=",
    "ThWoNZsa88VrTkeQt27fp6H-4HA8tr54sHON1vWl6FE=",
  ]
}
\r\n\r\n
"AeYbsHot0pmdWAcgTo5sD8iAuSQAfnH5U6wiIGpVNJQQoYKBYrPPxAoIc1i5SHC
IDS8KFFgf8i0tDq8XGizaCg=="
```

4.3.2 Inception Event Receipt

A witness, W_i (or validator), may create an inception event receipt by signing with its associated key-pair. Its signature of the inception event may be denoted as follows:

$$\sigma_{W_i}\left(\left\langle C_{\varepsilon_0}\right\rangle\right) = \sigma_{W_i}\left(\left\langle C, t_0, \mathtt{icp}, C^0, K_1, \widehat{C}_1, M_0, \widehat{W}_0\right\rangle \sigma_{C^0}\right), \tag{4.26}$$

The associated event receipt may then be denoted as follows:

$$\rho_{W_i}\left(C_{\varepsilon_0}\right) = \left\langle C_{\varepsilon_0}\right\rangle \sigma_{W_i} = \left\langle \left\langle C, t_0, \mathtt{icp}, C^0, K_1, \widehat{C}_1, M_0, \widehat{W}_0\right\rangle \sigma_{C^0}\right\rangle \sigma_{W_i}. \tag{4.27}$$

All events share the same key event receipt message format. This is described in Section 5.3.

4.3.3 Chained Rotation Event

The rotation event format enables changes to both the threshold and number of signatories with each rotation. The l^{th} multi-signature key *rotation* event event operation occurring as the k^{th} event in the sequence for a controller labeled *C* that is provided via a serialized key *rotation* event message labeled, C_{ε_k}, may be denoted as follows:

$$\varepsilon_k = C_{\varepsilon_k} = \left\langle C, t_k, \eta\left(\varepsilon_{k-1}\right), \mathtt{rot}, K_{l+1}, \widehat{C}_{l+1}, M_l, \widehat{X}_l, \widehat{Y}_l, \widehat{e}_l, \widehat{s}_l\right\rangle \widehat{\sigma}_l, \tag{4.28}$$

where the identifier *C* is uniquely bound to the key-pair $\left(C^0, c^0\right)$, with public key, C^0, which is

controlled by the private key, c^0, $t_k = k$ is the unique monotonically increasing sequence number for this event , $\eta(\varepsilon_{k-1})$ is the digest of the preceding, $(k-1)^{th}$, event, `rot` is the event *ilk* representing a rotation event, K_{l+1}, is the threshold number of required signatures out of L_{l+1} signers after the ensuing rotation, $\widehat{C}_{l+1} = \left[C^{r_{l+1}}, \ldots, C^{r_{l+1}+L_{l+1}-1}\right]_{l+1}$ is the list of ensuing public keys that form a set of L_{l+1} potential signers for the ensuing rotation to a multi-signature, r_{l+1} is the starting index of the subsequence of controlling key-pairs for the $(l+1)^{th}$ rotation (see Section 2.9), the l^{th} tally, M_l, is the number of witnesses sufficient for events to be deemed valid by a judge (validator), the l^{th} exclude (omit) witness list $\widehat{X}_l = \left[X_0, \ldots, X_{O_l-1}\right]_l$ contains a list of the identifiers (public keys) of O_l designated witnesses that are to be removed from the list of witnesses, the l^{th} include (append) witness list $\widehat{Y}_l = \left[Y_0, \ldots, Y_{P_l-1}\right]_l$ contains a list of the identifiers (public keys) of P_l designated witnesses that are to be added to the list of witnesses, $\widehat{e}_l = \left[e_0, \ldots, e_{E_l-1}\right]_l$ is a list of length E_l of integer indices into the list, $\left[C^{r_{l-1}}, \ldots, C^{r_{l-1}+L_{l-1}-1}\right]$, of ersters (erstwhile signers) as indicated by an earlier rotation event, r_{l-1} is the starting index of the subsequence of controlling key-pairs for the $(l-1)^{th}$ rotation (see Section 2.9), $\widehat{s}_l = \left[s_0, \ldots, s_{S_l-1}\right]_l$ is a list of length S_l of integer indices into the list, $\left[C^{r_l}, \ldots, C^{r_l+L_l-1}\right]$, of signers for this the l^{th} rotation event but as declared in the $(l-1)^{th}$ rotation event, r_l is the starting index of the subsequence of controlling key-pairs for the l^{th} rotation (see Section 2.9), and $\widehat{\sigma}_l$ is the set of digital signatures that sign the contents of the brackets, $\langle\ \rangle$. The inclusion of the ersters signatures chains the signing authority of successive rotation events which provides additional protection against forgery of key event rotation operations in a key event history as two sets of keys must be exploited not only one. The number of signatures in the set $\widehat{\sigma}_l$ is equal to the sum of the lengths of the ersters and signers lists, that is, $E_l + S_l$. Each element of the ersters and signers lists produce one signature. The appropriate index into the sequence of controlling key-pairs is computed from the offset given by each element of associated ersters and signers lists added to the starting index of the respective subsequence of key-pairs. This may be denoted as follows:

$$\widehat{\sigma}_l = \sigma_{C^{e0+\eta_{l-1}}} \ldots \sigma_{C^{eE_l-1+\eta_{l-1}}} \sigma_{C^{s0+\eta}} \ldots \sigma_{C^{sS_l-1+\eta}} . \tag{4.29}$$

In this derivation, the key-pair $\left(C^j, c^j\right)$ is the j^{th} key-pair in a pre-rotated sequence of key-pairs controlled by *C*. Each rotation indexed by *l* consumes one or more of these key-pairs. Furthermore, the list of zero based integer indices of ersters $\widehat{e}_l = \left[e_0, \ldots, e_{E_l-1}\right]_l$ is of length E_l and satisfies $K_{l-1} \le E_l \le L_{l-1}$. Each index refers to a key-pair from the ordered list of public keys $\left[C^{r_{l-1}}, \ldots, C^{r_{l-1}+L_{l-1}-1}\right]$ taken as a subsequence of the set of all $\left(C^j, c^j\right)$. The indices may be used to generate a list of signing ersters with the notation $\left[C^{e_0+r_{l-1}}, \ldots, C^{e_{E_l-1}+r_{l-1}}\right]$ (see Section 2.9). From this list the associated E_l signatures $\sigma_{C^{e0+\eta_{l-1}}} \ldots \sigma_{C^{eE_l-1+\eta_{l-1}}}$ are produced. Likewise, the list of zero based integer indices of signers $\widehat{s}_l = \left[s_0, \ldots, s_{S_l-1}\right]_l$ is of length S_l and satisfies $K_l \le S_l \le L_l$. Each index refers to a key-pair from the list of public keys $\left[C^{r_l}, \ldots, C^{r_l+L_l-1}\right]$ taken as a subsequence of

the set of all $\left(C^j, c^j\right)$. The indices may be used to generate a list of signing signers with the notation $\left[C^{s_0+r_l}, \ldots, C^{s_{S_l-1}+r_l}\right]$ (see Section 2.9). From this list the associated S_l signatures are produce $\sigma_{C^{s0+\eta}} \ldots \sigma_{C^{sS_l-1+\eta}}$ d. Together there are $E_l + S_l$ signatures attached to the event. The combination comprises $\hat{\sigma}_l$ defined in eq. 3.18.

This rotation event declares that the previously declared set of key-pairs in the subsequence $\left[C^{r_l}, \ldots, C^{r_l+L_l-1}\right]$ as the new controlling set that replaces the set of key-pairs in the subsequence $\left[C^{r_{l-1}}, \ldots, C^{r_{l-1}+L_{l-1}-1}\right]$. Furthermore this rotation event declares that the set of key-pairs in the subsequence $\hat{C}_{l+1} = \left[C^{r_{l+1}}, \ldots, C^{r_{l+1}+L_{l+1}-1}\right]_{l+1}$ will be the ensuing (pre-rotated) set.

As a clarifying example, let the sequence of inception event plus 3 rotation be as follows:

$$\begin{aligned}
\varepsilon_0 &= \left\langle C, t_0, \texttt{icp}, C^0, 2, \left[C^1, C^2, C^3\right], 2, \left[W_0, W_1, W_2\right]\right\rangle \sigma_{C^0} \\
\varepsilon_1 &= \left\langle C, t_1, \eta\left(\varepsilon_0\right), \texttt{rot}, 2, \left[C^4, C^5, C^6\right], 2, \left[X_0\right]_1, \left[Y_0\right]_1, \left[0\right]_1, \left[0, 2\right]_1\right\rangle \sigma_{C^0} \sigma_{C^1} \sigma_{C^3} \\
\varepsilon_2 &= \left\langle C, t_2, \eta\left(\varepsilon_1\right), \texttt{rot}, 3, \left[C^7, C^8, C^9, C^{10}\right], 3, \left[\ \right]_2, \left[Y_0\right]_2, \left[1, 2\right]_2, \left[0, 1\right]_2\right\rangle \sigma_{C^2} \sigma_{C^3} \sigma_{C^4} \sigma_{C^5} \\
\varepsilon_3 &= \left\langle C, t_3, \eta\left(\varepsilon_2\right), \texttt{rot}, 2, \left[C^{11}, C^{12}, C^{13}\right], 3, \left[X_0\right]_3, \left[Y_0\right]_3, \left[0, 1, 2\right]_3, \left[1, 2, 3\right]_3\right\rangle \hat{\sigma}_3
\end{aligned} \quad .(4.30)$$

where $\hat{\sigma}_3 = \sigma_{C^4} \sigma_{C^5} \sigma_{C^6} \sigma_{C^8} \sigma_{C^9} \sigma_{C^{10}}$.

The inception event, ε_0, always implies that $K_0 = 1$, $L_0 = 1$, and $r_0 = 0$. From this we can compute $r_1 = 1$. The inception event also declares that $K_1 = 2$ and by the given length of the ensuers list that $L_1 = 3$. From this we can compute $r_2 = 4$. These declarations mean that the ensuing multi-signature scheme will be 2 of 3. The witness tally is 2 out of the 3 provided witness public keys, $\left[W_0, W_1, W_2\right]$. The signature is σ_{C^0}.

The first rotation event, ε_1, declares that $K_2 = 2$ and by the given length of the ensuers list that $L_2 = 3$. From this we can compute $r_3 = 7$. The tally is still 2. There is one pruned witness whose public key is X_0 and one grafted witness whose public key is Y_0 leaving a total of 3 witnesses. The ersters list in this first rotation event is of length $E_1 = 1$. Because $K_0 = 1$ and $L_0 = 1$, it may have only one entry. From its entry, $\left[e_0 = 0\right]_1$ we may compute the index $j = e_0 + r_0 = 0 + 0 = 0$ of the associated key-pair. This gives the derived list of ersters as $\left[C^0\right]$. This corresponds to the original key-pair for the identifier. The signers list in this rotation must have at least $K_1 = 2$ but may have up to $L_1 = 3$ entries. In this case it has $S_1 = 2$ entries, $\left[s_0 = 0, s_1 = 2\right]_1$. From these entries we may compute the indices of the signing key-pairs. These indices are $j = s_0 + r_1 = 0 + 1 = 1$ and $j = s_1 + r_1 = 2 + 1 = 3$. This gives the derived list of signers as $\left[C^1, C^3\right]$. The associated signatures are $\sigma_{C^0} \sigma_{C^1} \sigma_{C^3}$.

The second rotation event, ε_2, declares that $K_3 = 3$ and by the given length of the ensuers list that $L_3 = 4$. From this we can compute $r_4 = 11$. The tally is increased to 3 with one additional grafted witness whose public key is Y_0 leaving a total of 4 witnesses. The ersters list in this rotation must have at least $K_1 = 2$ but may have up to $L_1 = 3$ entries. In this case it has 2 entries. From its entries, $\left[e_0 = 1, e_1 = 2\right]_2$ and $r_1 = 1$ we may derive the list of ersters as $\left[C^2, C^3\right]$. The

signers list in this rotation must have at least $K_2 = 2$ but may have up to $L_2 = 3$ entries. In this case is has 2 entries. From its entries $\left[s_0 = 0, s_1 = 1\right]_2$ and $r_2 = 4$. we may derive the list of signers as $\left[C^4, C^5\right]$. The associated signatures are $\sigma_{C^2}\sigma_{C^3}\sigma_{C^4}\sigma_{C^5}$.

The third rotation event, ε_3, declares that $K_4 = 2$ and by the given length of the ensuers list that $L_4 = 3$. From this we can compute $r_5 = 14$. The tally is still 3. There is one pruned witness whose public key is X_0 and one grafted witness whose public key is Y_0 leaving a total of 4 witnesses. The ersters list in this rotation must have at least $K_2 = 2$ but may have up to $L_2 = 3$ entries. In this case it has 3 entries. From its entries, $\left[e_0 = 0, e_1 = 1, e_1 = 2\right]_3$ and $r_2 = 4$ we may derive the list of ersters as $\left[C^4, C^5, C^6\right]$. The signers list in this rotation must have at least $K_3 = 3$ but may have up to $L_3 = 4$ entries. In this case is has 3 entries. From its entries $\left[s_0 = 1, s_1 = 2, s_1 = 3\right]_3$ and $r_3 = 7$. we may derive the list of signers as $\left[C^8, C^9, C^{10}\right]$. The associated signatures are $\sigma_{C^4}\sigma_{C^5}\sigma_{C^6}\sigma_{C^8}\sigma_{C^9}\sigma_{C^{10}}$.

A tuple of the field labels may be denoted as follows:

$$\left(id, sn, digest, ilk, threshold, ensuers, tally, prune, graft, ersters, signers\right) \tag{4.31}$$

These labels may be used as the field names in a JSON serialized version of the key rotation event message, ε_1, as follows:

```
{
  "id": "Xq5YqaL6L48pf0fu7IUhL0JRaU2_RxFP0AL43wYn148=",
  "sn": 1,
  "digest": "GEj6YfRWmGViKAesa08UkNWukUkPGsdFPPboBAsjRBw=",
  "ilk": "rot",
  "threshold": 2,
  "ensuers":
  [
    "WoNZsa88VrTkep6HQt27fTh-4HA8tr54sHON1vWl6FE=",
    "8tr54sHON1vWVrTkep6H-4HAl6FEQt27fThWoNZsa88=",
    "VrTkep6HHA8tr54sHON1Qt27fThWoNZsa88-4vWl6FE="
  ],
  "tally": 2,
  "prune":
  [
    "VrTkep6H-Qt27fThWoNZsa884HA8tr54sHON1vWl6FE=",
  ],
  "graft":
  [
    "HA8tr54sHON1vWl6FEVrTkep6H-Qt27fThWoNZsa884=",
  ],
  "ersters": [0],
  "signers": [0,2]
}
\r\n\r\n
"AeYbsHot0pmdWAcgTo5sD8iAuSQAfnH5U6wiIGpVNJQQoYKBYrPPxAoIc1i5SHC
IDS8KFFgf8i0tDq8XGizaCg=="
\r\n\r\n
```

```
"KFFgf8i0tDq8XGizaCgAeYbsHot0pmdWAcgTo5sD8iAuSQAfnH5U6wiIGpVNJQQ
oYKBYrPPxAoIc1i5SHCIDS8=="
\r\n\r\n
"Hot0pmdWAcgTo5sKFFgf8i0tDq8XGizaCgAeYbsD8iAuSQAfnH5U6wiIGpVNJQQ
oYKBYrPPxAoIc1i5SHCIDS8=="
```

4.3.4 Chained Rotation Event Receipt

A witness, W_i (or validator), may create an inception event receipt by signing with its associated key-pair. Its signature of the l^{th} rotation event operation occurring as the k^{th} event in the sequence may be denoted as follows:

$$.\sigma_{W_i}\left(\left\langle C_{\varepsilon_k}\right\rangle\right)=\sigma_{W_i}\left(\left\langle C,t_k,\eta\left(\varepsilon_{k-1}\right),\mathtt{rot},K_{l+1},\widehat{C}_{l+1},M_l,\widehat{X}_l,\widehat{Y}_l,\widehat{e}_l,\widehat{s}_l\right\rangle\widehat{\sigma}_l\right) \tag{4.32}$$

The associated event receipt may then be denoted using compact notation as follows:

$$\rho_{W_i}\left(C_{\varepsilon_k}\right)=\left\langle C_{\varepsilon_k}\right\rangle\sigma_{W_i}=\left\langle\left\langle C,t_k,\eta\left(\varepsilon_{k-1}\right),\mathtt{rot},K_{l+1},\widehat{C}_{l+1},M_l,\widehat{X}_l,\widehat{Y}_l,\widehat{e}_l,\widehat{s}_l\right\rangle\widehat{\sigma}_l\right\rangle\sigma_{W_i}. \tag{4.33}$$

All events share the same key event receipt message format. This is described in Section 5.3.

4.3.5 Unchained Rotation Event

The rotation event format enables changes to both the threshold and number of signatories with each rotation. When unchained the order of events is provided solely by the sequence number. In this case, the l^{th} multi-signature key *rotation* event event operation occurring as the k^{th} event in the sequence for a controller labeled *C* that is provided via a serialized key *rotation* event message labeled, C_{ε_k}, may be denoted as follows:

$$\varepsilon_k=C_{\varepsilon_k}=\left\langle C,t_k,\mathtt{rot},K_{l+1},\left[C^{r_{l+1}},\ldots,C^{r_{l+1}+L_{l+1}-1}\right],M_l,\widehat{X}_l,\widehat{Y}_l,\widehat{e}_l,\widehat{s}_l\right\rangle\widehat{\sigma} \tag{4.34}$$

where the field definition are the same as provided in the definition of eq. 4.28. The associated rotation event receipt and message are modified appropriately with the content above.

5 Other Events and Messages

5.1 Generic Interaction Event

One purpose establishing signing keys for an identifier is to sign attestations by the controller associated with that identifier for purposes other than key management, such as transactions or other interactions. In this case, of interaction (non-key management) a non-rotation event is applicable. Because interactions are varied, instead of defining a specific event *ilk* for every interaction type, a generic *interaction* event is defined where the interaction specific details are encapsulated in the event payload. To clarify, an interaction event is an event associated with an interaction between a controller and validator that is not a key management event such as key rotation. The payload of the generic interaction event includes a data structure that provides the details of the specific interaction. The data payload is what is being attested to. The generic interaction event message is serialized version of the event and includes an attached signature by the controller. The attached signature is made with the private key from the current signing key-pair(s) as specified by the most recent rotation event in the rotation history.

5.1.1 Single Signature Chained

The chained generic interaction event occurring as the k^{th} event in the sequence for a controller labeled *C* and signed with signing key-pair specified by the l^{th} rotation event may be denoted as follows:

$$\varepsilon_k = C_{\varepsilon_k} = \langle C, t_k, \eta(\varepsilon_{k-1}), \texttt{itc}, \{data\} \rangle \sigma_{C^l} \tag{5.1}$$

where the identifier *C* is uniquely bound to the key-pair (C^0, c^0), with public key, C^0, which is controlled by the private key, c^0, $t_k = k$ is the unique monotonically increasing sequence number for this event , $\eta(\varepsilon_{k-1})$ is the digest of the preceding event $(k-1)^{th}$, `itc` is the event *ilk* representing a generic *interaction* event, $\{data\}$ is the payload data structure for the event, and the digital signature σ_{C^l} made with private key c^l from key-pair, (C^l, c^l), signs the contents of the brackets $\langle\ \rangle$. The key-pair (C^l, c^l) is the l^{th} key-pair in the pre-rotated sequence of keys controlled by *C*.

For example when $l = 2$ and $k = 4$ the interaction event may be denoted as follows:

$$\varepsilon_4 = C_{\varepsilon_4} = \langle C, t_4, \eta(\varepsilon_3), \texttt{itc}, \{data\} \rangle \sigma_{C^2} \tag{5.2}$$

A tuple of the labels for the fields in the event may be denoted as follows:

$$(id, sn, digest, ilk, data) \tag{5.3}$$

These labels may be used as the field names in a JSON serialized version of the generic interaction event message which may be expressed as follows:

```
{
  "cid": "Xq5YqaL6L48pf0fu7IUhL0JRaU2_RxFP0AL43wYn148=",
  "sn": 4,
  "digest": "GEj6YfRWmGViKAesa08UkNWukUkPGsdFPPboBAsjRBw=",
  "ilk": "itc",
  "data":
  {
     "first":"John",
     "last":"Smith"
  },
}
\r\n\r\n
"AeYbsHot0pmdWAcgTo5sD8iAuSQAfnH5U6wiIGpVNJQQoYKBYrPPxAoIc1i5SHC
IDS8KFFgf8i0tDq8XGizaCg=="
```

5.1.2 Single Signature Unchained

The non-backward-chained generic interaction event signed with l^{th} rotation event specified signing key-pair occurring as the k^{th} event in the sequence may be denoted as follows:

$$\varepsilon_k = C_{\varepsilon_k} = \langle C, t_k, \texttt{itc}, \{data\} \rangle \sigma_{C^l} \tag{5.4}$$

where the field definitions are the same as in the chained version above (see eq. 5.1).

5.1.3 Multiple Signature Chained

The chained generic interaction event occurring as the k^{th} event in the sequence for a controller labeled *C* and signed with thresholded multi-signatures specified by the l^{th} rotation event may be denoted as follows:

$$\varepsilon_k = C_{\varepsilon_k} = \left\langle C, t_k, \eta\left(\varepsilon_{k-1}\right), \texttt{itc}, \{data\}, \hat{s}_k \right\rangle \hat{\sigma}_k \tag{5.5}$$

where the identifier *C* is uniquely bound to the key-pair $\left(C^0, c^0\right)$, with public key, C^0, which is controlled by the private key, c^0, $t_k = k$ is the unique monotonically increasing sequence number for this event , $\eta\left(\varepsilon_{k-1}\right)$ is the digest of the preceding event $(k-1)^{th}$, `itc` is the event *ilk* representing a generic *interaction* event, $\{data\}$ is the payload data structure for the event, and $\hat{s}_k = \left[s_0, \ldots, s_{S_k - 1} \right]_k$ is a list of length S_l of integer indices into the list, $\left[C^{r_l}, \ldots, C^{r_l + L_l - 1} \right]$, of controlling key-pair signers declared in the $(l-1)^{th}$ rotation event and activated in the l^{th} rotation, r_l is the starting index of the subsequence of controlling key-pairs for the l^{th} rotation (see Section 2.9), and $\hat{\sigma}_k$ is the set of digital signatures that sign the contents of the brackets, $\langle\ \rangle$. The number of signatures in the set $\hat{\sigma}_k$ is equal to the length, S_k, of the signers list. Each element of the signers list produces one signature. The appropriate index into the sequence of controlling key-pairs is computed from the offset given by each element of associated signers list added to the starting index of the respective subsequence of key-pairs. This may be denoted as follows:

$$\hat{\sigma}_k = \sigma_{C^{s_0 + r_l}} \ldots \sigma_{C^{s s_{k-1} + r_l}} . \tag{5.6}$$

In this derivation, the key-pair $\left(C^j, c^j\right)$ is the j^{th} key-pair in a pre-rotated sequence of key-pairs controlled by *C*. Each rotation indexed by *l* consumes one or more of these key-pairs. Furthermore, the list of zero based integer indices of signers $\hat{s}_k = \left[s_0, \ldots, s_{S_k - 1} \right]_k$ is of length S_k and satisfies $K_l \le S_k \le L_l$ where K_l is the threshold and L_l is total number of of signers as activated by the l^{th} rotation. Each index refers to a key-pair from the list of public keys $\left[C^{r_l}, \ldots, C^{r_l + L_l - 1} \right]$ taken as a subsequence of the set of all $\left(C^j, c^j\right)$. The indices may be used to generate a list of signing signers with the notation (see Section 2.9). From this list the associated S_l signatures $\hat{\sigma}_k = \sigma_{C^{s_0 + r_l}} \ldots \sigma_{C^{s s_{k-1} + r_l}}$ are produced. There are S_k signatures attached to the event. The combination comprises $\hat{\sigma}_k$ defined in eq. 5.6.

For example consider the following set of events:

$$\begin{aligned}
&\varepsilon_0 = \left\langle C, t_0, \texttt{icp}, C^0, 2, \left[C^1, C^2, C^3 \right] \right\rangle \sigma_{C^0} \\
&\varepsilon_1 = \left\langle C, t_1, \eta\left(\varepsilon_0\right), \texttt{rot}, 2, \left[C^4, C^5, C^6 \right], [0]_1, [0,1]_1 \right\rangle \sigma_{C^0} \sigma_{C^1} \sigma_{C^2} \\
&\ldots \\
&\varepsilon_4 = C_{\varepsilon_4} = \left\langle C, t_4, \eta\left(\varepsilon_3\right), \texttt{itc}, \{data\}, [0,2]_4 \right\rangle \sigma_{C^1} \sigma_{C^3}
\end{aligned} \tag{5.7}$$

The inception event, ε_0, and subsequent rotation event, ε_1, $l = 1$, declare that the current signature scheme is 2 of 3 of the signer set $\left[C^1, C^2, C^3 \right]$. Any non-rotation event must be signed by at least 2 of the 3 signers. The interaction event, ε_4, $k = 4$ has two signers indexed by

$[0,2]_4$. Given that $r_l = r_1 = 1$ the set of signers may be computed as $[C^1, C^3]$ and the signatures $\sigma_{C^1}\sigma_{C^3}$ are attached.

A tuple of the labels for the fields in the multi-signature generic interaction event may be denoted as follows:

$$(id, sn, digest, ilk, data, signers) \tag{5.8}$$

These labels may be used as the field names in a JSON serialized version of the generic interaction event message which may be expressed as follows:

```
{
  "cid": "Xq5YqaL6L48pf0fu7IUhL0JRaU2_RxFP0AL43wYn148=",
  "sn": 4,
  "digest": "GEj6YfRWmGViKAesa08UkNWukUkPGsdFPPboBAsjRBw=",
  "ilk": "itc",
  "data":
  {
     "first":"John",
     "last":"Smith"
  },
  "signers": [0,2]
}
\r\n\r\n
"AeYbsHot0pmdWAcgTo5sD8iAuSQAfnH5U6wiIGpVNJQQoYKBYrPPxAoIc1i5SHC
IDS8KFFgf8i0tDq8XGizaCg=="
\r\n\r\n
"QQoYKBYrPPxAoIc1i5SHCAeYbsHot0pmdWAcgTo5sD8iAuSQAfnH5U6wiIGpVNJ
IDS8KFFgf8i0tDq8XGizaCg=="
```

5.1.4 Multiple Signature Unchained

The unchained generic interaction event occurring as the k^{th} event in the sequence for a controller labeled *C* and signed with thresholded multi-signatures specified by the l^{th} rotation event may be denoted as follows:

$$\varepsilon_k = C_{\varepsilon_k} = \langle C, t_k, \texttt{itc}, \{data\}, \hat{s}_k \rangle \hat{\sigma}_k \tag{5.9}$$

where the field definitions are the same as in the chained version above (see eq. 5.5).

5.2 Combined Interaction Rotation Event

The purpose of the generic interaction event described above is to enable interactions between a controller and validator that are not expressly key management events such as key rotation. These interaction events are signed with the current signing key. Often numerous interaction events will occur between key rotations. Although this exposes the signing key, the pre-rotated ensuing signing key is not exposed and may be used to recover via a rotation from exploit of the signing key . This approach trades off security for convenience for applications where exposure to exploit of an interaction event is an acceptable trade-off for the convenience of reusing signing keys. In more critical interactions, however, that trade-off may not be acceptable. When not acceptable then better protection may be warranted. One way to better protect an interaction event is to combine it with a rotation event. In this case, the combined interaction and rotation

event only uses the signing keys once, i.e. effectively making the signing keys one-time usage keys that rotate with each usage. Thus the the interaction signing keys are not exposed via repeated use.

Any of the rotation operation events may be extended to support combined interaction and rotation (one time use signing) by adding a payload data structure, $\{data\}$, to the rotation event. This does not change the semantics of verification but merely enables the rotation to also be part of an interaction or vice versa and protects and interaction with a simultaneous rotation. The $\{data\}$ payload of the combined interaction-rotation event provides the details of the interaction. The combined interaction-rotation event message is a serialized version of the event and includes attached signature(s) by the controller as appropriate for a rotation event. The various extended rotation event denotations are as follows:

$$\begin{aligned}
C_{\varepsilon_k} &= \left\langle C, t_k, \eta(\varepsilon_{k-1}), \mathtt{rot}, C^{l+1}, \{data\} \right\rangle \sigma_{C^{l-1}} \sigma_{C^l} \\
C_{\varepsilon_k} &= \left\langle C, t_k, \eta(\varepsilon_{k-1}), \mathtt{rot}, K_{l+1}, \left[C^{r_{l+1}}, \ldots, C^{r_{l+1}+L_{l+1}-1} \right], \{data\}, \hat{e}_l, \hat{s}_l \right\rangle \hat{\sigma}_l \\
C_{\varepsilon_k} &= \left\langle C, t_k, \eta(\varepsilon_{k-1}), \mathtt{rot}, C^{l+1}, M_l, \hat{X}_l, \hat{Y}_l, \{data\} \right\rangle \sigma_{C^{l-1}} \sigma_{C^l} \\
C_{\varepsilon_k} &= \left\langle C, t_k, \eta(\varepsilon_{k-1}), \mathtt{rot}, K_{l+1}, \left[C^{r_{l+1}}, \ldots, C^{r_{l+1}+L_{l+1}-1} \right], M_l, \hat{X}_l, \hat{Y}_l, \{data\}, \hat{e}_l, \hat{s}_l \right\rangle \hat{\sigma}
\end{aligned} \quad . \tag{5.10}$$

As a clarifying example, the multi-signature witnessed rotation is extended below to include the interaction data payload. a JSON serialized version of the combined interaction-rotation event message follows:

```
{
  "id": "Xq5YqaL6L48pf0fu7IUhL0JRaU2_RxFP0AL43wYn148=",
  "sn": 1,
  "digest": "GEj6YfRWmGViKAesa08UkNWukUkPGsdFPPboBAsjRBw=",
  "ilk": "rot",
  "threshold": 2,
  "ensuers":
  [
    "WoNZsa88VrTkep6HQt27fTh-4HA8tr54sHON1vWl6FE=",
    "8tr54sHON1vWVrTkep6H-4HAl6FEQt27fThWoNZsa88=",
    "VrTkep6HHA8tr54sHON1Qt27fThWoNZsa88-4vWl6FE="
  ],
  "tally": 2,
  "prune":
  [
    "VrTkep6H-Qt27fThWoNZsa884HA8tr54sHON1vWl6FE=",
  ],
  "graft":
  [
    "HA8tr54sHON1vWl6FEVrTkep6H-Qt27fThWoNZsa884=",
  ],
  "data":
  {
     "first":"John",
     "last":"Smith"
  },
  "ersters": [0],
```

```
  "signers": [0,2]
}
\r\n\r\n
"AeYbsHot0pmdWAcgTo5sD8iAuSQAfnH5U6wiIGpVNJQQoYKBYrPPxAoIc1i5SHC
IDS8KFFgf8i0tDq8XGizaCg=="
\r\n\r\n
"KFFgf8i0tDq8XGizaCgAeYbsHot0pmdWAcgTo5sD8iAuSQAfnH5U6wiIGpVNJQQ
oYKBYrPPxAoIc1i5SHCIDS8=="
\r\n\r\n
"Hot0pmdWAcgTo5sKFFgf8i0tDq8XGizaCgAeYbsD8iAuSQAfnH5U6wiIGpVNJQQ
oYKBYrPPxAoIc1i5SHCIDS8=="
```

5.3 Event Receipt Message

The structure of the event receipt message is the same for any event. The only difference is specifying who created the receipt. The two main cases are validator created and witness created receipts.

5.3.1 Validator Event Receipt Message

A validator, V, may create an event receipt by signing with its associated key-pair. Its signature of the key event may be denoted as follows:

$$\sigma_V\left(\left\langle C_{\varepsilon_k}\right\rangle\right). \tag{5.11}$$

where the specific event details are dependent on the event itself.

Because the controller has a copy of the event message, a compact receipt message created by the validator need not include a copy of the event message but might include only the identifier of the controller, the event sequence number, the identifier of the validator and the signature of the validator. This compact event receipt message may be denoted as follows:

$$\left\langle C, t_k, \texttt{rct}, V, \sigma_V\left(\left\langle C_{\varepsilon_k}\right\rangle\right)\right\rangle = \left\langle C, t_k, \texttt{rct}, V, \sigma_V\right\rangle \tag{5.12}$$

where C is the identifier of the controller, t_k is the unique sequence number of the key event, `rct` is the event *ilk* representing an event receipt, V is the identifier of the validator bound to key-pair $\left(V^0, v^0\right)$, and $\sigma_V\left(\left\langle C_{\varepsilon_k}\right\rangle\right)$ is the signature of the validator on the key event message labeled C_{ε_k}.

When $k = 1$ this event receipt message may be simply expressed in a JSON serialized version as follows:

```
{
  "cid": "Xq5YqaL6L48pf0fu7IUhL0JRaU2_RxFP0AL43wYn148=",
  "sn": 1,
  "ilk": "rct",
  "vid": "3syVH2woCpOvPF0SD9Z0bu_OxNe2ZgxKjTQ961LlMnA=",
  "sig":
"KFFgf8i0tDq8XGizaCgAeYbsHot0pmdWAcgTo5sD8iAuSQAfnH5U6wiIGpVNJQQ
oYKBYrPPxAoIc1i5SHCIDS8==",
}
```

5.3.2 Witness Event Receipt Messages

A witness, W_i, may create an event receipt by signing with its associated key-pair. Its signature of the event may be denoted as follows:

$$\sigma_{W_i}\left(\left\langle C_{\varepsilon_k}\right\rangle\right), \tag{5.13}$$

where the specific event details are dependent on the event itself.

Because the controller has a copy of the event message, a compact receipt message created by the witness need not include a copy of the event message but may only include the identifier of the controller, the event sequence number, the event *ilk* of `rct`, the identifier of the witness and the signature of the witness. This compact event receipt message may be denoted as follows:

$$\left\langle C, t_k, \texttt{rct}, W_i, \sigma_{W_i}\left(\left\langle C_{\varepsilon_k}\right\rangle\right)\right\rangle = \left\langle C, t_k, \texttt{rct}, W_i, \sigma_{W_i}\right\rangle \tag{5.14}$$

where C is the identifier of the controller, t_0 is the unique sequence number of the event, `rct` is the event *ilk* representing an event receipt, W_i is the identifier of the witness bound to key-pair $\left(W_i^0, w_i^0\right)$, and $\sigma_{W_i}\left(\left\langle C_{\varepsilon_0}\right\rangle\right)$ is the signature of the witness on the key event message labeled C_{ε_0}.

When $k = 1$ this event receipt message may be simply expressed in a JSON serialized version as follows:

```
{
  "cid": "Xq5YqaL6L48pf0fu7IUhL0JRaU2_RxFP0AL43wYn148=",
  "sn": 1,
  "ilk": "rct",
  "wid": "3syVH2woCpOvPF0SD9Z0bu_OxNe2ZgxKjTQ961LlMnA=",
  "sig":
"KFFgf8i0tDq8XGizaCgAeYbsHot0pmdWAcgTo5sD8iAuSQAfnH5U6wiIGpVNJQQ
oYKBYrPPxAoIc1i5SHCIDS8==",
}
```

6 Example Sequences

These examples illustrate the sequencing of key management events interspersed with generic interaction events and how that affects the indices of the pre-rotated key-pairs. In general the best practice is to immediately follow the inception event with a rotation. This means that the original controlling key pair will have the least possible exposure because it is only used to sign the inception event and first rotation. This makes it difficult for any exploiter to forge an alternate key event history starting at the inception. Rotation events require signatures of the current and previous signing keys. This mean an exploit of two successive keys is required after the inception event.

6.1 Online Protocol Without Witnesses

6.1.1 Single Signature Chained

$$
\begin{aligned}
\varepsilon_0 &= \left\langle C, t_0, \texttt{icp}, C^0, C^1 \right\rangle \sigma_{C^0} \\
\varepsilon_1 &= \left\langle C, t_1, \eta(\varepsilon_0), \texttt{rot}, C^2 \right\rangle \sigma_{C^0} \sigma_{C^1} \\
\varepsilon_2 &= \left\langle C, t_2\ \eta(\varepsilon_1), \texttt{itc}, data \right\rangle \sigma_{C^1} \\
\varepsilon_3 &= \left\langle C, t_3, \eta(\varepsilon_2), \texttt{rot}, C^3 \right\rangle \sigma_{C^1} \sigma_{C^2} \\
\varepsilon_4 &= \left\langle C, t_4, \eta(\varepsilon_3), \texttt{itc}, data \right\rangle \sigma_{C^2} \\
\varepsilon_5 &= \left\langle C, t_5, \eta(\varepsilon_4), \texttt{itc}, data \right\rangle \sigma_{C^2}
\end{aligned}
\quad . \tag{6.1}
$$

6.1.2 Multiple Signature Chained

$$
\begin{aligned}
\varepsilon_0 &= \left\langle C, t_0, \texttt{icp}, C^0, 2, \left[C^1, C^2, C^3 \right] \right\rangle \sigma_{C^0} \\
\varepsilon_1 &= \left\langle C, t_1, \eta(\varepsilon_0), \texttt{rot}, 2, \left[C^4, C^5, C^6 \right], [0], [0,2] \right\rangle \sigma_{C^0} \sigma_{C^1} \sigma_{C^3} \\
\varepsilon_2 &= \left\langle C, t_2, \eta(\varepsilon_1), \texttt{itc}, data, [0,2]_1 \right\rangle \sigma_{C^1} \sigma_{C^3} \\
\varepsilon_3 &= \left\langle C, t_3, \eta(\varepsilon_2), \texttt{rot}, 2, \left[C^7, C^8, C^9 \right], [1,2], [0,1] \right\rangle \sigma_{C^2} \sigma_{C^3} \sigma_{C^4} \sigma_{C^5} \\
\varepsilon_4 &= \left\langle C, t_4, \eta(\varepsilon_3), \texttt{itc}, data, [0,2]_4 \right\rangle \sigma_{C^4} \sigma_{C^6} \\
\varepsilon_5 &= \left\langle C, t_5, \eta(\varepsilon_4), \texttt{itc}, data, [1,2]_4 \right\rangle \sigma_{C^5} \sigma_{C^6}
\end{aligned}
\quad . \tag{6.2}
$$

6.2 Offline Protocol With Witnesses

6.2.1 Single Signature Chained

$$
\begin{aligned}
\varepsilon_0 &= \left\langle C, t_0, \texttt{icp}, C^0, C^1, 2, \left[W_0, W_1, W_2 \right] \right\rangle \sigma_{C^0} \\
\varepsilon_1 &= \left\langle C, t_1, \eta(\varepsilon_0), \texttt{rot}, C^2, 2, \left[X_0 \right]_1, \left[Y_0 \right]_1 \right\rangle \sigma_{C^0} \sigma_{C^1} \\
\varepsilon_2 &= \left\langle C, t_2\ \eta(\varepsilon_1), \texttt{itc}, data \right\rangle \sigma_{C^1} \\
\varepsilon_3 &= \left\langle C, t_3, \eta(\varepsilon_2), \texttt{rot}, C^3, 3, [\ ]_2, \left[Y_0 \right]_2 \right\rangle \sigma_{C^1} \sigma_{C^2} \\
\varepsilon_4 &= \left\langle C, t_4, \eta(\varepsilon_3), \texttt{itc}, data \right\rangle \sigma_{C^2} \\
\varepsilon_5 &= \left\langle C, t_5, \eta(\varepsilon_4), \texttt{itc}, data \right\rangle \sigma_{C^2}
\end{aligned}
\quad . \tag{6.3}
$$

6.2.2 Multiple Signature Chained

$$
\begin{aligned}
\varepsilon_0 &= \left\langle C, t_0, \texttt{icp}, C^0, 2, \left[C^1, C^2, C^3\right], 2, \left[W_0, W_1, W_2\right]\right\rangle \sigma_{C^0} \\
\varepsilon_1 &= \left\langle C, t_1, \eta(\varepsilon_0), \texttt{rot}, 2, \left[C^4, C^5, C^6\right], 2, \left[X_0\right]_1, \left[Y_0\right]_1, \left[0\right]_1, \left[0,2\right]_1\right\rangle \sigma_{C^0}\sigma_{C^1}\sigma_{C^3} \\
\varepsilon_2 &= \left\langle C, t_2, \eta(\varepsilon_1), \texttt{itc}, data, \left[0,2\right]_1\right\rangle \sigma_{C^1}\sigma_{C^3} \\
\varepsilon_3 &= \left\langle C, t_3, \eta(\varepsilon_2), \texttt{rot}, 2, \left[C^7, C^8, C^9\right], 3, \left[\ \right]_2, \left[Y_0\right]_2, \left[1,2\right]_2, \left[0,1\right]_2\right\rangle \sigma_{C^2}\sigma_{C^3}\sigma_{C^4}\sigma_{C^5} \\
\varepsilon_4 &= \left\langle C, t_4, \eta(\varepsilon_3), \texttt{itc}, data, \left[0,2\right]_4\right\rangle \sigma_{C^4}\sigma_{C^6} \\
\varepsilon_5 &= \left\langle C, t_5, \eta(\varepsilon_4), \texttt{itc}, data, \left[1,2\right]_4\right\rangle \sigma_{C^5}\sigma_{C^6}
\end{aligned}
\quad . \tag{6.4}
$$

7 DELEGATION

In some applications, one important interaction for a controller may be to delegate its signing authority. One approach is to delegate the controller's authority to another identifier (authorized delegation). In this way, delegation may authorize some other identifier and its associated controlling keys as authoritative for the delegating controller. Likewise other delegative interactions may be possible. When the delegation interaction is a combined rotation-interaction event it provides protection to the delegation because the controller keys are only used once. Given the importance of some delegation operations this may be the preferred approach. A common use case would be to delegate authority to a sequence of signing keys distinct from the keys used for the controller's identifier through delegated inception and rotation events. This is a type of hierarchical key management. This approach may be applied recursively to enable multiple levels of delegation thereby creating a chained graph of key event streams. With this approach any number of layers or levels of management keys may be supported via composition of delegation events. Moreover, in addition to inception and rotation, delegation may be used to manage other types of authorizations.

Delegated authorizations are of primary importance for a general DKMI. The KERI design approach is to build composable primitives instead of custom functionality that is so typical of other DKMI approaches. We believe that the simpler but composable design of the KERI core is the most appropriate primitive. Consequently when applied recursively, delegation may be used to compose arbitrarily complex graphs of hierarchical (delegative) key management event streams. This is a most powerful capability that may provide essential building blocks for a generic universal DKMI that is compatible with the demands of event streaming applications.

7.1 Delegated Signing

The section defines delegative inception and rotation operations for managing a different set of signing keys. This is the simple case where only one level of delegation is supported (not recursive). A recursive delegation definition is beyond the scope of this work. Herein the controlling keys are merely used to manage (incept and rotate) other identified key event streams that provide signing keys. In order to limit exposure, the controlling keys may not be repurposed as signing keys. To clarify, in this approach, a master controller (identifier) key event stream may delegate signing authority to one or more slave identifier key event streams thereby forming a chained tree of key event streams. The associated delegative version of the KERI core state verification engine must also lookup and verify the delegator's key event log in order to confirm the delegator's control authority. In this sense, the delegate's event stream is linked to the delegator's event stream. A digest of the delegator's delegating event may be included in the delegate's inception/rotation event to chain the two streams together. Multiple delegates from the same del-

egator would form a tree. The following diagram shows a delegating (master) identifier with its associated sequence of controlling events and several delegate (slave) identifier with their sequence of controlled events.

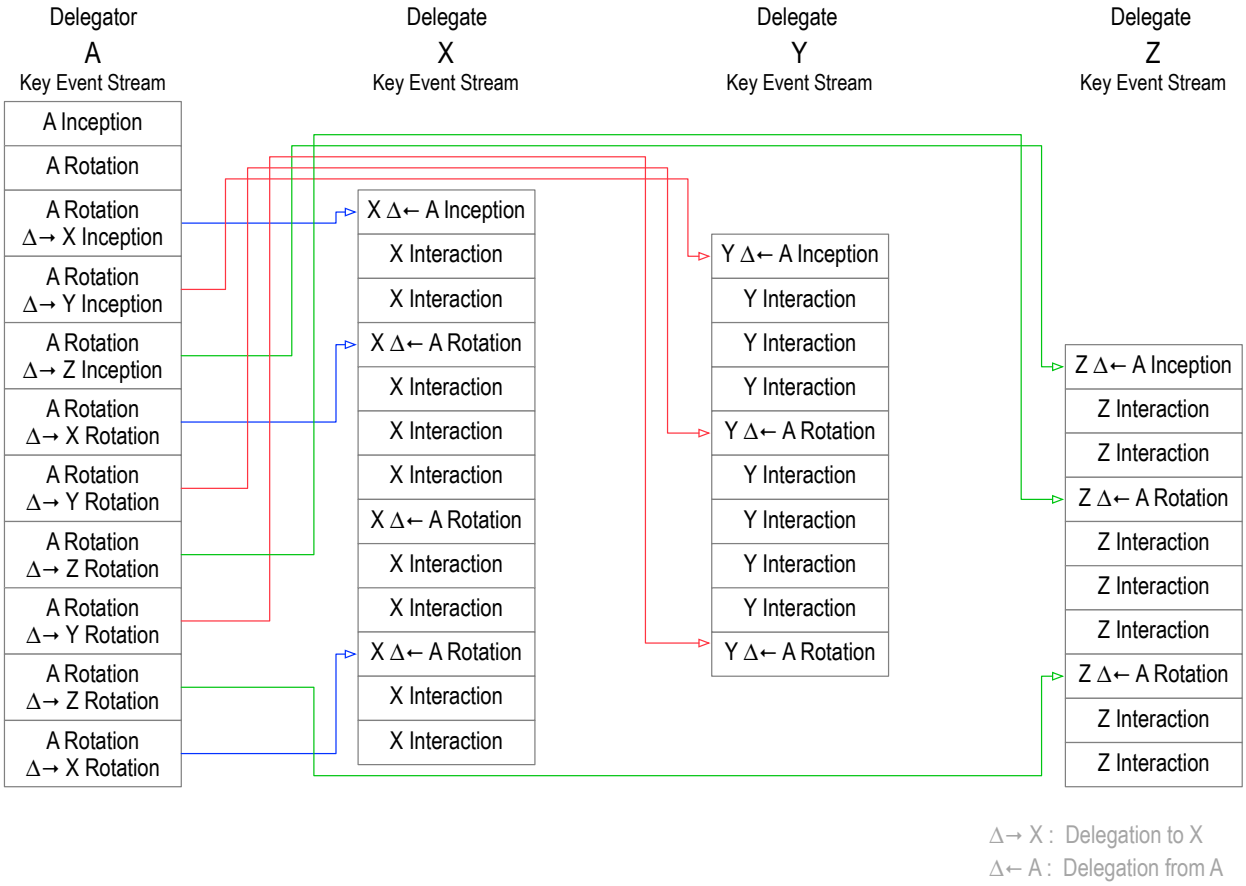

Figure 7.1. KERI Multiple Delegated Signing Key Event Streams

One advantage of this approach is that each delegated key event stream is a dedicated chain of events with a dedicated sequence numbering set and dedicated digest chain instead of an interleaved sequence number and digest chain. This makes it easier to manage each stream independently of the others. This makes it more scalable as well and easier to adapt to different key support infrastructures for each delegated signing key event stream. The delegating controller has one key-pair sequence this is represented by the associated public keys as follows:

$$C^j \Big|_{j=0,1,\ldots}, \tag{7.1}$$

such as $C^0, C^1, C^2, \ldots$.

Each controlled delegate also has a key-pair sequence that is represented by the associated public keys as follows:

$$D^j \Big|_{j=0,1,\ldots}, \tag{7.2}$$

such as $D^0, D^1, D^2, \ldots$. Without loss of generality the event messages for a pair of delegator and delegate may be extended to any number of delegates for the same delegator.

7.2 Delegated Inception Event

A delegated event stream is created by a delegated inception message that may be denoted as follows:

$$\varepsilon_0 = D_{\varepsilon_0} = \left\langle D, t_0, \texttt{dip}, \hat{\Delta}_k^C, D^0, perms, K_0^D, \hat{D}_0^D, M_0^D, \hat{W}_0^D \right\rangle \sigma_{D^0}, \tag{7.3}$$

where the identifier D is uniquely bound to the key-pair $\left(D^0, d^0\right)$, with public key, D^0, which is controlled by the private key, d^0, t_0 is the unique monotonically increasing sequence number for this event , `dip` is the event *ilk* representing a delegated inception event, $\hat{\Delta}_k^C = \left\{C, t_k^C, \eta\left(\varepsilon_k^C\right)\right\}$ is the delegation structure from the delegator, within this delegation structure, C is the identifier of the controlling delegator and controlled by key pair $\left(C^0, c^0\right)$, t_k^C is the unique monotonically increasing sequence number of the delegating event in its event sequence and $\eta\left(\varepsilon_k^C\right)$ is the digest of the delegating event, D^0 is the originating public key of the delegate, *perms* is a field that when not empty may hold a data structure of specific permissions for the delegation but when empty indicates the default permissions, K_0^D, is the threshold number of required signatures out of L_0^D signers, $\hat{D}_0^D = \left[D^1, \ldots, D^{L_0^D} \right]_0$ is the list of signing public keys that form a set of L_0^D signers for any subsequent interaction events, indeed these represent the authorized delegated signing key pairs, the tally, M_0^D is the number of witnesses sufficient for the event to be deemed valid by a judge (validator), the witness list contains $\hat{W}_0^D = \left[W_0^D, \ldots, W_{N^D-1}^D \right]_0$ a list of the identifiers (public keys) of N_0^D designated witnesses, and σ_{D^0} is the digital signature of the contents of the brackets, $\langle\ \rangle$, made with private key d^0. The signature σ_{D^0} demonstrates control over the identifier D. This prevents malicious entities from forging delegated inception events for an identifier not under their control. Verification of the delegation is performed by verifying the delegating event message from the controlling delegator's event stream.

An example JSON serialized version of the delegated inception event as follows:

```
{
  "id": "UhL0JRaU2_RxFP0AL4Xq5YqaL6L48pf0fu7I3wYn148=",
  "sn": 0,
  "ilk": "dip",
  "delegator":
  {
    "id": "Xq5YqaL6L48pf0fu7IUhL0JRaU2_RxFP0AL43wYn148=",
    "sn": "2",
    "digest": "GViKAesa08UkNWukGEj6YfRWmUkPGsdFPPboBAsjRBw=",
  },
  "signer": "UhL0JRaU2_RxFP0AL4Xq5YqaL6L48pf0fu7I3wYn148=",
  "perms": {},
  "threshold": 2,
  "signers":
  [
    "Qt27fThWoNZsa88VrTkep6H-4HA8tr54sHON1vWl6FE=",
```

```
    "VrTkep6H-4HA8tr54sHON1vWl6FEQt27fThWoNZsa88=",
    "HA8tr54sHON1Qt27fThWoNZsa88VrTkep6H-4vWl6FE="
  ],
  "tally": 2,
  "witnesses":
  [
    "VrTkep6H-Qt27fThWoNZsa884HA8tr54sHON1vWl6FE=",
    "HON1vWl6FEQt27fThWoNZsa88VrTkep6H-4HA8tr54s=",
    "ThWoNZsa88VrTkeQt27fp6H-4HA8tr54sHON1vWl6FE=",
  ]
}
\r\n\r\n
"AeYbsHot0pmdWAcgTo5sD8iAuSQAfnH5U6wiIGpVNJQQoYKBYrPPxAoIc1i5SHC
IDS8KFFgf8i0tDq8XGizaCg=="
```

7.3 Rotation Event Delegating Inception

A delegated event inception is verified against a rotation message with the associated delegation structure in the delegator's event stream. The rotation message that authorizes a delegated inception may be denoted as follows:

$$\varepsilon_k = \left\langle C, t_k, \eta(\varepsilon_{k-1}), \texttt{rot}, K_{l+1}, \widehat{C}_{l+1}, M_l, \widehat{X}_l, \widehat{Y}_l, \widehat{\Delta}_0^D, \widehat{e}_l, \widehat{s}_l \right\rangle \widehat{\sigma}_l, \tag{7.4}$$

where

$$\widehat{\Delta}_0^D = \left\{ D, \texttt{dip}, D^0, perms, K_0^D, \widehat{D}_0^D, M_0^D, \widehat{W}_0^D \right\} \tag{7.5}$$

is the delegation structure and the other fields are as defined in eq. 4.28 eq. 5.10, and eq. 9.8. The main difference is the $\{data\}$ field has been set to the delegation structure $\widehat{\Delta}_0^D$, i.e. $\{data\} = \widehat{\Delta}_0^D$. The contents of the delegation structure are as follows: D is the unique identifier of the delegate controlled by key-pair $\left(D^0, d^0\right)$, `dip` is the value of the delegated event *ilk*, D^0 is the associated public key, *perms* is a data structure that when not empty provides specific permission to the delegate and when empty indicates default permissions, K_0^D is the threshold number of a total of L_0^D authorized delegated signing key-pairs, $\widehat{D}_0^D = \left[D^1, \ldots, D^{L_0^D} \right]_0$ is a list of the public keys for the L_0^D associated delegated signing key-pairs, M_0^D is the tally of number of N_0^D delegated witnesses required for validation, and $\widehat{W}_0^D = \left[W_0^D, \ldots, W_{N^D-1}^D \right]_0$ is the list of delegated witnesses. The contents of the delegation structure are used to create the delegated inception event (see 7.1.2). The signatures, $\widehat{\sigma}_l = \sigma_{C^{e0+\eta-1}} \ldots \sigma_{C^{eE_l-1+\eta-1}} \sigma_{C^{s0+\eta}} \ldots \sigma_{C^{sS_l-1+\eta}}$, attached to this rotation event authorize the delegation.

An example JSON serialized version of the rotation event message delegating inception follows:

```
{
  "id": "Xq5YqaL6L48pf0fu7IUhL0JRaU2_RxFP0AL43wYn148=",
  "sn": 2,
  "digest": "GEj6YfRWmGViKAesa08UkNWukUkPGsdFPPboBAsjRBw=",
  "ilk": "rot",
```

```
  "threshold": 2,
  "ensuers":
  [
    "WoNZsa88VrTkep6HQt27fTh-4HA8tr54sHON1vWl6FE=",
    "8tr54sHON1vWVrTkep6H-4HAl6FEQt27fThWoNZsa88=",
    "VrTkep6HHA8tr54sHON1Qt27fThWoNZsa88-4vWl6FE="
  ],
  "tally": 2,
  "prune":
  [
    "VrTkep6H-Qt27fThWoNZsa884HA8tr54sHON1vWl6FE=",
  ],
  "graft":
  [
    "HA8tr54sHON1vWl6FEVrTkep6H-Qt27fThWoNZsa884=",
  ],
  "delegate":
  {
    "id": "UhL0JRaU2_RxFP0AL4Xq5YqaL6L48pf0fu7I3wYn148=",
    "ilk": "dip",
    "signer": "UhL0JRaU2_RxFP0AL4Xq5YqaL6L48pf0fu7I3wYn148=",
    "perms": {},
    "threshold": 2,
    "signers":
    [
      "Qt27fThWoNZsa88VrTkep6H-4HA8tr54sHON1vWl6FE=",
      "VrTkep6H-4HA8tr54sHON1vWl6FEQt27fThWoNZsa88=",
      "HA8tr54sHON1Qt27fThWoNZsa88VrTkep6H-4vWl6FE="
    ],
    "tally": 2,
    "witnesses":
    [
      "VrTkep6H-Qt27fThWoNZsa884HA8tr54sHON1vWl6FE=",
      "HON1vWl6FEQt27fThWoNZsa88VrTkep6H-4HA8tr54s=",
      "ThWoNZsa88VrTkeQt27fp6H-4HA8tr54sHON1vWl6FE=",
    ],
  },
  "ersters": [0],
  "signers": [0,2]
}
\r\n\r\n
"AeYbsHot0pmdWAcgTo5sD8iAuSQAfnH5U6wiIGpVNJQQoYKBYrPPxAoIc1i5SHC
IDS8KFFgf8i0tDq8XGizaCg=="
\r\n\r\n
"KFFgf8i0tDq8XGizaCgAeYbsHot0pmdWAcgTo5sD8iAuSQAfnH5U6wiIGpVNJQQ
oYKBYrPPxAoIc1i5SHCIDS8=="
\r\n\r\n
"Hot0pmdWAcgTo5sKFFgf8i0tDq8XGizaCgAeYbsD8iAuSQAfnH5U6wiIGpVNJQQ
oYKBYrPPxAoIc1i5SHCIDS8=="
```

7.4 Delegated Rotation Event

The signing keys in a delegated event stream are rotated by a delegated rotation message. The the e^{th} delegated rotation operation that occurs as the d^{th} event may be denoted as follows:

$$\varepsilon_d = D_{\varepsilon_d} = \left\langle D, t_d, \eta\left(\varepsilon_{d-1}\right), \texttt{drt}, \widehat{\Delta}_k^C, perms, K_e^D, \widehat{D}_e^D, M_e^D, \widehat{X}_e^D, \widehat{Y}_e^D \right\rangle \sigma_{D^0}, \tag{7.6}$$

where the identifier *D* is uniquely bound to the key-pair $\left(D^0, d^0\right)$, with public key, D^0, which is controlled by the private key, d^0, t_d is the unique monotonically increasing sequence number for this the d^{th} event, $\eta\left(\varepsilon_{d-1}\right)$ is the digest of the previous event in the delegates event sequence, `drt` is the event *ilk* representing a delegated rotation event, $\widehat{\Delta}_k^C = \left\{C, t_k^C, \eta\left(\varepsilon_k^C\right)\right\}$ is the delegation structure from the delegator, within this delegation structure, *C* is the identifier of the controlling delegator as controlled by key pair $\left(C^0, c^0\right)$, t_k^C is the unique monotonically increasing sequence number of the delegating event in its event sequence and $\eta\left(\varepsilon_k^C\right)$ is the digest of the delegating event, *perms* is a field that when not empty may hold a data structure of specific permissions for the delegation but when empty indicates the default permissions, K_e^D, is the threshold number of required signatures out of L_e^D signers for this the e^{th} rotation, $\widehat{D}_e^D = \left[D^{r_e^D}, \ldots, D^{r_e^D + L_e^D - 1} \right]_e$ is the list of signing public keys for this the e^{th} rotation that form a set of L_e^D signers for any subsequent interaction events, indeed these represent the authorized delegated signing key pairs, the tally, M_e^D is the number of witnesses sufficient out of a total N_e^D for events to be deemed valid by a judge (validator), the e^{th} exclude (omit) witness list $\widehat{X}_e^D = \left[X_0^D, \ldots, X_{O_e^D - 1}^D \right]_e$ contains a list of the identifiers (public keys) of O_e^D designated witnesses that are to be removed from the list of witnesses, the e^{th} include (append) witness list $\widehat{Y}_e^D = \left[Y_0, \ldots, Y_{P_e^D - 1} \right]_e$ contains a list of the identifiers (public keys) of P_e^D designated witnesses that are to be added to the list of witnesses, and σ_{D^0} is the digital signature of the contents of the brackets, $\langle\ \rangle$, made with private key d^0. The signature σ_{D^0} demonstrates control over the identifier *D*. This prevents malicious entities from forging delegated rotation events for an identifier not under their control. Verification of the delegation is performed by verifying the delegating event message from the controlling delegator's event stream.

An example JSON serialized version of the delegated rotation event message, follows:

```
{
  "id": "UhL0JRaU2_RxFP0AL4Xq5YqaL6L48pf0fu7I3wYn148=",
  "sn": 1,
  "digest": "PGsdFPPboGEj6YfRWmGViKAesa08UkNWukUkBAsjRBw=",
  "ilk": "drt",
  "delegator":
  {
    "id": "Xq5YqaL6L48pf0fu7IUhL0JRaU2_RxFP0AL43wYn148=",
    "sn": "4",
    "digest": "8UkNWukUkPGsdFPPboBAsjRBwGEj6YfRWmGViKAesa0=",
```

```
  },
  "perms": {},
  "threshold": 2,
  "signers":
  [
    "WoNZsa88VrTkep6HQt27fTh-4HA8tr54sHON1vWl6FE=",
    "8tr54sHON1vWVrTkep6H-4HAl6FEQt27fThWoNZsa88=",
    "VrTkep6HHA8tr54sHON1Qt27fThWoNZsa88-4vWl6FE="
  ],
  "tally": 2,
  "prune":
  [
    "VrTkep6H-Qt27fThWoNZsa884HA8tr54sHON1vWl6FE=",
  ],
  "graft":
  [
    "HA8tr54sHON1vWl6FEVrTkep6H-Qt27fThWoNZsa884=",
  ]
}
\r\n\r\n
"AeYbsHot0pmdWAcgTo5sD8iAuSQAfnH5U6wiIGpVNJQQoYKBYrPPxAoIc1i5SHC
IDS8KFFgf8i0tDq8XGizaCg=="
```

7.5 Rotation Event Delegating Rotation

A delegated event rotation is verified against a rotation message with the associated delegation structure in the delegator's event stream. The message conveying the e^{th} delegated rotation operation that occurs as the d^{th} event that authorizes a delegated rotation may be denoted as follows:

$$\varepsilon_k = \left\langle C, t_k, \eta\left(\varepsilon_{k-1}\right), \texttt{rot}, K_{l+1}, \widehat{C}_{l+1}, M_l, \widehat{X}_l, \widehat{Y}_l, \widehat{\Delta}_e^D, \widehat{e}_l, \widehat{s}_l \right\rangle \widehat{\sigma}_l, \tag{7.7}$$

where

$$\widehat{\Delta}_e^D = \left\{ D, \texttt{drt}, perms, K_e^D, \widehat{D}_e^D, M_e^D, \widehat{X}_e^D, \widehat{Y}_e^D \right\}, \tag{7.8}$$

is the delegation structure and the other fields are as defined in eq. 4.28 eq. 5.10, and eq. 9.8. The main difference is the $\{data\}$ field has been set to the delegation structure $\widehat{\Delta}_e^D$, i.e. $\{data\} = \widehat{\Delta}_e^D$. The contents of the delegation structure are as follows: *D* is the unique identifier of the delegate controlled by key-pair $\left(D^0, d^0\right)$, `dip` is the value of the delegated event *ilk*, *perms* is a data structure that when not empty provides specific permission to the delegate and when empty indicates default permissions, K_e^D is the threshold number of a total of L_0^D authorized delegated signing key-pairs, $\widehat{D}_e^D = \left[D^{r_e^D}, \ldots, D^{r_e^D + L_e^D - 1} \right]_l$ is a list of the public keys for the L_e^D associated delegated signing key-pairs, M_e^D is the tally of witnesses sufficient out of a total N_e^D for events to be deemed valid by a judge (validator), the e^{th} exclude (omit) witness list $\widehat{X}_e^D = \left[X_0^D, \ldots, X_{O_e^D - 1}^D \right]_e$ contains a list of the identifiers (public keys) of O_e^D designated witnesses

that are to be removed from the list of witnesses, the e^{th} include (append) witness list $\hat{Y}_e^D = \left[Y_0, \ldots, Y_{P_e^D - 1} \right]_e$ contains a list of the identifiers (public keys) of P_e^D designated witnesses that are to be added to the list of witnesses. The contents of the delegation structure are used to create the delegated inception event (see 7.1.2). The signatures, $\hat{\sigma}_l = \sigma_{C^{e0+\eta-1}} \ldots \sigma_{C^{eE_l-1+\eta-1}} \sigma_{C^{s0+\eta}} \ldots \sigma_{C^{sS_l-1+\eta}}$, attached to this rotation event authorize the delegation.

An example JSON serialized version of the rotation event message delegating rotation follows:

```
{
  "id": "Xq5YqaL6L48pf0fu7IUhL0JRaU2_RxFP0AL43wYn148=",
  "sn": 2,
  "digest": "GEj6YfRWmGViKAesa08UkNWukUkPGsdFPPboBAsjRBw=",
  "ilk": "rot",
  "threshold": 2,
  "ensuers":
  [
    "WoNZsa88VrTkep6HQt27fTh-4HA8tr54sHON1vWl6FE=",
    "8tr54sHON1vWVrTkep6H-4HAl6FEQt27fThWoNZsa88=",
    "VrTkep6HHA8tr54sHON1Qt27fThWoNZsa88-4vWl6FE="
  ],
  "tally": 2,
  "prune":
  [
    "VrTkep6H-Qt27fThWoNZsa884HA8tr54sHON1vWl6FE=",
  ],
  "graft":
  [
    "HA8tr54sHON1vWl6FEVrTkep6H-Qt27fThWoNZsa884=",
  ],
  "delegate":
  {
    "id": "UhL0JRaU2_RxFP0AL4Xq5YqaL6L48pf0fu7I3wYn148=",
    "ilk": "drt",
    "perms": {},
    "threshold": 2,
    "signers":
    [
      "WoNZsa88VrTkep6HQt27fTh-4HA8tr54sHON1vWl6FE=",
      "8tr54sHON1vWVrTkep6H-4HAl6FEQt27fThWoNZsa88=",
      "VrTkep6HHA8tr54sHON1Qt27fThWoNZsa88-4vWl6FE="
    ],
    "tally": 2,
    "prune":
    [
      "VrTkep6H-Qt27fThWoNZsa884HA8tr54sHON1vWl6FE=",
    ],
    "graft":
    [
      "HA8tr54sHON1vWl6FEVrTkep6H-Qt27fThWoNZsa884=",
    ]
```

```
  },
  "ersters": [0,2],
  "signers": [3,4]
}
\r\n\r\n
"AeYbsHot0pmdWAcgTo5sD8iAuSQAfnH5U6wiIGpVNJQQoYKBYrPPxAoIc1i5SHC
IDS8KFFgf8i0tDq8XGizaCg=="
\r\n\r\n
"KFFgf8i0tDq8XGizaCgAeYbsHot0pmdWAcgTo5sD8iAuSQAfnH5U6wiIGpVNJQQ
oYKBYrPPxAoIc1i5SHCIDS8=="
\r\n\r\n
"Hot0pmdWAcgTo5sKFFgf8i0tDq8XGizaCgAeYbsD8iAuSQAfnH5U6wiIGpVNJQQ
oYKBYrPPxAoIc1i5SHCIDS8=="
\r\n\r\n
"AeYbsHot0pmdWAcgTo5sD8iAuSQAfnH5U6wiIGpVNJQQoYKBYrPPxAoIc1i5SHC
IDS8KFFgf8i0tDq8XGizaCg=="
```

7.6 Delegate Interaction Event

A delegate receives authority to sign interaction event messages via the associated delegated inception and rotation events. The actual signed interaction event is not itself delegated but is produced solely by the delegate. The delegate interaction event occurring as the d^{th} event in the sequence for delegate labeled *D* and signed with thresholded multi-signatures specified by the e^{th} delegated rotation event may be denoted as follows:

$$\varepsilon_d = D_{\varepsilon_d} = \left\langle D, t_d, \eta\left(\varepsilon_{d-1}\right), \texttt{itc}, data, \hat{s}_d^D \right\rangle \hat{\sigma}_d \tag{7.9}$$

where the identifier *D* is uniquely bound to the key-pair $\left(D^0, d^0\right)$, with public key, D^0, which is controlled by the private key, d^0, t_d is the unique monotonically increasing sequence number for this the d^{th} event, $\eta\left(\varepsilon_{d-1}\right)$ is the digest of the previous event in the delegates event sequence, `itc` is the event *ilk* representing a generic *interaction* event, $\{data\}$ is the payload data structure for the event, and $\hat{s}_d^D = \left[s_0, \ldots, s_{S_d-1}\right]_d$ is a list of length S_d^D of integer indices into the list, $\hat{D}_e^D = \left[D^{r_e^D}, \ldots, D^{r_e^D + L_e^D - 1}\right]_e$, of controlling key-pair signers activated in the e^{th} rotation event, r_e is the starting index of the subsequence of controlling key-pairs for the e^{th} delegated rotation (see Section 2.9), and $\hat{\sigma}_d$ is the set of digital signatures that sign the contents of the brackets, $\langle\ \rangle$. The number of signatures in the set $\hat{\sigma}_d$ is equal to the length, , of the S_d^D signers list. Each element of the signers list produces one signature. The appropriate index into the sequence of controlling key-pairs is computed from the offset given by each element of associated signers list added to the starting index of the respective subsequence of key-pairs. This may be denoted as follows:

$$\hat{\sigma}_d = \sigma_{D^{s_0 + r_e^D}} \ldots \sigma_{D^{s_{S_d^D - 1} + r_e^D}}. \tag{7.10}$$

In this derivation, the key-pair $\left(D^j, d^j\right)$ is the j^{th} key-pair in a pre-rotated sequence of key-pairs controlled by *D*. Each rotation indexed by *e* consumes one or more of these key-pairs.

Furthermore, the list of zero based integer indices of signers $\hat{s}_d^D = \left[s_0, \ldots, s_{S_d^D - 1} \right]_d$ is of length S_d^D and satisfies $K_e^D \le S_d^D \le L_e^D$ where K_e^D is the threshold and L_e^D is total number of of signers as activated by the e^{th} rotation. Each index refers to a key-pair from the list of public keys $\hat{D}_e^D = \left[D^{r_e^D}, \ldots, D^{r_e^D + L_e^D - 1} \right]_e$ taken as a subsequence of the set of all $\left(D^j, d^j \right)$. The indices may be used to generate a list of signing signers with the notation (see Section 2.9). From this list the associated S_d^D signatures $\hat{\sigma}_d = \sigma_{D^{s_0 + r_e^D}} \ldots \sigma_{D^{s_{S_d^D - 1} + r_e^D}}$ are produced. There are S_d^D signatures attached to the event. The combination comprises $\hat{\sigma}_k$ defined in eq. 7.10.

An example JSON serialized version of the delegate interaction event message follows:

```
{
  "cid": "UhL0JRaU2_RxFP0AL4Xq5YqaL6L48pf0fu7I3wYn148=",
  "sn": 3,
  "digest": "GEj6YfRWmGViKAesa08UkNWukUkPGsdFPPboBAsjRBw=",
  "ilk": "itc",
  "data":
  {
     "first":"John",
     "last":"Smith"
  },
  "signers": [0,2]
}
\r\n\r\n
"AeYbsHot0pmdWAcgTo5sD8iAuSQAfnH5U6wiIGpVNJQQoYKBYrPPxAoIc1i5SHC
IDS8KFFgf8i0tDq8XGizaCg=="
\r\n\r\n
"QQoYKBYrPPxAoIc1i5SHCAeYbsHot0pmdWAcgTo5sD8iAuSQAfnH5U6wiIGpVNJ
IDS8KFFgf8i0tDq8XGizaCg=="
```

8 Weighted Threshold Multi-Signature Scheme

Any of the threshold multi-signature forms presented elsewhere in this work may be modified to support a fractional weighted threshold by changing the threshold number (integer) into a list of fractional values where each value is between 0 and 1. In this scheme, a valid signature set is any subset of signatures whose corresponding weights sum to 1 or greater. For example a threshold signature scheme for the following ordered list of signers:

$$\hat{C} = \left[C^1, C^2, C^3 \right], \tag{8.1}$$

where any 2 of 3 signatures is valid may be represented equivalently by the following ordered list of fractional weights,

$$\hat{K} = \left[\tfrac{1}{2}, \tfrac{1}{2}, \tfrac{1}{2} \right], \tag{8.2}$$

where any combination of two or more weights would sum to at least 1. To generalize let the l^{th} ordered set of controlling signers be denoted as follows:

$$\hat{C}_l = \left[C_l^1, \ldots, C_l^{L_l} \right]_l, \tag{8.3}$$

where $\widehat{C}_l$ represents the l^{th} list, L_l is the number of signers in the list, and each C_l^j in the list is the public key from a signing key-pair. The corresponding l^{th} ordered list of fractional weights (one-to-one for each signer) may be denoted as follows:

$$\widehat{K}_l = \left[U_l^1, \ldots, U_l^{L_1} \right]_l, \tag{8.4}$$

where $\widehat{K}_l$ represents the l^{th} list, L_l is the number of weights in the list, and each U_l^j in the list is the weight for the corresponding signer. Each weight satisfies:

$$0 < U_l^j \leq 1. \tag{8.5}$$

A subset of the full list of signatures may be attached to some event. Let it be the k^{th} event. This subset may be represented by a list of zero based indexes (offsets) into the l^{th} list of signers and weights given by eq. 8.3 and eq. 8.4 The indexed list may denoted as follows:

$$\widehat{s}_k^l = \left[s_0, \ldots, s_{S_k^l - 1} \right]_k^l, \tag{8.6}$$

where $\widehat{s}_k^l$ represents the ordered indexed list of offsets into the l^{th} list of signers attached to the k^{th} event, S_k^l is the number of attached signers, and each s_i is a zero based offset into both $\widehat{C}_l$ and $\widehat{K}_l$. A set of signatures is valid when the associated weights satisfy sum to greater than or equal 1, such as:

$$\overline{U}_l = \sum_{i=s_0}^{s_{S_k - 1}} U_l^i \geq 1, \tag{8.7}$$

where $\overline{U}_l$ represents the sum, *i* in the summation is assigned to successive values from the list of offsets, $\widehat{s}_k^l = \left[s_0, \ldots, s_{S_k^l - 1} \right]_k^l$, that is, $i \in \widehat{s}_k^l$, and U_l^i is the weight at the offset *i*. Care must be taken when using floating point representations to account for floating point rounding errors in the summation. One way to avoid this problem is to use rational fractional number computations. Some programming languages support explicit rational fractional calculations. For example the *fractions* module in Python.

In its simplest form, as the example above showed, this weighted threshold scheme may be made to act equivalently to the more simpler *K* of *L* threshold scheme by assigning equal weights as follows:

$$U_l^j = 1 \Big/ K_l, \tag{8.8}$$

where K_l is the threshold count of signatures in a conventional *K* of *L* threshold scheme. The real power of the weighted threshold scheme is expressed by unequal weights. This allows different combinations of signers to reach a valid signature set.

Suppose for example that the l^{th} weight list is as follows:

$$\widehat{K}_l = \left[\tfrac{1}{2}, \tfrac{1}{2}, \tfrac{1}{4}, \tfrac{1}{4}, \tfrac{1}{4}, \tfrac{1}{4} \right]_l. \tag{8.9}$$

In this case, a valid signature set occurs either when both the first two signers sign, or when any one of the first two sign and any two or more of the last four sign, or when all four of the last four sign. This allows different degrees of signing strength or signing authority to be assigned to signers as a way to reflect different degrees of trust in the signers. This allows the assignment of role based signing authority and hierarchical authority. Effectively, in the example above, the first two signers have the sam authority together as the last four have together. The first two

could be higher level managers and the last four lower level managers. Using delegation further extends the capability. Multiple distinct delegated sets of weighted multi-signature scheme provides extreme flexibility for managing signing authority arrangements.

As described above, simply replacing the integer valued K_l with the list valued $\widehat{K}_l$ in any of the aforementioned multiple signature capable event definitions may enable weighted multiple signatures. For example the inception event, $\varepsilon_0 = \left\langle C, t_0, \mathtt{icp}, C^0, K_1, \widehat{C}_1, M_0, \widehat{W}_0 \right\rangle \sigma_{C^0}$ becomes $\varepsilon_0 = \left\langle C, t_0, \mathtt{icp}, C^0, \widehat{K}_1, \widehat{C}_1, M_0, \widehat{W}_0 \right\rangle \sigma_{C^0}$ where $\widehat{K}_1$ is a weight list. Furthermore the delegated inception $D_{\varepsilon_0} = \left\langle D, t_0, \mathtt{dip}, \widehat{\Delta}_k^C, D^0, perms, K_0^D, \widehat{D}_0^D, M_0^D, \widehat{W}_0^D \right\rangle \sigma_{D^0}$ becomes $D_{\varepsilon_0} = \left\langle D, t_0, \mathtt{dip}, \widehat{\Delta}_k^C, D^0, perms, \widehat{K}_0^D, \widehat{D}_0^D, M_0^D, \widehat{W}_0^D \right\rangle \sigma_{D^0}$ where $\widehat{K}_0^D$ is a weight list.

9 IMPLEMENTATION

9.1 Universal Semantics and Syntax

One purpose of KERI is to provide minimally sufficient means for decentralized key management infrastructure (DKMI). This allows it to be used in the broadest range of potential applications. Given this broad application range another potential advantage of KERI is that it may enable a portable DKMI. Decentralized identifiers allow control by disparate entities, not a central administrator but are also interoperable amongst the entities, thus making the identifiers portable. Portable decentralized identifiers enable network effects among applications and tooling that use the identifiers. Likewise a portable DKMI extends network effect advantages to not just the identifiers but the key management infrastructure supporting the identifiers. Because DKMI requires user participation, the lack of interoperable portable DKMI means that despite portable decentralized identifiers, the user experience of client application and wallets that use those identifiers will not be very portable. KERI is a candidate for a portable interoperable DKMI. By defining its semantics and syntax to be sufficiently flexible to accommodate the vast majority of use cases, and employing best practices for security but in a scalable and performant manner, KERI has the potential to be a universal DKMI specification.

The following sections map the various options for KERI onto a single universal syntax with consistent conventions for semantics. This allows a single implementation of both servers and clients but that accommodates application specific tuning of the features.

In general the protocol variation that supports offline multiple signatures and witnesses with the interaction data payload is the most generic set of syntax and semantics. All the other variations are special cases of that variation. The differences lie in the event syntax and associated semantics. Specifically a given variation may not include one or more of the fields in the associated events and as a result has different semantics. However if we adopt the convention that missing fields have defined default values then all the syntax variations can be mapped to a single universal syntax expression for each event. This enables a single semantic implementation of the underlying verification and validation logic. The following sections describe these mappings.

9.1.1 Inception

9.1.1.1 Multiple Signature With Witnesses

The most general expression for the inception event is as follows:

$$\varepsilon_0 = \left\langle C, t_0, \mathtt{icp}, C^0, K_1, \widehat{C}_1, M_0, \widehat{W}_0 \right\rangle \sigma_{C^0}, \tag{9.1}$$

where,

$$\widehat{C}_1 = \left[C^1, \ldots, C^{L_1} \right]_1, \tag{9.2}$$

and

$$\widehat{W}_0 = \left[W_0, \ldots, W_{N-1} \right]_0. \tag{9.3}$$

This expression supports multiple-signatures with a threshold as well as multiple witnesses with a tally. This is described above in Section 4.3.1.

9.1.1.2 Single Signature With Witnesses

A subset of the general inception expression is to only use a single signature with witnesses. This is expressed as follows:

$$\varepsilon_0 = \left\langle C, t_0, \texttt{icp}, C^0, C^1, M_0, \widehat{W}_0 \right\rangle \sigma_{C^0}. \tag{9.4}$$

The expression above is described in Section 4.1.1. It is equivalent to a multi-signature event expression but where the threshold is 1 and only one ensuer is declared. Therefore an implementation may map an event with a missing threshold field and a single ensuer (ensuing public key) to an equivalent event with a threshold of 1 and an ensuer array with one element. This is denoted as follows:

$$\varepsilon_0 = \left\langle C, t_0, \texttt{icp}, C^0, 1, \left[C^1 \right], M_0, \widehat{W}_0 \right\rangle \sigma_{C^0}. \tag{9.5}$$

9.1.1.3 Single Signature Without Witnesses

A smaller subset of the general inception expression is to only use a single signature but without witnesses. This is expressed as follows:

$$\varepsilon_0 = \left\langle C, t_0, \texttt{icp}, C^0, C^1 \right\rangle \sigma_{C^0}. \tag{9.6}$$

The expression above is described in Section 3.1.1. It is equivalent to a witnessed event expression but where the tally is 0 and the witness list is empty. Furthermore, like the previous variation, the threshold is 1 and only one ensuer is declared. Therefore an implementation may map an event with missing tally, missing witness list fields, missing threshold field, and a single ensuer (ensuing public key) to an equivalent event with a tally of zero, empty witness list, a threshold of 1, and an ensuer array with one element. This is denoted as follows:

$$\varepsilon_0 = \left\langle C, t_0, \texttt{icp}, C^0, 1, \left[C^1 \right], 0, [\ ] \right\rangle \sigma_{C^0}.$$

9.1.1.4 Multiple Signature Without Witnesses

A different subset of the general inception expression is to only use multiple signatures but without witnesses. This is expressed as follows:

$$\varepsilon_0 = \left\langle C, t_0, \texttt{icp}, C^0, K_1, \left[C^1, \ldots, C^{L_1} \right] \right\rangle \sigma_{C^0}. \tag{9.7}$$

The expression above is described in Section 3.2.1. It is equivalent to a witnessed event expression but where the tally is 0 and the witness list is empty. Therefore an implementation may map an event with missing tally and missing witness list fields to an equivalent event with a tally of zero and empty witness list. This is denoted as follows:

$$\varepsilon_0 = \left\langle C, t_0, \texttt{icp}, C^0, K_1, \left[C^1, \ldots, C^{L_1} \right], 0, [\ ] \right\rangle \sigma_{C^0}.$$

9.1.2 Rotation

9.1.2.1 Multiple Signature With Witnesses

The most general expression for the extended rotation event is as follows:

$$\varepsilon_k = \left\langle C, t_k, \eta\left(\varepsilon_{k-1}\right), \mathtt{rot}, K_{l+1}, \widehat{C}_{l+1}, M_l, \widehat{X}_l, \widehat{Y}_l, \{data\}, \widehat{e}_l, \widehat{s}_l \right\rangle \widehat{\sigma}_l, \tag{9.8}$$

where

$$\widehat{C}_{l+1} = \left[C^{r_{l+1}}, \ldots, C^{r_{l+1}+L_{l+1}-1} \right]_{l+1} \tag{9.9}$$

$$\widehat{X}_l = \left[X_0, \ldots, X_{O_l-1} \right]_l, \tag{9.10}$$

$$\widehat{Y}_l = \left[Y_0, \ldots, Y_{P_l-1} \right]_l, \tag{9.11}$$

$\{data\}$ is interaction payload data structure, (9.12)

$$\widehat{e}_l = \left[e_0, \ldots, e_{E_l-1} \right]_l, \tag{9.13}$$

$$\widehat{s}_l = \left[s_0, \ldots, s_{S_l-1} \right]_l, \tag{9.14}$$

and

$$\widehat{\sigma}_l = \sigma_{C^{e_0+\eta_{l-1}}} \ldots \sigma_{C^{e_{E_l-1}+\eta_{l-1}}} \sigma_{C^{s_0+\eta_l}} \ldots \sigma_{C^{s_{S_l-1}+\eta_l}}. \tag{9.15}$$

This expression supports multiple-signatures with a threshold as well as multiple witnesses with a tally. A rotation event includes the ensuers list of ensuing signers and two sets of signatures, from the previously declared ersters and signers. Each set may have multiple signatures. This is described above in Section 4.3.3.

9.1.2.2 Single Signature With Witnesses

A subset of the general rotation expression is to only use a single ensuer and a single signature for each of the erster and signer but with witnesses. This is expressed as follows:

$$\varepsilon_k = \left\langle C, t_k, \eta\left(\varepsilon_{k-1}\right), \mathtt{rot}, C^{l+1}, M_l, \widehat{X}_l, \widehat{Y}_l \right\rangle \sigma_{C^{l-1}} \sigma_{C^l}, \tag{9.16}$$

where

$$\widehat{X}_l = \left[X_0, \ldots, X_{O_l-1} \right]_l, \tag{9.17}$$

and,

$$\widehat{Y}_l = \left[Y_0, \ldots, Y_{P_l-1} \right]_l. \tag{9.18}$$

The expression above is described in Section 4.1.3. It is equivalent to a multi-signature event expression but where the threshold is 1 and only one ensuer is declared and the interaction payload is empty. Likewise for each of the erster and signer index lists only one index is needed. Therefore an implementation may map an event with a missing threshold field, a single ensuer (ensuing public key), and missing erster and signer index lists to an equivalent event with a threshold of 1, an ensuer array with one element., erster index list with one element, and signer index list with one element. This is denoted as follows:

$$\varepsilon_k = \left\langle C, t_k, \eta\left(\varepsilon_{k-1}\right), \mathtt{rot}, 1, \left[C^{l+1} \right], M_l, \widehat{X}_l, \widehat{Y}_l, \{\ \}, [0]_l, [0]_l \right\rangle \widehat{\sigma}_l, \tag{9.19}$$

where

$$\widehat{X}_l = \left[X_0, \ldots, X_{O_l-1} \right]_l, \tag{9.20}$$

$$\widehat{Y}_l = \left[Y_0, \ldots, Y_{P_l-1} \right]_l, \tag{9.21}$$

and

$$\widehat{\sigma}_l = \sigma_{C^{l-1}} \sigma_{C^l}. \tag{9.22}$$

9.1.2.3 Single Signature Without Witnesses

A subset of the general rotation expression is to only use a single signature but without witnesses. This is expressed as follows:

$$\varepsilon_k = \left\langle C, t_k, \eta(\varepsilon_{k-1}), \texttt{rot}, C^{l+1} \right\rangle \sigma_{C^{l-1}} \sigma_{C^l}. \tag{9.23}$$

The expression above is described in Section 3.1.3. It is equivalent to a witnessed event expression but where the tally is 0 and the witness list is empty. Furthermore, like the previous variation, the threshold is 1 and only one ensuer is declared and the interaction data payload is empty. Moreover the ersters and signers index lists would have only one element. Therefore an implementation may map an event with missing tally, missing witness list fields, missing threshold field, a single ensuer (ensuing public key), missing ersters index list, and missing signers index list to an equivalent event with a tally of zero, empty witness list, a threshold of 1, and an ensuers array with one element, and ersters index list with one element, and a signers index list with one element. This is denoted as follows:

$$\varepsilon_k = \left\langle C, t_k, \eta(\varepsilon_{k-1}), \texttt{rot}, 1, \left[C^{l+1} \right], 0_l, [\ ]_l, [\ ]_l, \{\ \}, [0]_l, [0]_l \right\rangle \sigma_{C^{l-1}} \sigma_{C^l}.$$

9.1.2.4 Multiple Signature Without Witnesses

A different subset of the general expression is to only use multiple signatures but without witnesses. This is expressed as follows:

$$\varepsilon_k = \left\langle C, t_k, \eta(\varepsilon_{k-1}), \texttt{rot}, K_{l+1}, \widehat{C}_{l+1}, \widehat{e}_l, \widehat{s}_l \right\rangle \widehat{\sigma}_l, \tag{9.24}$$

where

$$\widehat{C}_{l+1} = \left[C^{r_{l+1}}, \ldots, C^{r_{l+1}+L_{l+1}-1} \right]_{l+1}, \tag{9.25}$$

$$\widehat{e}_l = \left[e_0, \ldots, e_{E_l-1} \right]_l, \tag{9.26}$$

$$\widehat{s}_l = \left[s_0, \ldots, s_{S_l-1} \right]_l, \tag{9.27}$$

and

$$\widehat{\sigma}_l = \sigma_{C^{e0+\eta-1}} \ldots \sigma_{C^{eE_l-1+\eta-1}} \sigma_{C^{s0+\eta}} \ldots \sigma_{C^{sS_l-1+\eta}}. \tag{9.28}$$

The expression above is described in Section 3.2.3. It is equivalent to a witnessed event expression but where the tally is 0 and the witness list is empty. Therefore an implementation may map an event with missing tally and missing witness list fields to an equivalent event with a tally of zero and empty witness list. This is denoted as follows:

$$\varepsilon_k = \left\langle C, t_k, \eta(\varepsilon_{k-1}), \texttt{rot}, K_{l+1}, \widehat{C}_{l+1}, 0_l, [\ ]_l, [\ ]_l, \{\ \}, \widehat{e}_l, \widehat{s}_l \right\rangle \widehat{\sigma}, \tag{9.29}$$

where

$$\widehat{C}_{l+1} = \left[C^{r_{l+1}}, \ldots, C^{r_{l+1}+L_{l+1}-1} \right]_{l+1}, \tag{9.30}$$

$$\widehat{e}_l = \left[e_0, \ldots, e_{E_l-1} \right]_l , \tag{9.31}$$

$$\widehat{s}_l = \left[s_0, \ldots, s_{S_l-1} \right]_l , \tag{9.32}$$

and

$$\widehat{\sigma}_l = \sigma_{C^{e0+\eta-1}} \ldots \sigma_{C^{eE_l-1+\eta-1}} \sigma_{C^{s0+\eta}} \ldots \sigma_{C^{sS_l-1+\eta}} . \tag{9.33}$$

9.1.3 Interaction

9.1.3.1 Multiple Signature

The most general expression for the generic interaction event is as follows:

$$\varepsilon_k = C_{\varepsilon_k} = \left\langle C, t_k, \eta\left(\varepsilon_{k-1}\right), \texttt{itc}, \left\{ data \right\}, \widehat{s}_k \right\rangle \widehat{\sigma}_k , \tag{9.34}$$

where

$$\widehat{s}_k = \left[s_0, \ldots, s_{S_k-1} \right]_k , \tag{9.35}$$

and

$$\widehat{\sigma}_k = \sigma_{C^{s0+\eta}} \ldots \sigma_{C^{sS_k-1+\eta}} . \tag{9.36}$$

This expression supports multiple-signatures with a threshold where the threshold and set of potential signers is declared in a previous rotation event. This is described above in Section 5.1.3.

9.1.3.2 Single Signature

A subset of the general interaction expression is to only use a single signature. This is expressed as follows:

$$\varepsilon_k = \left\langle C, t_k, \eta\left(\varepsilon_{k-1}\right), \texttt{itc}, data \right\rangle \sigma_{C^l} . \tag{9.37}$$

The expression above is described in Section 5.1.1. It is equivalent to a multi-signature event but where the signers index list has only one element and the default threshold is one. Therefore an implementation may map an event with signers index list to an equivalent event with a signers index list with one element. This is denoted as follows:

$$\varepsilon_k = \left\langle C, t_k, \eta\left(\varepsilon_{k-1}\right), \texttt{itc}, data, \left[0\right]_k \right\rangle \sigma_{C^l} .$$

9.2 State Engine

9.2.1 Controlling State Verification Engine

Using the general representations provided above for the three events, namely, inception, rotation, and interaction a generic state verification algorithm or engine can be formulated. This becomes the KERI verifier core and would be portable to all applications. The state verifier engine maintains the current verified state. The state engine process events from an event stream and updates the state from the current to the next state as a function of the current state and a verified event message. Processing is largely signature verification. The state maintains the list of designated witnesses but does not perform any validation against the number of event receipts. A witness, judge, or validator application would include the KERI state engine to provide event message verification. A controller application also maintains its copy of the state with a complementary state generator engine that instead of updating state based on external events, updates state based on internally generated events. The controller and validator applications may also includes an identifier mapper that maps the public key to/from the associated identifier. This map-

per is the interface that allows identifier independence of the core state verifier engine. A judge application would also perform additional validation based on witnessed event receipts and the current state of the tally and witness list from the KERI core state engine. Diagrams of the KERI event message state verification or state engine and the component applications are shown below:

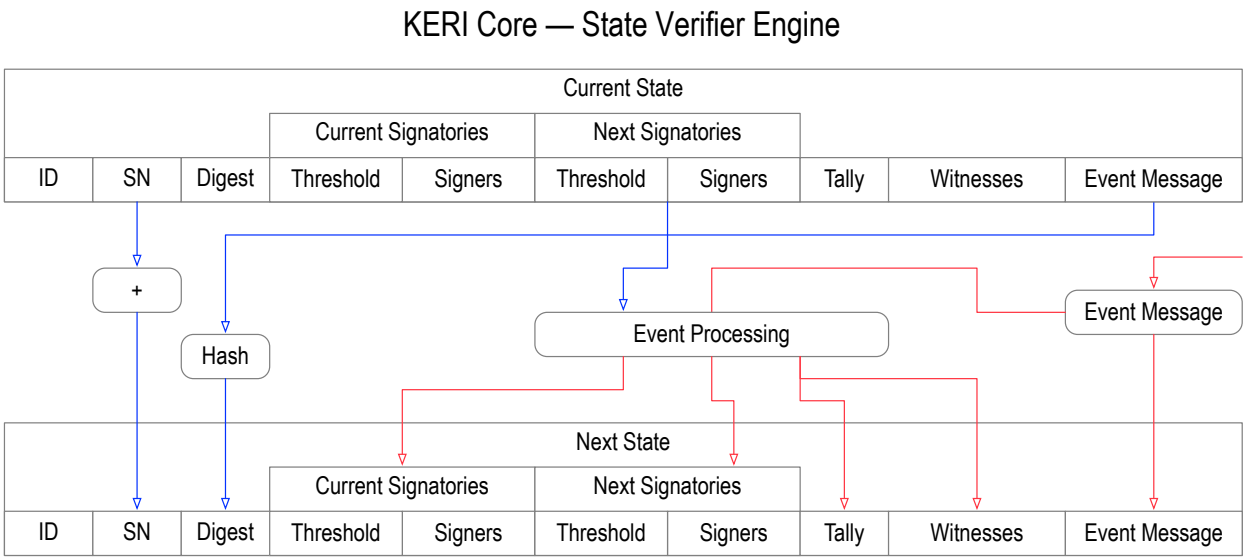

Figure 9.1. KERI Core State Verification Engine

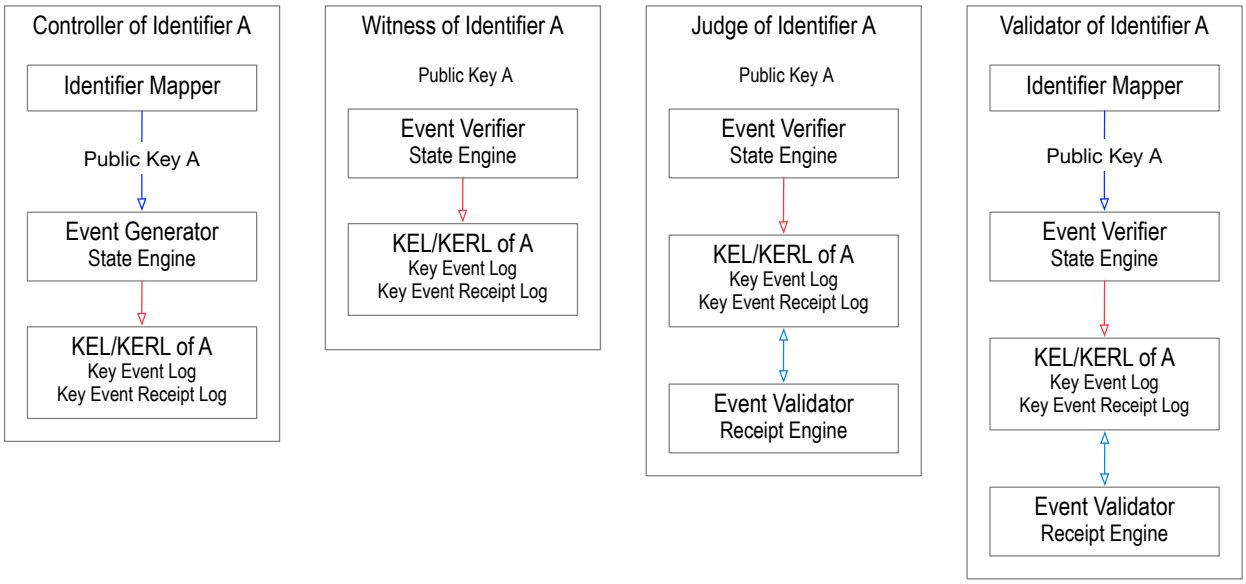

Figure 9.2. KERI Components that Employ the Core Engine.

9.2.2 Delegated State Verification Engine

A delegated key event stream has a different verification engine. It includes delegation data and requires verification of the delegator's delegating event. A diagram is shown below:

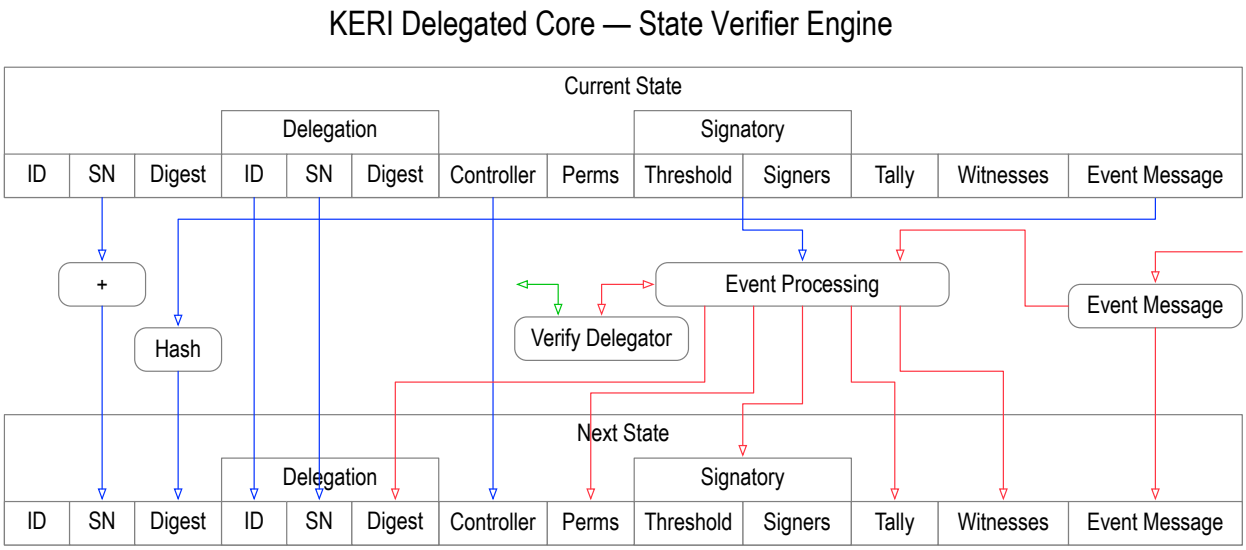

Figure 9.3. KERI Delegated Core State Verifier Engine

9.3 Identifier Independence

The KERI events reference entities with identifiers that are uniquely bound to key-pairs. A controller has an identifier labeled C that is bound to a key-pair $\left(C^0, c^0\right)$. Because the likelihood of collision of public keys generated with cryptographic strength digital key systems is remote, therefore, with loss of generality, in the event descriptions, the identifier bound to a key-pair may be equal to the public key. For example, $C = C^0$. Using only the public key enables the core parts of a KERI implementation to be independent of the syntax for specific types of self-certifying identifiers. In the case of DIDs this means that the KERI core may be DID method independent. Indeed the KERI core may be made independent of any identifier syntax other than that the identifier be uniquely bound to a cryptographic strength signing (public, private) key-pair. The public key becomes the universally unique identifier for the core and identifier specific syntax for other sections of the identifier may be handled by an identifier specific client.

This feature means that KERI may be used for any identity system based on self-certifying identifiers. This includes Ethereum smart contracts for example or other systems. In addition to its scalability and performance adaptability this feature makes KERI a candidate for a universal portable DKMI system.

9.4 Hierarchical Key Infrastructure

As discussed earlier, a common approach to key rotation management is to have a special purpose management (administrative) rotation key-pair whose function is to authorize rotation operations on a signing key-pair. This poses the problem that after multiple rotations the management rotation key becomes exposed and may be vulnerable to exploit. Consequently another higher level management rotation key may be needed to authorize rotation's of the lower level management rotation key and so on. In it simplest form KERI's pre-rotation approach uses just one level of keys. In essence a pre-rotated key is a self-managed (self-rotating) key where the pre-rotated key is effectively a one-time use management key that authorizes the next rotation. Once the rotation is completed a new management key is created and the old management key may be repurposed as a new signing key.

Pre-rotation, however, can be applied in hierarchical arrangements as well. For example, a pre-rotated key may be used only as a self-managed rotation key that is never used to sign attesta-

tions besides rotation/management but instead controls a different of a set of signing keys. The self-rotated key thereby avoids the infinite regress of even higher level management keys. Because KERI follows the design aesthetic of minimally sufficient means, the KERI core key state verification engine employs this single level approach where as operational primitive, management rotation keys are one-time use but then may be optionally repurposed as signing keys. More complex applications may be supported via composition of this primitive into multi-layer designs instead of designing a bespoke primitive for each application. This section will discuss how to compose systems with different features using the KERI core engine. For the sake of simplicity the examples in the following subsections are based on single-signature versions of the events but without loss of generality may be extended to thresholded multiple-signature versions of the events.

9.4.1 Repurposed Key Mode

The KERI core key state verification engine may be analyzed as managing a single sequence of key-pairs. What may not be obvious is that each key-pair may serve two different roles in its life-cycle. Each key starts as a one time use administrative (management) key for rotation and then after it is used to authorize a single rotation operation event may optionally become a multiple use signing key for interaction operation events. To better understand how this works consider the sequence of key-pairs as represented by the associated public key for each pair as follows:

$$C^j \Big|_{j=0,1,\ldots}, \tag{9.38}$$

such as $C^0, C^1, C^2, \ldots$ where the public key represents the key-pair.

When a key-pair is exposed in the sense that the private key has been used to create a signature a dot is added to the symbol for the key-pair, such as, $\dot{C}^j$.

When a key-pair's role is to sign key management events such as inception or rotation, i.e. administrative, an uppercase *A* is added as a subscript such as, C_A^j. When a key-pair's role is to sign interaction events an uppercase *S* is added as a subscript, such as, C_S^j. Using this notation an inception events message may be denoted as follows:

$$\varepsilon_0 = \left\langle C, t_0, \mathtt{icp}, C_A^0, C_A^1 \right\rangle \sigma_{C_A^0} \tag{9.39}$$

After the message is created $\dot{C}_A^0$ is exposed and C_A^1 has not yet been exposed. As mentioned previously best practice is to immediately follow an inception with a rotation so that the originating key-pair for the identifier has minimal exposure. Therefore, let the next event be a rotation denoted as follows:

$$\varepsilon_1 = \left\langle C, t_1, \eta\left(\varepsilon_0\right), \mathtt{rot}, C_A^2 \right\rangle \sigma_{\dot{C}_A^0} \sigma_{C_A^1}. \tag{9.40}$$

Notice that $\dot{C}_A^0$ which was used to sign the inception message is also used to sign the rotation message. This bridges its signing authority from the prior inception message which means that two keys must be exploited in order to forge the rotation message. After the rotation message is created $\dot{C}_A^0$ may be discarded. It is effectively a one time use administrative key that is used for one pair of immediately successive administrative messages. Furthermore $\dot{C}_S^1$ is now exposed and may now be repurposed as a signing key for subsequent interaction messages. Finally, C_A^2 has not yet been exposed and is the new administrative key.

Let the next event be an interaction event denoted as follows:

$$\varepsilon_2 = \left\langle C, t_2\ \eta(\varepsilon_1), \texttt{itc}, \{data\} \right\rangle \sigma_{\dot{C}_S^1}. \tag{9.41}$$

where the signing key is $\dot{C}_S^1$. Furthermore let there be a total of *I* successive interaction events. Each event is signed with the same signing key. Following the I^{th} interaction event a rotation event occurs. It may be denoted as follows:

$$\varepsilon_{1+I} = \left\langle C, t_{1+I}, \eta(\varepsilon_I), \texttt{rot}, C_A^3 \right\rangle \sigma_{\dot{C}_S^1} \sigma_{C_A^2}. \tag{9.42}$$

After this event $\dot{C}_S^1$ may be discarded, $\dot{C}_S^2$ is exposed and becomes the new signing key, and C_A^3 is the unexposed new administrative key. This may be followed by another set of interaction events all signed with $\dot{C}_S^2$. The next event may be denoted as follows:

$$\varepsilon_{2+I} = \left\langle C, t_{2+I}, \eta(\varepsilon_{1+I}), \texttt{itc}, \{data\} \right\rangle \sigma_{\dot{C}_S^2}. \tag{9.43}$$

One major purpose, of KERI is to support data streaming applications that may benefit from more performant computation infrastructure that includes the key support infrastructure. In this case only one key support infrastructure is needed for both the management and signing and re-purposing the one-time used management key does not pose a problem. In this sense KERI provides self-managing signing keys that inhabit the same computation infra-structure as the interaction messages they are signing.

9.4.2 Varied Key Support Infrastructure

In some key management schemes, however, different key support infrastructures are used for management (administrative) keys and signing keys. Management operations may be more critical but happen less frequently. Consequently a more secure albeit less convenient infra-structure such as a hardware security module (HSM) may be employed to store private keys and create signatures. In contrast, signing operations for interactions may be less critical but happen much more frequently and therefore may use a more convenient infra-structure for storage and signing. A disparity in key support infrastructure for administrative and signing keys may pose a problem for the simple KERI pre-rotation approach described above where the administrative keys are re-purposed as signing keys. When the infrastructure is disparate, in order for a management key to be repurposed as a signing key it would have to first be moved from one support infrastructure to the other. An important caveat is that KERI only uses a key as a management key for one-time use whereas in many other systems the management keys may be used multiple times. Pre-rotation forces rotation of the management key on each rotation whereas with other (hierarchical) approaches it may not be forced. In order for a disparate key support system with hierarchical keys (i.e. where administrative and signing keys use different infrastructure) to have the equivalent security properties to KERI, the administrative keys must be one-time use keys. In other words, the management (administrative) keys must be rotated on each use.

9.4.2.1 Double Rotation Mode

When disparate key support infra-structures for administrative versus signing keys are desirable then KERI may be used in double rotation mode. This does not require any changes to the KERI core state verification engine but merely a change to how the key generation engine is used. This may be thought of as analogous to the CISC (complex instruction set computer) vs RISC (reduced instruction set computer) design tradeoff. A CISC computer employs a larger set of more complex instructions each with a more narrowly designed application. This makes the overall computer hardware more complex and often each infrastructure may execute slower as a result. In contrast, a RISC computer employs a smaller set of simpler instructions each with a more generally designed application. More complex tasks are performed by using appropriate

combinations of the simpler instructions. This requires more effort by the compiler but may be implemented with simpler less expensive hardware where each instruction may execute faster as a result. Likewise the design approach of KERI is to employ simple key management primitives that may be composed to create more complex application specific key management operations. This allows the KERI core state verification engine to remain as simple as possible. Two pre-rotations per effective rotation makes it possible to avoid moving the management (administrative) keys from their support infrastructure to the signing key support infra-structure.

The following examples explains how double rotation mode enables management and signing keys sets to each have disparate key support infrastructures where management keys may never need be moved from the administrative key support to the signing key support infrastructure. In double rotation mode there are two alternating sub-sequences of keys, administrative and signing. The administrative sub-sequence is denoted as follows:

$$C_A^j \Big|_{j=0,2,4,\ldots}, \tag{9.44}$$

such as, $C_A^0, C_A^2, C_A^4, \ldots$. The signing sub-sequence is denoted as follows:

$$C_S^j \Big|_{j=1,3,5,\ldots}, \tag{9.45}$$

such as $C_S^1, C_S^3, C_S^5, \ldots$. As the expressions above show the administrative sub-sequence are the even numbered and the signing sub-sequence the odd numbered key-pairs from the original sequence.

In this double rotation mode, rotation operation occur in pairs with no intervening events. The inception event counts as a special rotation and is immediately followed by another rotation. each pair of rotations counts as a one time use of the associated keys. The first pair may be denoted as follows:

$$\begin{aligned} \varepsilon_0 &= \left\langle C, t_0, \texttt{icp}, C_A^0, C_A^1 \right\rangle \sigma_{C_A^0} \\ \varepsilon_1 &= \left\langle C, t_1, \eta\left(\varepsilon_0\right), \texttt{rot}, C_A^2 \right\rangle \sigma_{\dot{C}_A^0} \sigma_{C_S^1} \end{aligned}. \tag{9.46}$$

After the first (inception) event in the operation pair $\dot{C}_A^0$ is exposed and C_S^1 is unexposed. $\dot{C}_A^0$ uses the administrative key support infrastructure and C_S^1 uses the signing key support infrastructure. After the second (rotation) event $\dot{C}_S^1$ becomes the current signing key, C_A^2 becomes the current (unexposed) administrative key, and $\dot{C}_A^0$ may be discarded after its one time use for the double rotation operation. It is never used to sign interaction events. Although $\dot{C}_S^1$ was used to sign the second event, its signature merely provides additional security. There is no security lack with respect to key support infrastructure because $\dot{C}_A^0$ signed both events as the administrative key and employs the administrative key support infra-structure. This operation pair may now be followed with a set of *I* interaction events signed only with $\dot{C}_S^1$. Moreover $\dot{C}_S^1$ always resides in the signing infrastructure. The successive interaction event may be denoted as follows:

$$\varepsilon_2 = \left\langle C, t_2\ \eta\left(\varepsilon_1\right), \texttt{itc}, \{data\} \right\rangle \sigma_{\dot{C}_S^1}. \tag{9.47}$$

Let a total of I interaction events follow the rotation. Suppose that after the I^{th} interation event another pair of rotation events may occur as follows:

$$\begin{aligned}\varepsilon_{1+I} &= \left\langle C, t_{1+I}, \eta\left(\varepsilon_I\right), \texttt{rot}, C_A^3 \right\rangle \sigma_{\dot{C}_S^1} \sigma_{C_A^2} \\ \varepsilon_{2+I} &= \left\langle C, t_{2+I}, \eta\left(\varepsilon_{1+I}\right), \texttt{rot}, C_A^4 \right\rangle \sigma_{\dot{C}_A^2} \sigma_{C_S^3}\end{aligned} . \tag{9.48}$$

After the first (rotation) event in the pair $\dot{C}_A^2$ is exposed and C_S^3 is unexposed. $\dot{C}_A^2$ uses the administrative key support infrastructure and C_S^3 uses the signing key support infrastructure. After the second (rotation) event $\dot{C}_S^3$ becomes the current signing key, C_A^4 becomes the current (unexposed) administrative key, and $\dot{C}_A^2$ may be discarded after its one time use for the double rotation operation. It is never used to sign interaction events. Although $\dot{C}_S^3$ was used to sign the second event it merely provides additional security. There is no security lack relative to key support infrastructure because $\dot{C}_A^2$ which uses the administrative infrastructure signed both events as the administrative key. This pair may now be followed with interaction events signed only with $\dot{C}_S^3$. Moreover $\dot{C}_S^3$ does not need to be moved but may always reside in the signing infrastructure. The successive interaction event may be denoted as follows:

$$\varepsilon_{3+I} = \left\langle C, t_{3+I}, \eta\left(\varepsilon_{2+I}\right), \texttt{itc}, \{data\} \right\rangle \sigma_{\dot{C}_S^2} . \tag{9.49}$$

This process may be repeated indefinitely where the sub-sequence of C_A^j key-pairs may use a different key support infrastructure than the sub-sequence of C_S^j key-pairs without having to move any keys between the two infrastructures. Importantly this double rotation mode works just as well as the single rotation mode with the KERI core state verification engine without modification to the engine.

9.4.2.2 Delegation Mode

A limitation of double rotation mode is that it only supports one set of signing keys managed managed by one set of pre-rotated administrative keys. In essence double rotation defines two sets of keys, that are, a pre-rotated sequence of management key and a sequence of signing keys. A more general approach is to use delegation as described in 6.7 to create the set of signing keys. With delegation any number of sets of signing keys may be managed by one set of administrative keys. Delegation means that double rotation is not needed as the controlling key sequence is only used one time each for management events and is never used as a signing key.

From this perspective the administrative sequence is denoted as follows:

$$A^j \Big|_{j=0,1,\ldots} , \tag{9.50}$$

such as, $A^0, A^1, A^2, \ldots$. The signing key sequence is denoted as follows:

$$S^j \Big|_{j=0,1,\ldots} , \tag{9.51}$$

such as $S^0, S^1, S^2, \ldots$. The administrative key-pairs, A^j, may use different key support infrastructure from the signing key-pairs S^j. The delegated signing key in the rotation event is not part of the sequence of keys used for administration. This semantically creates a key hierarchy. In essence the rotation operation is delegating signing authority from the administrative key to the signing key. Because the key-pairs for the administrative and signing keys are unique they may be associated with different identifiers. In other words, there are two identifiers each with a controller but one controller (delegate) receives its authority via delegation from the delegating controller (delegator). Let the identifier of the delegator be *A* which is controlled by the rotated sequence $A^0, A^1, A^2, \ldots$. Controller *A* is delegating signing

authority to a signing key identified by S which is controlled by rotated sequence $S^0, S^1, S^2, \ldots$. Furthermore, if A only performs key management operations (rotation or delegated rotation), that is, none of the A^j are used to sign plain interaction events then the A^j may reside in a different key support infrastructure than the S^j. Double rotation is no longer required to satisfy this condition. Furthermore, using delegation provides a powerful new building block that allows composition in more ways. The only needed additional infrastructure is a version of the KERI core state verification engine that supports delegated inception and rotation events.

9.4.2.3 Duet Engine

The three modes of using KERI, that are, (1) Repurposed Key Mode, (2) Double Rotation Mode, and (3) Delegation Mode provide generic composable building blocks for DKMI. All three modes use the KERI core state verification engine and the third (delegation mode) also uses the KERI delegated state verification engine. With these two engines various architectures are supported. These range from a single self-managed sequence of signing keys, to a hierarchical tree of managed signing keys. The KERI core engine may be called a solo engine because it maintains the state of a single sequence of keys. This sequence may be used in two different modes repurpose and double rotation. One may argue that both the double rotation mode and the delegated mode are both more cumbersome than a custom set of inception and rotation events that explicitly uses two sets of keys, that are, a pre-rotated sequence of management key and a sequence of signing keys. Although simple and focused are an valuable feature, the drawback of this approach is that it would require a new version of the state verification engine just to support that configuration. Other use cases might drive further variants to accommodate application specific features. Indeed each additional customization to the syntax and semantics of the events would require yet another version of the state verification engine. The task of building a universal DKMI will become problematic if every custom application needed to create its own key event set. The design approach behind KERI is to formulate a minimal set of primitives that allow building almost any type of DMKI. Nonetheless the use case of a single administrative key sequence managing a single signing key sequence may be a very important one. Maybe important enough to justify a custom state verification engine for that use case. We call this the duet engine because is maintains the state of two distinct key sequences. A description of the key events for a notional duet engine is provided in an appendix (see Section 10.).

9.5 Implementation Choices

Because KERI is designed to be compatible with event streaming applications, its design lends itself to a simple state verification engine with compact and efficient syntax. Furthermore KERI has advanced key management features. KERI provides for reconfigurable thresholded multiple signature schemes where both the threshold total number of signatures may change are each rotation. The pre-rotation makes a forward commitment to an unexposed key-pair(s) that provides security that may not be undone via exploit of any exposed key-pairs. This allows for reconfigurability without sacrificing security. These advanced key management features alone make KERI desirable even in non-data streaming applications where scalability and performance or not so important. Likewise many applications for other reasons require a distributed consensus ledger. In that case the best approach might be to use KERI but without witnesses and leverage the trust provided by the distributed consensus nodes. A smart contract system like Ethereum (public or private) has the capacity to support the semantics of the KERI Core state verification engine on chain. Alternatively, KERI could be implemented as a side state channel with a Judge or Validator periodically anchoring the current state to a distributed ledger such as Ethereum or Bitcoin. This side channel approach will work with most distributed ledgers.

As a specific comparison, consider other smart contract based systems such as identifiers created with the ERC-1056 standard [8; 9] or control of tokenized assets using the Gnosis MuliSig-Wallet on Ethereum [18]. Both are vulnerable to capture via exploit of the exposed signing key-pairs. In the case of ERC-1056 only one key-air need be exploited or a threshold number of key-pairs for MultiSig. The Gnosis MultiSig wallet is a smart contract with advanced features for multiple signatures. It allows changes to the threshold and number of signatures as well as revoking and replacing signatures. In this sense it is of comparable complexity to the KERI engine. The crucial limitation, however, with the Gnosis MultiSig wallet is that an exploiter of a threshold number of exposed signing key-pairs may undo or permanently capture the wallet. Whereas KERI's pre-rotation scheme makes a forward commitment to unexposed key-pairs that may not be undone via exploit of any exposed signing key-pairs. This means that recovery from an exploit in KERI may be overcome (recaptured) by performing a rotation but not with the Gnosis multi-sig wallet. At least the Gnosis MultiSig Wallet has the advantage of multiple-signatures which ERC-1056 does not. But neither benefit from the in-stride pre-rotation security of KERI.

Non distributed consensus ledger implementations uniquely benefit from KERI's event streaming design. An example implementation target may be Apache Kafka [3] or Apache Flink [2]. Both Apache Kafka and Flink provide libraries for building scalable event processing streams for data intensive applications. The features and semantics may differ somewhat. Nonetheless, the KERI core stater verification engine can be implemented as either a Kafka streams application or a Flink ProcessFunction. This allows implementation of KERI witnesses and validators as Kafka or Flink streams. The witness receipt validation function of a KERI Judge may also be implemented as a Kafka or Flink stream. Thus the whole KERI DKMI may be hosted on scalable Kafka or Flink clusters. Similarly KERI may be easily implemented using asynchronous flow based processing frameworks such as Ioflo [19].

10 KERI DUET ENGINE

This section describes a custom engine and key events that maintain two sets of keys. These are a pre-rotated sequence of management keys and a sequence of signing keys. This may be called a duet engine or (double key) approach, the administrative sequence is denoted as follows:

$$A^j \Big|_{j=0,1,\ldots}, \tag{10.1}$$

such as, $A^0, A^1, A^2, \ldots$. The signing key sequence is denoted as follows:

$$S^j \Big|_{j=0,1,\ldots}, \tag{10.2}$$

such as $S^0, S^1, S^2, \ldots$. The administrative key-pairs, A^j, may use different key support infrastructure from the signing key-pairs S^j.

The custom inception event is denoted as follows:

$$\varepsilon_0 = \left\langle C, t_0, \texttt{icp}, A^0, A^1, S^0 \right\rangle \sigma_{A^0}, \tag{10.3}$$

where A^0 represents the incepting (zeroth) administrative key-pair, $A^0 = C^0 = C$, A^1 represents the first pre-rotated administrative key-pair, S^0 represents the incepting (zeroth) signing key-pair, and the event is signed with the private key for A^0. After this event $\dot{A}^0$ is exposed. Any number of interaction events may follow all signed with S^0 without further exposing any of the A^j. If

the next event is an interaction event is may be denoted as follows:

$$\varepsilon_1 = \left\langle C, t_1\ \eta(\varepsilon_0), \texttt{itc}, \{data\} \right\rangle \sigma_{S^0}\,. \tag{10.4}$$

After this event $\dot{S}^0$ is exposed (or a single set of signatures if multi-signature). Suppose a total of I interaction events occur followed by another rotation event. This rotation event may be signed only by A^1 as follows:

$$\varepsilon_{1+I} = \left\langle C, t_{1+I}, \eta(\varepsilon_I), \texttt{rot}, A^2, S^1 \right\rangle \sigma_{A^1} \tag{10.5}$$

Subsequent interaction events will be signed by S^1 (or a set of signers if multi-signature) . For example the next successive interaction event may be denoted as follows:

$$\varepsilon_{2+I} = \left\langle C, t_{2+I}\ \eta(\varepsilon_{1+I}), \texttt{itc}, \{data\} \right\rangle \sigma_{S^1}\,. \tag{10.6}$$

In general the l^{th} rotation event appearing as the k^{th} event with a single signature (single set of signatures if multi-signature) may be denoted as follows:

$$\varepsilon_k = \left\langle C, t_k, \eta(\varepsilon_{k-1}), \texttt{rot}, A^{l+1}, S^l \right\rangle \sigma_{A^l}\,. \tag{10.7}$$

This may be slightly more vulnerable to forgery than the KERI solo engine because only one signature (set of signatures) is used. Nonetheless because A^1 is only used once in the management infra-structure it may be sufficient. Using more signatures would make this equivalent to double rotation mode and hence provides no advantage to implementing this custom duet engine.

While this example of a custom event set is clean and only requires one event per rotation operation it still uses more keys that the the single rotation (repurposed key mode) for KERI albeit it uses the same number of keys as double rotation mode. Its main difference is that it only uses one signature instead of two for each rotation. This has the advantage of less computation and may be semantically easier to reason about or at least more familiar to reason about than the solo engine but may have increased vulnerability to forgery (but maybe not significantly so). If the only use case, however, is covered by this duet engine then it may be preferable. This description may be extended to multi-signature witnessed versions in a straightforward manner.

11 Conclusion

KERI is a decentralized key management infrastructure (DKMI) based on the principle of minimally sufficient means. KERI is designed to be compatible with event streaming applications but may be employed in distributed ledger systems as well. The event streaming design lends itself to a simple state verification engine. The syntax is compact and efficient. Nonetheless KERI has advanced key management features. The principle key management operation is key rotation via a novel in-stride key pre-rotation scheme. This utilizes a single sequence of controlling key-pairs for easier management. KERI provides for reconfigurable thresholded multiple signature schemes where both the threshold and total number of signatures may change at each rotation. KERI also provides for fractional weighted multiple signatures schemes. The pre-rotation makes a forward commitment to unexposed key-pair(s) that provide security that may not be undone via exploit of any exposed key-pairs. This allows for reconfigurability without sacrificing security. KERI also provides for reconfigurable designation of witnesses and quorum (tally) sizes where both the total number of witnesses and quorum size may change from rotation to rotation. Once again the pre-rotation forward commitment to unexposed key-pairs means that the witness configuration may not be undone via exploit of any exposed key-pairs. KERI is the only event-streaming capable system that we know of that provides this combination of advanced features.

A delegated version of KERI is also provided that enables hierarchical key management where a master controller (identifier) key event stream may delegate signing authority to one or more slave identifier key event streams thereby forming a chained tree of key event streams.

KERI scalable design supports multiple use cases. Two primary trust modalities motivated the design, these are an online or pair-wise mode and an offline or any-wise mode. In the online mode the identity controller establishes control via verified signatures of the controlling key-pair. The offline mode extends that trust basis with witnessed key event receipt logs (KERL) for validating events. This gives rise to the acronym KERI for key event receipt infrastructure. The KERI approach may be much more performant and scalable than more complex approaches that depend on a total ordering distributed consensus ledger. Nevertheless KERI may employ a distributed consensus ledger when other considerations make it the best choice. In other words KERI may be augmented with distributed consensus ledgers but does not require them. KERI is applicable to DKMI in data streaming, web 3.0, and IoT applications where performance and scalability are important. KERI is designed to support DIDs but its core services are identifier independent (this includes DID method independence). This makes KERI a simple universal portable DKMI.

Author

Samuel M. Smith Ph.D. is an expert in decentralized identity and reputations systems. Samuel received a Ph.D. in Electrical and Computer Engineering from Brigham Young University in 1991. He then spent 10 years at Florida Atlantic University, eventually reaching full professor status. In addition to decentralized identity and reputation, he has performed pioneering research in automated reasoning, machine learning, and autonomous vehicle systems. He has over 100 refereed publications in these areas and was principal investigator on numerous federally funded research projects. Dr. Smith has been an active participant in open standards development for networking protocols, and decentralized identity. He is also a serial entrepreneur.

12 References

[1] Allen, C., “The Path to Self-Sovereign Identity,” Life With Alacrity, 2016/04/25
http://www.lifewithalacrity.com/2016/04/the-path-to-self-soverereign-identity.html

[2] “Apache Flink,” Apache Software Foundation,
https://flink.apache.org

[3] “Apache Kafka,” Apache Software Foundation,
https://kafka.apache.org

[4] Arciszewski, S., “No Way, JOSE! Javascript Object Signing and Encryption is a Bad Standard That Everyone Should Avoid,” Paragon Initiative, 2017/03/14
https://paragonie.com/blog/2017/03/jwt-json-web-tokens-is-bad-standard-that-everyone-should-avoid

[5] Aumasson, J.-P., “BLAKE2 — fast secure hashing,”
https://blake2.net

[6] Bernstein, D. J., “Ed25519: high-speed high-security signatures,”
https://ed25519.cr.yp.to

[7] Conway, S., Hughes, A., Ma, M. et al., “A DID for Everything,” Rebooting the Web of Trust RWOT 7, 2018/09/26
https://github.com/SmithSamuelM/Papers/blob/master/whitepapers/A_DID_for_everything.pdf

[8] “ERC-1056 Lightweight Identity,” Ethereum Foundation EIP,
https://github.com/ethereum/EIPs/issues/1056

[9] “Ethr-DID Library,” uPort.me,
https://github.com/uport-project/ethr-did

[10] Fielding, R. and Reschke, J., “RFC-7320: Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing,” Internet Engineering Task Force (IETF), 2014/06/01
https://tools.ietf.org/html/rfc7230#section-3.2.1

[11] Girault, M., “Self-certified public keys,” Workshop on the Theory and Application of of Cryptographic Techniques
EUROCRYPT 1991: Advances in Cryptology — EUROCRYPT ’91 pp 490-497, 1991
https://link.springer.com/content/pdf/10.1007%2F3-540-46416-6_42.pdf

[12] “ISO 8601,” Wikipedia,
https://en.wikipedia.org/wiki/ISO_8601

[13] “ISO 8601 and Nanosecond Precision Across Languages,” nbsoft solutions, 2016/06/14
https://nbsoftsolutions.com/blog/iso-8601-and-nanosecond-precision-across-languages

[14] Josefsson, S., “RFC-3548: The Base16, Base32, and Base64 Data Encodings,” IETF, 2006-10-01
https://tools.ietf.org/html/rfc4648

[15] Kaminsky, M. and Banks, E., “SFS-HTTP: Securing the Web with Self-Certifying URLs,” MIT, 1999
https://pdos.csail.mit.edu/~kaminsky/sfs-http.ps

[16] Mazieres, D. and Kaashoek, M. F., “Escaping the Evils of Centralized Control with self-certifying pathnames,” MIT Laboratory for Computer Science, 2000
http://www.sigops.org/ew-history/1998/papers/mazieres.ps

[17] Mazieres, D., “Self-certifying File System,” MIT Ph.D. Dissertation, 2000/06/01
https://pdos.csail.mit.edu/~ericp/doc/sfs-thesis.ps

[18] “MultiSigWallet,” Gnosis,
https://github.com/gnosis/MultiSigWallet

[19] Smith, S. M., “Ioflo,”
https://github.com/ioflo/ioflo

[20] “self-sovereign-identity,” GitHub,

https://github.com/WebOfTrustInfo/self-sovereign-identity

[21] Smith, S. M. and Khovratovich, D., “Identity System Essentials,” 2016/03/29

https://github.com/SmithSamuelM/Papers/blob/master/whitepapers/Identity-System-Essentials.pdf

[22] Smith, S. M., “Key Event Receipt Infrastructure (KERI) Design and Build,” 2019/07/03

https://github.com/SmithSamuelM/Papers/blob/master/whitepapers/KERI_WP.pdf

[23] Smith, S. M., “Decentralized Autonomic Data (DAD) and the three R’s of Key Management,” Rebooting the Web of Trust RWOT 6, Spring 2018

https://github.com/SmithSamuelM/Papers/blob/master/whitepapers/DecentralizedAutonomicData.pdf

[24] Staff, P. I. E., “One Login To Rule them All - Seamless and Secure Cross-Domain Authentication,” Paragon Initiative, 2016/02/22

https://paragonie.com/blog/2016/02/one-login-rule-them-all-seamless-and-secure-cross-domain-authentication

[25] Staff, P. I. E., “Split Tokens: Token-Based Authentication Protocols without Side-Channels,” Paragon Initiative, 2017/02/28

https://paragonie.com/blog/2017/02/split-tokens-token-based-authentication-protocols-without-side-channels

[26] W3C, “Decentralized Identifiers (DIDs),” W3C Draft Community Group Report,

https://w3c-ccg.github.io/did-spec/

[27] W3C, “A Primer for Decentralized Identifiers: An introduction to self-administered identifiers for curious people,” W3C,

https://w3c-ccg.github.io/did-primer/

[28] Windley, P. J., “Soverign-Source Identity, Autonomy, and Learning,” Technometria, 2016/01/19

http://www.windley.com/archives/2016/01/soverign-source_identity_autonomy_and_learning.shtml