Cryptography in IOHK

Technical Report # CR-01

Iñigo Querejeta-Azurmendi <querejeta.azurmendi@iohk.io>

Report Registration Date: February 9, 2022

Version 0.1.0
February 9, 2022

Dissemination Level		
PU	Public	
CO	Confidential, only for company distribution	
DR	Draft, not for general circulation	√

Contents

List of Figures

A Formal Specification of the cryptographic primitives used in IOHK

Iñigo Querejeta-Azurmendi
querejeta.azurmendi@iohk.io

February 9, 2022

Abstract

This document contains the list of the different cryptographic primitives of interest for IOHK. We present in detail the instantitations that are being used/implemented, as well as the parameters used in our production systems. A first iteration of this document

List of Contributors

Including new primitives

If a project needs a particular primitive, which is not present in this document, the mechanism to include is simple. Add an issue to the repo, `https://github.com/input-output-hk/cryptography_spec`, specifying why such a primitive is of interest, and how it would be used. To make the discussion more effective, it is also of interest to link references to the specification, or available implementations.

1 Introduction

Table of primitives

Table 1: This table presents a list of the different cryptographic primitives that are being used in IOHK, or primitives that are being worked on. We divide the status in four groups: 🟢, 🟡, 🟠, ⚫, to represent primitives used in production, being tested or audited, being under development, or deprecated respectively.

List of cryptographic primitives								
Name	Protocol	Language	Tech. Details	Project	Status	Provider	Internal Contact	Plutus support
ed25519	EdDSA implemented over Curve25519.	C	Section ??	Cardano node	🟢	libsodium		Yes
KES	Key Evolving Signatures. Implemented on top of a digital signature scheme, in particular ed25519. Haskell implementation depends on libsodium.	Haskell	Section ??	Cardano node	🟢	IOHK (based on libsodium)		Yes
		Rust		Jormungandr	🟢	IOHK		
MuSig2	Schnorr-based multi signatures. In particular, this implementation is compatible with ed25519 signatures.	C	Section ??	Hydra	🟠	IOHK (based on libsodium)		Yes
STM	Stake-based Threshold Multi signatures. This construction depends on a pairing friendly curve.	Rust	Section ??	Mithril	🟠	IOHK/Galois		No
ATM	Ad-hoc Threshold Multi signatures. Achieves the same goal as Threshold Signatures, with the difference that the key generation does not require a distributed procedure. Each party generates their key pair, and then a global public key is computed.	Rust	Section ??	Side chains	🟠	IOHK/Galois		No
VRF	Verifiable Random Function, implemented over Curve25519.	C	Section ??	Cardano node	🟢	IOHK		-
		Rust		Jormungandr	🟢	IOHK		
Plonk	zk-SNARK. No generic computation available yet, restricted to the gadgets implemented.	Rust	Section ??	Midnight	🟠	ZK-Garage		No

2 Notation

In this section we introduce some generic notation used throughout the spec. The primitive-specific notation is introduced in the respective sections.

We denote by $E(\mathbb{F}_p)$ the finite abelian group based on an elliptic curve over a finite prime-order field $\mathbb{F}_p$ (note that we simplify the notation and drop the explicit dependency on $\mathbb{F}_p$ and security parameter κ). Most importantly, we assume the order of the group E to be of the form $\texttt{cofvar} \cdot q$ for some small *cofactor* `cofvar` (sometimes equal to 1) and large prime number q, and that the (hence) unique (sub)group $\mathbf{G}$ of order q is generated by a known base point G, i.e., $\mathbf{G} = \langle G \rangle$, in which the computational Diffie-Hellman (CDH) problem is believed to be hard. We use `H` to denote a cryptographically safe hash function, modeled as a random oracle, $\texttt{H} : \{0,1\}^* \rightarrow \{0,1\}^{\ell(\kappa)}$.

3 Ed25519

3.1 Summary

Ed25519 is the signature scheme being used in Cardano, and what is currently (as per 14th January 2022) supported in Plutus. Ed25519 is an instantiation of the EdDSA over the elliptic curve Edwards25519, which is in turn closely related to Schnorr signatures.

3.2 Generalised specification

This section presents the generalized signature system EdDSA, and in Section **??**, we present the specific parameters used in Cardano. EdDSA is parametrized with the following parameters. An integer $b \geq 10$, a cryptographic hash function `H` producing a $2b$-bit output, and a finite abelian group based on an elliptic curve. An EdDSA signature consists of the following three algorithms:

- $\texttt{KeyGen}(1^\lambda)$ takes as input the security parameter λ and returns a key-pair (x, vk). First, it chooses $x \leftarrow \{0,1\}^b$. Next, let $(h_0, h_1, \dots, h_{2b-1}) \leftarrow \texttt{H}(x)$, and compute the signing key, $sk \leftarrow 2^{b-2} + \sum_{3 \leq i \leq b-3} 2^i h_i$. Finally, compute $vk \leftarrow sk \cdot G$, and return (x, vk).

- $\texttt{Sign}(x, vk, m)$ takes as input a keypair (x, vk) and a message m, and returns a signature σ. Let $r \leftarrow H(h_b, \dots, h_{2b-1}, m)$, and interpret the result as a little-endian integer in $\{0, 1, \dots, 2^{2b} - 1\}$. Let $R \leftarrow r \cdot G$, and $S \leftarrow (r + H(R, A, M) \cdot sk) \mod q$. Return $\sigma \leftarrow (R, S)$.

- $\texttt{Verify}(m, vk, \sigma)$ takes as input a message m, a verification key vk and a signature σ, and returns $r \in \{true, false\}$ depending on whether the signature is valid or not. The algorithm returns *true* if the following equation holds and $false$ otherwise:

$$S \cdot G = R + H(R, vk, m) \cdot vk.$$

3.3 Parameters of instantiation

In this section we set to describe the concrete instantiation of the algorithm presented above. Not only we describe the Curves and Hash functions used, but we also specify how deserialization happens, as this results in important differences of the acceptance criteria of valid signatures. The algorithm we use is Ed25519 [Ber+11]. However, our implementation slightly differs in the deserialization criteria. The details are as follows:

Parameter b: We set $b = 256$

Curve: We define the curve, and by consequence the finite prime order field, security parameter, cofactor, prime order subgroup and generator, as described in [Ber+11]. In particular, we use Edwards25519 which is birationally equivalent to Curve25519 [Ber06].

Hash: As a hashing algorithm we use SHA512 [ST02].

Deserialization: A signature is represented by 64 bytes: the first 32 bytes, $b_{[..32]}$, represent the point R, and the final 32 bytes, $b_{[32..]}$, represent the scalar S. A public key is also represented as 32 bytes, b_{pk}. Deserialization is valid only if:

- $b_{[32..]}$, read as a little-endian integer, is smaller than q.
- $b_{[..32]}$ does not represent a low order point (by checking against a precomputed blacklist of size `cofvar`.
- b_{pk} does not represent a low order point (by checking against a precomputed blacklist of size `cofvar`, and when read as a little-endian integer, it is smaller than p.

3.4 External links

We currently rely in the implementation available in version 1.0.18 of libsodium. For Jormungandr we use a Rust implementation. However, note that this implementation does not check if $b_{[..32]}$ has low order, or if b_{pk} is canonical. This means that the verification criteria for Jormungander is currently different to that of Cardano.

4 Key Evolving Signatures

4.1 Summary

In Cardano, nodes 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.

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. 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. The details of when these keys are evolved are out of the scope of this document, and the reader is directed to the ledger spec [CVG19]. In this section, we describe the detailed cryptographic construction used.

4.2 Generalised specification

We use the iterated sum construction from Section 4.3 of [MMM01]. A KES signature algorithm is parametrized by four algorithms, $\texttt{KeyGen}, \texttt{Sign}, \texttt{Update}$ and $\texttt{Verify}$. The sum construction depends on two signing algorithms, $\Sigma_0 = (\texttt{KeyGen}_0, \texttt{Sign}_0, \texttt{Update}_0, \texttt{Verify}_0)$ and $\Sigma_1 = (\texttt{KeyGen}_1, \texttt{Sign}_1, \texttt{Update}_1, \texttt{Verify}_1)$, which are forward-secure with T_0 and T_1 time periods respectively. As specified in the paper, any digital signature algorithm can be considered as a forward-secure signature algorithm with 1 time period. We use two specialized versions of the key generation, one to generate the public part, and another to generate the secret part, $P\texttt{KeyGen}, S\texttt{KeyGen}$ respectively. The KES algorithm uses a length doubling pseudorandom generator, $F : \{0,1\}^\lambda \rightarrow \{0,1\}^\lambda \times \{0,1\}^\lambda$, where given a random seed, s, returns two random seeds of the same size s_1, s_2. The sum construction begins by generating a signing algorithm with two periods by merging two instances of the base signature, and proceeds recursively until the desired level of periods is reached.

As with the VRF (see Section **??**), the Babbage hard fork brought optimizations to the KES signature size, and we therefore have different verification criteria pre and post-Babbage. Both instances use a tree of depth 6. For ease of exposure, we compare both constructions using a simple binary tree of depth 3 (see Figure **??**). Consider each node as a KES instance with 2^n periods, where n is the height of the node with respect to the leafs. For instance, $\boxed{\text{A}}$ is a KES instance with 8 periods, while node $\boxed{\text{D}}$ is a KES instance with 2 periods. Note that each KES instance is created by two child instances with half the periods. The only relevant details for compatibility on how KES algorithm works is signature generation and signature verification, however we provide a description of the other functions. The details on how the keys are managed are details of implementation which will not be covered here.

4.2.1 Pre-Babbage

The signing instances of nodes $\boxed{\text{H}}$-$\boxed{\text{O}}$ are single period KES instances, and nodes $\boxed{\text{A}}$-$\boxed{\text{G}}$ are defined by recursively calling on the algorithms presented above. In pre-babbage eras the signatures are computed in a naive manner, meaning that a signature is represented (recursively) as the underlying signature and the two public keys, $\sigma = (\sigma', vk_0, vk_1)$. So for instance, the signature in node $\boxed{\text{A}}$ of period 0 contains the signature of node $\boxed{\text{B}}$, σ_b, the public key of node $\boxed{\text{B}}$ vk_b (which is a hash of vk_d and vk_e) and the public key of node $\boxed{\text{C}}$ vk_c (which is a hash of vk_f and vk_g). Signature σ_b in turn contains signature of node $\boxed{\text{D}}$ σ_d, the public key of node $\boxed{\text{D}}$ vk_d (which is a hash of vk_h and vk_i) and public key of node $\boxed{\text{E}}$ vk_e (which is a hash of vk_j and vk_k). The signature of node $\boxed{\text{D}}$, σ_d contains in turn the signature of node $\boxed{\text{H}}$, σ_h, the public key of node $\boxed{\text{H}}$, vk_h and

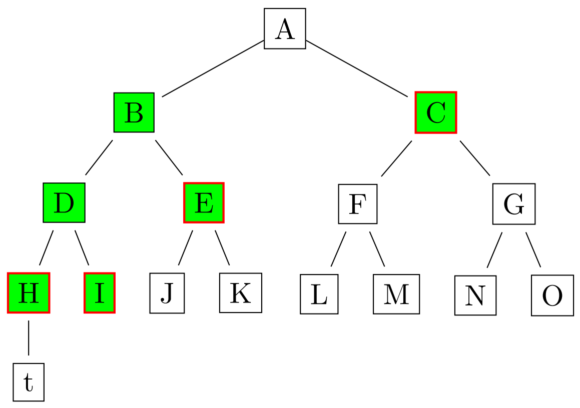

Figure 1: Binary tree of depth 4. For the examples described below, we set the period to zero, $t = 0$. In green, we have the nodes for which we need to store the public key in a signature during pre-babbage eras. With red borders we have the nodes for which we need to store the public keys in post-babbage eras.

the public key of node $\boxed{\text{I}}$ vk_i. Verification of the signature works in the naive recursive manner. We check that $\mathtt{H}(vk_b, vk_c) = vk_a$, and $\mathtt{H}(vk_d, vk_e) = vk_b$, and $\mathtt{H}(vk_h, vk_i) = vk_d$, and finally that $\mathtt{Verify}(m, vk_h, \sigma_h) = \mathtt{true}$. More specifically:

- $\mathtt{KeyGen}(r)$ takes as input a random seed. It then extends the seed into two parts, $(r_0, r_1) \leftarrow F(r)$, and uses each seed to generate the key material of the next layer. In particular $(sk_0, vk_0) \leftarrow \mathtt{KeyGen}(r_0)$ and $vk_1 \leftarrow P\mathtt{KeyGen}(r_1)$. Finally, it computes the pair's public key $vk \leftarrow \mathtt{H}(vk_0, vk_1)$ and returns $(\langle sk_0, r_1, vk_0, vk_1\rangle, vk)$.

- $\mathtt{Sign}(t, \overbrace{\langle sk', r_1, vk_0, vk_1\rangle}^{sk}, m)$ takes as input a time period t, a signing key sk and a message. If $t < T_0$, then it computes the signature using the first signature algorithm, $\sigma' \leftarrow \mathtt{Sign}_0(t, sk', m)$, otherwise it uses the other $\sigma' \leftarrow \mathtt{Sign}_1(t - T_0, sk', m)$. Finally, returns $(\langle \sigma', vk_0, vk_1\rangle, t)$.

- $\mathtt{Update}(t, \overbrace{\langle, sk', r_1, vk_0, vk_1\rangle}^{sk})$ takes as input a time period t, and a signing key sk. If $t + 1 < T_0$, then $sk' \leftarrow \mathtt{Update}_0(t, sk')$. Otherwise, it checks if its changing the key generation algorithm. Specifically, if $t+1 = T_0$, then $sk' \leftarrow S\mathtt{KeyGen}_1(r_1)$ and sets the seed to zero $r_1 \leftarrow 0$. Otherwise, $sk' \leftarrow \mathtt{Update}_1(t - T_0, sk')$.

- $\mathtt{Verify}(vk, m, \overbrace{\langle \sigma', vk_0, vk_1\rangle}^{\sigma}, t)$ takes as input a verification key, vk, a message, m, a signature, σ, and a time period, t. First, it checks that $\mathtt{H}(vk_0, vk_1) = vk$. If that is not the case, it returns $\mathtt{false}$. Otherwise, if $t < T_0$ then $\mathtt{Verify}_0(vk_0, m, \sigma, t)$, else $\mathtt{Verify}_1(vk_1, m, \sigma, t - T_0)$. If verification fails, it returns $\mathtt{false}$, otherwise it returns $\mathtt{true}$.

4.2.2 Post-Babbage

The naive definition of the signature used in Pre-Babbage results in poor performance with respect to the signature size. We don't need to verify the hash equality at each level, and we simply need to do so at the root. In the Babbage hardfork we introduced such an optimization. In particular, instead of storing both public keys in each signature, we only store the one of the branch that we are not in. For the case of the KES instances with 1 period, the KES signature contains not only the underlying signature, but also the public key, which allows us to re-walk the merkle path. Again, assume we are in period 0. Then, the signature in node $\boxed{\text{A}}$ of period 0 contains the signature of node $\boxed{\text{B}}$, σ_b, and the public key of node $\boxed{\text{C}}$ vk_c (which is a hash of vk_f and vk_g). Signature σ_b in turn contains signature of node $\boxed{\text{D}}$ σ_d and public key of node $\boxed{\text{E}}$ vk_e (which is a hash of vk_j and vk_k). The signature of node $\boxed{\text{D}}$, σ_d contains in turn the signature of node $\boxed{\text{H}}$, σ_h, and the public key of node $\boxed{\text{I}}$ vk_i. In this case, $\sigma_h = (\sigma_{0,h}, vk_h)$, where $\sigma_{0,h}$ is the underlying signature. Verification of the signature works going up the tree, rather than down. We check $\texttt{Verify}(m, vk_h, \sigma_{0,h}) = \texttt{true}$. Then we compute the expected key of node $\boxed{\text{D}}$, $vk'_d \leftarrow \texttt{H}(vk_h, vk_i)$. Then we use that to compute the expected key of node $\boxed{\text{B}}$, $vk'_b \leftarrow \texttt{H}(vk'_d, vk_e)$. Finally, we check that the leaf indeed is part of the merkle tree by checking that $\texttt{H}(vk'_b, vk_c) = vk_a$. To derive the missing public key, we introduce a new function, `DeriveVerKey`.

Specifically, post-babbage KES signature algorithm modifies the `Sign` and `Verify` algorithms, and introduces `DeriveVerKey` as follows:

- $\texttt{KeyGen}(r)$ takes as input a random seed. It then extends the seed into two parts, $(r_0, r_1) \leftarrow F(r)$, and uses each seed to generate the key material of the next layer. In particular $(sk_0, vk_0) \leftarrow \texttt{KeyGen}(r_0)$ and $vk_1 \leftarrow P\texttt{KeyGen}(r_1)$. Finally, it computes the pair's public key $vk \leftarrow \texttt{H}(vk_0, vk_1)$ and returns $(\langle sk_0, r_1, vk_0, vk_1\rangle, vk)$.

- $\texttt{Sign}(t, \overbrace{\langle sk', r_1, vk_0, vk_1\rangle}^{sk}, m)$ takes as input a time period t, a signing key sk and a message. If $t < T_0$, then it computes the signature using the first signature algorithm, $\sigma' \leftarrow \texttt{Sign}_0(t, sk', m)$ and lets $vk_c = vk_1$, otherwise it uses the other $\sigma' \leftarrow \texttt{Sign}_1(t - T_0, sk', m)$, and lets $vk_c = vk_0$. Finally, returns $(\langle \sigma', vk\rangle, t)$.

- $\texttt{Update}(t, \overbrace{\langle, sk', r_1, vk_0, vk_1\rangle}^{sk})$ takes as input a time period t, and a signing key sk. If $t + 1 < T_0$, then $sk' \leftarrow \texttt{Update}_0(t, sk')$. Otherwise, it checks if its changing the key generation algorithm. Specifically, if $t + 1 = T_0$, then $sk' \leftarrow S\texttt{KeyGen}_1(r_1)$ and sets the seed to zero $r_1 \leftarrow 0$. Otherwise, $sk' \leftarrow \texttt{Update}_1(t - T_0, sk')$.

- $\texttt{DeriveVerKey}(\overbrace{\langle \sigma', vk_c\rangle}^{\sigma}, m, t)$ takes as input a signature σ and a period t. If $t < T_0$, then $(vk_n, res) = \texttt{DeriveVerKey}(\sigma', m, t)$ and return $(\texttt{H}(vk_n, vk_c), res)$, otherwise $vk_n = \texttt{DeriveVerKey}(\sigma', m, t - T_0)$ and return $(\texttt{H}(vk_c, vk_n), res)$.

- $\texttt{Verify}(vk, m, \overbrace{\langle\sigma', vk_c\rangle}^{\sigma}, t)$ takes as input a verification key, vk, a message, m, a signature, σ, and a time period, t. First, it computes $vk_n \leftarrow \texttt{DeriveVerKey}(\overbrace{\langle\sigma', vk_c\rangle}^{\sigma}, m, t)$, and then, if $t < T_0$, check that $\texttt{H}(vk_n, vk_c) = vk$, otherwise, check that $\texttt{H}(vk_c, vk_n) = vk$. If verification fails, it returns `false`, otherwise it returns `true`.

For this recursive explanation to be complete, we need to define what happens when we call `DeriveVerKey` on a leaf signature. Recall that the signature of a leaf contains not only the underlying signature, but also the public key with which it is signed. The derive function at the leaf takes as input $(\overbrace{\langle\sigma', vk_c\rangle}^{\sigma}, m, t)$. It proceeds by verifying the leaf signature. Parse $\sigma' = (\sigma_0, vk)$, and compute the result $res \leftarrow \texttt{Verify}(m, vk, \sigma_0)$. If $t = 0$, then return $(\texttt{H}(vk, vk_c), res)$, else return $(\texttt{H}(vk_c, vk), res)$.

4.3 Parameters of instantiation

The instantiation of both eras is the same. The underlying signature scheme is Ed25519 (see Section **??**). Regarding P`KeyGen`and S`KeyGen`, in Cardano, these functions simply call a seeded version of Ed25519's `KeyGen` and extract the public or private part. As a hashing function we use Blake2b [Aum15]. Defining the pseudo random function F is not required for compatibility purposes, as it is only used for private key material. However, for sake of completeness, we specify that the cardano node uses Blake2b as a length doubling pseudorandom generator.

4.4 External links

We currently rely in the implementation available in the cardano base. As a layer 0 primitive, it uses Ed25519 (see Section **??**). For Jormungandr we use a Rust implementation. However, note that this implementation is not compatible with that of mainnet. We are actively working in a compatible version in Rust.

5 Verifiable Random Functions

5.1 Summary

Verifiable Random Functions (`VRF`) allow key-pair owners, $(sk_{\texttt{VRF}}, vk_{\texttt{VRF}})$, to evaluate a pseudorandom function in a provable way given a randomness seed. Any party with access to the verification key, $vk_{\texttt{VRF}}$, the randomness seed, the proof and the generated randomness can indeed verify that the value is computed as expected. The VRF specification has changed overtime in Cardano, and the nodes use a different algorith, pre-Babbage and post-Babbage (if there exists a reference to the HF, we should point it out). We expose both specifications, and the motivations for such a change.

5.2 Generalised specification

We use an additional hash function to that introduced in Section **??**. In particular one that maps a binary input to an element of the group $\mathbf{G}$, $\mathtt{H}_{s2c} : \{0,1\}^* \to \mathbf{G}$.

5.2.1 Pre Babbage

A `VRF` function, in pre-Babbage eras, is defined by the following three algorithms:

- $\mathtt{VrfKeyGen}(1^\lambda)$ follows the exact same procedure as $\mathtt{KeyGen}(1^\lambda)$ described in Ed25519, see Section **??**. Output key pair $(x, vk_{\mathtt{VRF}})$. We refer to the signing key sk in the Ed25519 section as $sk_{\mathtt{VRF}}$.

- $\mathtt{GenerateProof}(x, vk_{\mathtt{VRF}}, m)$ takes as input a keypair $(x, vk_{\mathtt{VRF}})$ and a message m, and returns the `VRF` randomness β together with a proof Π. Use x to derive $sk_{\mathtt{VRF}}$. Let $H \leftarrow \mathtt{H}_{s2c}(vk_{\mathtt{VRF}}, m)$. Let $\Gamma \leftarrow sk_{\mathtt{VRF}} \cdot H$. Compute r as defined in procedure `Sign` from Section **??**. Let $c \leftarrow \mathtt{H}(H \mid\mid \Gamma \mid\mid k \cdot G \mid\mid k \cdot H)[..128]$. Compute $s \leftarrow (r + c \cdot sk_{\mathtt{VRF}}) \mod q$. Finally, return the proof $\Pi \leftarrow (\Gamma, c, s)$ and the randomness $\beta \leftarrow \mathtt{H}(\mathtt{suite\_string} \mid\mid 0x03 \mid\mid \mathtt{cofvar} \cdot \Gamma \mid\mid 0x00)$.

- $\mathtt{Verify}(m, vk_{\mathtt{VRF}}, \Pi)$ takes as input a message m, a verification key $vk_{\mathtt{VRF}}$ and a vrf proof Π, and returns β or `false`. It parses the proof as $(\Gamma, c, s) = \Pi$, and computes $H \leftarrow \mathtt{H}_{s2c}(vk_{\mathtt{VRF}}, m)$. Let $U \leftarrow s \cdot G - c \cdot vk_{\mathtt{VRF}}$ and $V \leftarrow s \cdot H - c \cdot \Gamma$. Compute the challenge $c' \leftarrow \mathtt{H}(H \mid\mid \Gamma \mid\mid U \mid\mid V)[..128]$. If $c' = c$, then return $\beta \leftarrow \mathtt{H}(\mathtt{suite\_string} \mid\mid 0x03 \mid\mid \mathtt{cofvar} \cdot \Gamma \mid\mid 0x00)$, otherwise, return `false`.

5.2.2 Post Babbage

A `VRF` function, in post-Babbage eras, is defined by the following three algorithms:

- $\mathtt{VrfKeyGen}(1^\lambda)$ follows the exact same procedure as $\mathtt{KeyGen}(1^\lambda)$ described in Ed25519, see Section **??**. Output key pair $(x, vk_{\mathtt{VRF}})$. We refer to the signing key sk in the Ed25519 section as $sk_{\mathtt{VRF}}$.

- $\mathtt{GenerateProof}(x, vk_{\mathtt{VRF}}, m)$ takes as input a keypair $(x, vk_{\mathtt{VRF}})$ and a message m, and returns the `VRF` randomness β together with a proof Π. Use x to derive $sk_{\mathtt{VRF}}$. Let $H \leftarrow \mathtt{H}_{s2c}(vk_{\mathtt{VRF}}, m)$. Let $\Gamma \leftarrow sk_{\mathtt{VRF}} \cdot H$. Compute r as defined in procedure `Sign` from Section **??**. Let $c \leftarrow \mathtt{H}(H \mid\mid \Gamma \mid\mid k \cdot G \mid\mid k \cdot H)$. Compute $s \leftarrow (r + c \cdot sk_{\mathtt{VRF}}) \mod q$. Finally, return the proof $\Pi \leftarrow (\Gamma, k \cdot G, k \cdot H, s)$ and the randomness $\beta \leftarrow \mathtt{H}(\mathtt{suite\_string} \mid\mid 0x03 \mid\mid \mathtt{cofvar} \cdot \Gamma \mid\mid 0x00)$.

- $\mathtt{Verify}(m, vk_{\mathtt{VRF}}, \Pi)$ takes as input a message m, a verification key $vk_{\mathtt{VRF}}$ and a vrf proof Π, and returns β or `false`. It parses the proof as $(\Gamma, U, V, s) = \Pi$, and computes $H \leftarrow \mathtt{H}_{s2c}(vk_{\mathtt{VRF}}, m)$. Next, compute $c \leftarrow \mathtt{H}(H \mid\mid \Gamma \mid\mid k \cdot G \mid\mid k \cdot H)[..128]$. Finally, if $U = s \cdot G - c \cdot vk_{\mathtt{VRF}}$ and $V = s \cdot H - c \cdot \Gamma$, then return $\beta = \mathtt{H}(\mathtt{suite\_string} \mid\mid 0x03 \mid\mid \mathtt{cofvar} \cdot \Gamma \mid\mid 0x00)$, otherwise, return `false`.

This change allows for batch verification of proofs, which achieve up to a times two improvement in verification time, in exchange of a larger proof.

5.3 Parameters of instantiation

Some of the concrete parameter instantiations also differ between pre-Babbage and post-Babbage eras. We begin by describing those which coincide, and follow with a separate description for the ones that differ.

Parameter $\ell(\kappa)$ and suite `suite_string`: We set $\ell(\kappa) = 512$. We choose the suite `ECVRF_EDWARDS25519_SHA512_ELL2`, as defined in the standard draft [Gol+21]. This sets the parameter `suite_string` as $0x04$ and the following parameters.

Curve: We define the curve, and by consequence the finite prime order field, security parameter, cofactor, prime order subgroup and generator, as described in [Ber+11]. In particular, we use Edwards25519 which is birationally equivalent to Curve25519 [Ber06].

Hash: As a hashing algorithm we use SHA512 [ST02].

5.3.1 Pre Babbage

We proceed with the specifications on pre-Babbage eras.

Draft version: pre-Babbage eras are build on top of Version 03 of the standards draft [Gol+18].

Deserialization: A `VRF` proof is represented by 80 bytes: the first 32 bytes, $b_{[..32]}$, represent the point Γ, the following 16 bytes, $b_{[32..48]}$, represent the scalar c, and the final 32 bytes, $b_{[48..]}$, represent the scalar s. A public key is also represented as 32 bytes, b_{pk}. Deserialization is valid only if:

- $b_{[..32]}$ when read as a little-endian integer, it is smaller than p.
- b_{pk} does not represent a low order point (by checking against a precomputed blacklist of size `cofvar`, and when read as a little-endian integer, it is smaller than p.

Hash to curve $\mathtt{H}_{s2c}$: Elliptor mapping, over a scalar computed by hashing `suite_string` $||$ $0x01$ $||$ $vk_{\mathtt{VRF}}$ $||$ m. The mechanism is described in Section 5.4.1.2 of version 3 of the draft [Gol+18]. Note that for implementing the mechanism as described in the draft, the sign bit is cleared before calling the elligator function, meaning that the latter always works with sign = 0 (see `_vrf_ietfdraft03_hash_to_curve_elligator2_25519` function). See Appendix A.4 of version 3 of the draft for test vectors.

5.3.2 Post Babbage

We finalize with the specifications of post-Babbage eras.

Draft version: post-Babbage eras are build on top of a batch-compatible version of Version 10 of the standards draft [Gol+21]. The specific construction is described in the technical spec studying the security of batch compatibility [Bad+21].

Deserialization: A `VRF` proof is represented by 128 bytes: the first 32 bytes, $b_{[..32]}$, represent the point Γ, the following 32 bytes, $b_{[32..64]}$, represent the point U, the following 32 bytes, $b_{[64..96]}$, represent the point V, and the final 32 bytes, $b_{[96..]}$, represent the scalar s. A public key is also represented as 32 bytes, b_{pk}. Deserialization is valid only if:

- $b_{[..32]}$ when read as a little-endian integer, it is smaller than p.
- $b_{[96..]}$ when read as a little-endian integer, is smaller than q.
- b_{pk} does not represent a low order point (by checking against a precomputed blacklist of size `cofvar`, and when read as a little-endian integer, it is smaller than p.

Hash to curve $\mathtt{H}_{s2c}$: Hash to curve algorithm as defined in the hash to curve standard [FH+21], version 12, Section 6.8.2 (non uniform version). For test vectors, one can use those presented in Appendix J.5.2 of that same document. Reference implementation as called in the libsodium fork.

5.4 External links

We currently rely in the implementation available in the fork of libsodium. For Jormungandr we use a Rust implementation. However, note that this implementation is not compatible with that of mainnet. We are actively working in a compatible version in Rust. However, we rely on the merge of a PR which has been inactive for a while.

References

[Aum15] Jean-Philippe Aumasson. *The BLAKE2 Cryptographic Hash and Message Authentication Code (MAC)*. Informational. Internet Engineering Task Force, 2015. URL: `https://datatracker.ietf.org/doc/html/rfc7693`.

[Bad+21] Christian Badertscher et al. *On UC-Secure Range Extension and Batch Verification for ECVRF*. Technical report. `https://iohk.io/en/research/library/papers/on-uc-secure-range-extension-and-batch-verification-for-ecvrf/`. 2021.

[Ber06] Daniel J. Bernstein. “Curve25519: New Diffie-Hellman Speed Records”. In: 2006, pp. 207–228. DOI: `10.1007/11745853_14`.

[Ber+11] Daniel J. Bernstein et al. “High-Speed High-Security Signatures”. In: 2011, pp. 124–142. DOI: `10.1007/978-3-642-23951-9_9`.

[CVG19] Jared Corduan, Polina Vinogradova, and Matthias Güdemann. *A Formal Specification of the Cardano Ledger*. Deliverable. `https://hydra.iohk.io/build/8201171/download/1/ledger-spec.pdf`. 2019.

[FH+21] Armando Faz-Hernández et al. *Hashing to Elliptic Curves*. Internet-Draft draft-irtf-cfrg-hash-to-curve-13. Work in Progress. Internet Engineering Task Force, Nov. 2021. 175 pp. URL: `https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-hash-to-curve-13`.

[Gol+18] Sharon Goldberg et al. *Verifiable Random Functions (VRFs)*. Internet-Draft draft-irtf-cfrg-vrf-03. Work in Progress. Internet Engineering Task Force, 2018. 42 pp. URL: `https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-vrf-03`.

[Gol+21] Sharon Goldberg et al. *Verifiable Random Functions (VRFs)*. Internet-Draft draft-irtf-cfrg-vrf-10. Work in Progress. Internet Engineering Task Force, Nov. 2021. 42 pp. URL: `https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-vrf-10`.

[MMM01] Tal Malkin, Daniele Micciancio, and Sara Miner. *Composition and Efficiency Tradeoffs for Forward-Secure Digital Signatures*. 2001. URL: `http://eprint.iacr.org/2001/034`.

[ST02] National Institute of Standards and Technology. *Secure Hash Standard (SHS)*. Tech. rep. Federal Information Processing Standards Publications (FIPS PUBS) 180-2. Washington, D.C.: U.S. Department of Commerce, 2002.

Example Section

Summary

This section covers a summary of the properties of the cryptographic primitives. What it can do, and how different it is from similar primitives. For instance, explicitly distinguishing threshold signatures from multi-signatures, from ATMs, etc.

Generalised specification

This section covers the specific instantiation as presented from the reference (either academic paper or standard).

Parameters of instantiation

This section covers the details of the primitive. It should cover all that is required for an external observer to implement a compatible version without needing to look into the code. This should cover, the curve parameters, hashing algorithms, format, serialisation/deserialisation mechanisms, etc.

External links

Links of interest, such as papers, gdocs, jira pages, etc.