3 Cryptographic primitives

Figure 2 introduces the cryptographic abstractions used in this document. We begin by listing the abstract types, which are meant to represent the corresponding concepts in cryptography. Their exact implementation remains open to interpretation and we do not rely on any additional properties of public key cryptography that are not explicitly stated in this document. The types and rules we give here are needed in order to guarantee certain security properties of the delegation process, which we discuss later.

The cryptographic concepts required for the formal definition of witnessing include public-private key pairs, one-way functions, signatures and multi-signature scripts. The constraint we introduce states that a signature of some data signed with a (private) key is only correct whenever we can verify it using the corresponding public key.

Abstract data types in this paper are essentially placeholders with names indicating the data types they are meant to represent in an implementation. Derived types are made up of data structures (i.e. products, lists, finite maps, etc.) built from abstract types. The underlying structure of a data type is implementation-dependent and furthermore, the way the data is stored on physical storage can vary as well.

Serialization is a physical manifestation of data on a given storage device. In this document, the properties and rules we state involving serialization are assumed to hold true independently of the storage medium and style of data organization chosen for an implementation. The type Ser denotes the serialized representation of a term of any serializable type.

Abstract types

$$
\begin{array}{rcll}
sk & \in & \mathsf{SKey} & \text{private signing key} \\
vk & \in & \mathsf{VKey} & \text{public verifying key} \\
hk & \in & \mathsf{KeyHash} & \text{hash of a key} \\
\sigma & \in & \mathsf{Sig} & \text{signature} \\
d & \in & \mathsf{Ser} & \text{data} \\
script & \in & \mathsf{Script} & \text{multi-signature script} \\
hs & \in & \mathsf{ScriptHash} & \text{hash of a script}
\end{array}
$$

Derived types

$$
(sk, vk) \in \mathsf{KeyPair} \quad \text{signing-verifying key pairs}
$$

Abstract functions

$$
\begin{array}{rcll}
\mathsf{hashKey} & \in & \mathsf{VKey} \to \mathsf{KeyHash} & \text{hash a verification key} \\
\mathsf{verify} & \in & \mathbb{P}\ (\mathsf{VKey} \times \mathsf{Ser} \times \mathsf{Sig}) & \text{verification relation} \\
\mathsf{sign} & \in & \mathsf{SKey} \to \mathsf{Ser} \to \mathsf{Sig} & \text{signing function} \\
\mathsf{hashScript} & \in & \mathsf{Script} \to \mathsf{ScriptHash} & \text{hash a serialized script}
\end{array}
$$

Constraints

$$
\forall (sk, vk) \in \mathsf{KeyPair},\ d \in \mathsf{Ser},\ \sigma \in \mathsf{Sig} \cdot \mathsf{sign}\ sk\ d = \sigma \implies (vk, d, \sigma) \in \mathsf{verify}
$$

Notation for serialized and verified data

$$
\begin{array}{ll}
[\![x]\!] \in \mathsf{Ser} & \text{serialised representation of } x \\
\mathcal{V}_{vk}[\![d]\!]_\sigma = \mathsf{verify}\ vk\ d\ \sigma & \text{shorthand notation for verify}
\end{array}
$$

Figure 2: Cryptographic definitions

When we get to the blockchain layer validation, we will use key evolving signatures (KES).

This is another asymmetric key cryptographic scheme, also relying on the use of public and private key pairs. These signature schemes provide forward cryptographic security, meaning that a compromised key does not make it easier for an adversary to forge a signature that allegedly had been signed in the past. Figure 3 introduces the additional cryptographic abstractions needed for KES.

In KES, the public verification key stays constant, but the corresponding private key evolves incrementally. For this reason, KES signing keys are indexed by integers representing the step in the key's evolution. This evolution step parameter is also an additional parameter needed for the signing (denoted by $\mathsf{sign}_{\mathsf{ev}}$) and verification (denoted by $\mathsf{verify}_{\mathsf{ev}}$) functions.

Since the private key evolves incrementally in a KES scheme, the ledger rules require the pool operators to evolve their keys every time a certain number of slots have passed, as determined by the global constant SlotsPerKESPeriod.

Abstract types

$$\begin{array}{rcll} sk & \in & \mathbb{N} \to \mathsf{SKey}_{\mathsf{ev}} & \text{private signing keys} \\ vk & \in & \mathsf{VKey}_{\mathsf{ev}} & \text{public verifying key} \end{array}$$

Notation for evolved signing key

$$sk_n = sk\ n \qquad\qquad n\text{-th evolution of } sk$$

Derived types

$$(sk_n, vk) \in \mathsf{KeyPair}_{\mathsf{ev}} \quad \text{signing-verifying key pairs}$$

Abstract functions

$$\begin{array}{rcll} \mathsf{verify}_{\mathsf{ev}} & \in & \mathbb{P}\ (\mathsf{VKey} \times \mathbb{N} \times \mathsf{Ser} \times \mathsf{Sig}) & \text{verification relation} \\ \mathsf{sign}_{\mathsf{ev}} & \in & (\mathbb{N} \to \mathsf{SKey}_{\mathsf{ev}}) \to \mathbb{N} \to \mathsf{Ser} \to \mathsf{Sig} & \text{signing function} \end{array}$$

Constraints

$$\begin{array}{c} \forall n \in \mathbb{N}, (sk_n, vk) \in \mathsf{KeyPair}_{\mathsf{ev}},\ d \in \mathsf{Ser},\ \sigma \in \mathsf{Sig} \cdot \\ \mathsf{sign}_{\mathsf{ev}}\ sk\ n\ d = \sigma \implies \mathsf{verify}_{\mathsf{ev}}\ vk\ n\ d\ \sigma \end{array}$$

Notation for verified KES data

$$\mathcal{V}^{\mathsf{KES}}_{vk}[\![d]\!]^n_\sigma = \mathsf{verify}_{\mathsf{ev}}\ vk\ n\ d\ \sigma \qquad \text{shorthand notation for } \mathsf{verify}_{\mathsf{ev}}$$

Figure 3: KES Cryptographic definitions

Figure 4 shows the types for multi-signature schemes. Multi-signatures effectively specify one or more combinations of cryptographic signatures which are considered valid. This is realized in a native way via a script-like DSL which allows for defining terms that can be evaluated. Multi-signature scripts is the only type of script (for any purpose, including output-locking) that exist in Shelley.

The terms form a tree like structure and are evaluated via the evalMultiSigScript function. The parameters are a script and a set of key hashes. The function returns True when the supplied key hashes are a valid combination for the script, otherwise it returns False. The following are the four constructors that make up the multisignature script scheme:

RequireSig : the signature of a key with a specific hash is required;

RequireAllOf : signatures of all of the keys that hash to the values specified in the given list are required;

RequireAnyOf : a single signature is required, by a key hashing to one of the given values in the list (this constructor is redundant and can be expressed using RequireMOf);

RequireMOf : m of the keys with the hashes specified in the list are required to sign

MultiSig Type

$$\begin{aligned} \mathsf{MSig} &\subseteq \mathsf{Script} \\ \\ msig \in \mathsf{MSig} &= \mathsf{RequireSig\ KeyHash} \\ &\uplus \mathsf{RequireAllOf\ [Script]} \\ &\uplus \mathsf{RequireAnyOf\ [Script]} \\ &\uplus \mathsf{RequireMOf\ } \mathbb{N}\ \mathsf{[Script]} \end{aligned}$$

Functions

$$\begin{aligned} &\mathsf{evalMultiSigScript} \in \mathsf{MSig} \to \mathbb{P}\ \mathsf{KeyHash} \to \mathsf{Bool} \\ &\mathsf{evalMultiSigScript}\ (\mathsf{RequireSig}\ hk)\ vhks = hk \in vhks \\ &\mathsf{evalMultiSigScript}\ (\mathsf{RequireAllOf}\ ts)\ vhks = \forall t \in ts : \mathsf{evalMultiSigScript}\ t\ vhks \\ &\mathsf{evalMultiSigScript}\ (\mathsf{RequireAnyOf}\ ts)\ vhks = \exists t \in ts : \mathsf{evalMultiSigScript}\ t\ vhks \\ &\mathsf{evalMultiSigScript}\ (\mathsf{RequireMOf}\ m\ ts)\ vhks = \\ &\qquad\qquad m \leq \Sigma\ ([\mathsf{if}\ (\mathsf{evalMultiSigScript}\ t\ vhks)\ \mathsf{then}\ 1\ \mathsf{else}\ 0 | t \leftarrow ts]) \end{aligned}$$

Figure 4: Multi-signature via Native Scripts

Figure 5 shows the cryptographic abstractions needed for Verifiable Random Functions (VRF). VRFs allow key-pair owners, $(sk, vk) \in \mathsf{KeyPair}$, to evaluate a pseudorandom function in a provable way given a randomness seed. Any party with access to the verification key, vk, the randomness seed, the proof and the generated randomness can indeed verify that the value is pseudorandom.

Abstract types

$$\begin{array}{rcll} \mathit{seed} & \in & \mathsf{Seed} & \text{seed for pseudo-random number generator} \\ \mathit{prf} & \in & \mathsf{Proof} & \text{VRF proof} \end{array}$$

Abstract functions (T an arbitrary type)

$$\begin{array}{rcll} \star & \in & \mathsf{Seed} \to \mathsf{Seed} \to \mathsf{Seed} & \text{binary seed operation} \\ \mathsf{vrf}_{\mathsf{T}} & \in & \mathsf{SKey} \to \mathsf{Seed} \to \mathsf{T} \times \mathsf{Proof} & \text{verifiable random function} \\ \mathsf{verifyVrf}_{\mathsf{T}} & \in & \mathsf{VKey} \to \mathsf{Seed} \to \mathsf{Proof} \times \mathsf{T} \to \mathsf{Bool} & \text{verify vrf proof} \end{array}$$

Derived Types

$$\mathsf{PoolDistr} = \mathsf{KeyHash}_{pool} \mapsto \Big([0,1] \times \mathsf{KeyHash}_{vrf}\Big) \qquad \text{stake pool distribution}$$

Constraints

$$\forall (sk, vk) \in \mathsf{KeyPair},\ \mathit{seed} \in \mathsf{Seed},\ \mathsf{verifyVrf}_T\ vk\ \mathit{seed}\ \ (\mathsf{vrf}_T\ sk\ \mathit{seed})$$

Constants

$$\begin{array}{ll} 0_{\mathit{seed}} \in \mathsf{Seed} & \text{neutral seed element} \\ \mathsf{Seed}_{\ell} \in \mathsf{Seed} & \text{leader seed constant} \\ \mathsf{Seed}_{\eta} \in \mathsf{Seed} & \text{nonce seed constant} \end{array}$$

Figure 5: VRF definitions

4 Addresses

Addresses are described in section 4.2 of the delegation design document [SL-D1]. The types needed for the addresses are defined in Figure 6. They all involve a credential, which is either a key or a multi-signature script. There are four types of UTxO addresses:

- Base addresses, $\mathsf{Addr}_{\mathsf{base}}$, containing the hash of a payment credential and the hash of a staking credential. Note that the payment credential hash is the hash of the key (or script) which has contol of the funds at this address, i.e. is able to witness spending them. The staking credential controls the delegation decision for the Ada at this address (i.e. it is used for rewards, staking, etc.). The staking credential must be a (registered) delegation credential (see Section 9 for a discussion of the delegation mechanism).
- Pointer addresses, $\mathsf{Addr}_{\mathsf{ptr}}$, containing the hash of a payment credential and a pointer to a stake credential registration certificate.
- Enterprise addresses, $\mathsf{Addr}_{\mathsf{enterprise}}$, containing only the hash of a payment credential (and which have no staking rights).
- Bootstrap addresses, $\mathsf{Addr}_{\mathsf{bootstrap}}$, corresponding to the addresses in Byron, behaving exactly like enterprise addresses with a key hash payment credential.

Where a credential is either a key or a multi-signature script. Together, these four address types make up the Addr type, which will be used in transaction outputs in Section 8. The notations $\mathsf{Credential}_{pay}$ and $\mathsf{Credential}_{stake}$ do not represent distinct types. The subscripts are annotations indicating how the credential is being used.

Section 5.5.2 of [SL-D1] provides the motivation behind enterprise addresses and explains why one might forgo staking rights. Bootstrap addresses are needed for the Byron-Shelley transition in order to accommodate having UTxO entries from the Byron era during the Shelley era.

There are also subtypes of the address types which correspond to the credential being either a key hash (the *vkey* subtype) or a script hash (the *script* subtype). So for example $\mathsf{Addr}^{script}_{base}$ is the type of base addresses which have a script hash as pay credential. This approach is used to facilitate expressing the restriction of the domain of certain functions to a specific credential type.

Note that for security, privacy and usability reasons, the staking (delegating) credential associated with an address should be different from its payment credential. Before the stake credential is registered and delegated to an existing stake pool, the payment credential can be used for transactions, though it will not receive rewards from staking. Once a stake credential is registered, the shorter pointer addresses can be generated.

Finally, there is an account style address $\mathsf{Addr}_{\mathsf{rwd}}$ which contains the hash of a staking credential. These account addresses will only be used for receiving rewards from the proof of stake leader election. Appendix A of [SL-D1] explains this design choice. The mechanism for transferring rewards from these accounts will be explained in Section 8 and follows the approach outlined in the document [Zah18].

Note that, even though in the Cardano system, most of the accounting is UTxO-style, the reward addresses are a special case. Their use is restricted to only special cases (e.g. collecting rewards from them), outlined in the rules in Sections 8 and Section 11. For each staking credential, we use the function $\mathsf{addr}_{\mathsf{rwd}}$ to create the reward address corresponding to the credential, or to access an existing one if it already exists. Note that $\mathsf{addr}_{\mathsf{rwd}}$ uses the global constant NetworkId to attach a network ID to the given stake credential.

Base, pointer and enterprise addresses contain a payment credential which is either a key hash or a script hash. Base addresses contain a staking credential which is also either a key hash or a script hash.