$a01b9fe136e5703668c9c7776a76cc8541b84824635d237e270670a8ca56a392$, and the registered addresses were reimbursed in transaction
$8cab8049e3b8c4802d7d11277b21afc74056223a596c39ef00e002c2d1f507ad$.

17.5 Byron redeem addresses

The bootstrap addresses from Figure 6 were not intended to include the Byron era redeem addresses (those with addrtype 2, see the Byron CDDL spec). These addresses were, however, not spendable in the Shelley era. At the Allegra hard fork they were removed from the UTxO and the Ada contained in them was returned to the reserves.

17.6 Block hash used in the epoch nonce

In the CHAIN rule in Figure 75, the TICKN rule uses the last block hash of the previous epoch to create the new epoch nonce. In the implemenation, however, the *penultimate* block hash of the previous epoch is used.

17.7 Deposit tracking

In this specification, individual deposits (for stake credential and stake pool registrations) are not tracked by the ledger. Deposits are returned according to the current protocol parameters. When the values of these two protocol parameters change, the deposoit pot is adjusted by adding to, or removing from, the reserves.

This has several problems:

- Most people expect a deposit to be paid back exactly.
- We cannot increase the deposit amount once the reserves hits zero.
- If it becomes known that the deposit amount is going to be increased, free Lovelace can be earned by registering credentials.
- Because of the problems above, it is going to be incredibly hard to ever change the values.
- There is a serious issue involving hard forks. The consensus layer makes the decision about whether or not to enact a hard fork based on the protocol parameter update state two stability windows before the end of the epoch. However, the ledger will reject a protocol parameter update on the epoch boundary if the deposit pot adjustments cannot be reconciled with the reseve pot. This means that if quorum is met regarding changing the major protocol version, but the update is rejected on the epoch boundary, consensus will change the era but the ledger will not change the major protocol version, leaving the ledger in a split-brain state.

Because we never actually changed the values of the two deposits amounts in the protocol parameters on mainnet, we were able to retroactively change the behavior. We made the following changes:

- Individual deposits are tracked in the DState.
- The amount deposited is always returned.

This was implemented here.

References

[AP10] B. Akbarpour and L. C. Paulson. Metitarski: An automatic theorem prover for real-valued special functions. *J. Autom. Reasoning*, 44(3):175–205, 2010. doi:10.1007/s10817-009-9149-2.

[BC-D1] IOHK Formal Methods Team. Byron Blockchain Specification, IOHK Deliverable BC-D1, 2019. URL `https://github.com/input-output-hk/cardano-ledger/tree/master/docs/`.

[BDN18] D. Boneh, M. Drijvers, and G. Neven. Compact multi-signatures for smaller blockchains. In T. Peyrin and S. D. Galbraith, editors, *Advances in Cryptology - ASIACRYPT 2018 - 24th International Conference on the Theory and Application of Cryptology and Information Security, Brisbane, QLD, Australia, December 2-6, 2018, Proceedings, Part II*, volume 11273 of *Lecture Notes in Computer Science*, pages 435–464. Springer, 2018. doi:10.1007/978-3-030-03329-3_15.

[Bir19] H. Birkholz. Concise Data Definition Language (CDDL). RFC 8610, RFC Editor, 6 2019. URL `https://tools.ietf.org/html/rfc8610`.

[BL-D1] IOHK Formal Methods Team. Byron Ledger Specification, IOHK Deliverable BL-D1, 2019. URL `https://github.com/input-output-hk/cardano-ledger/tree/master/docs/`.

[Bor13] C. Bormann. Concise Binary Object Representation (CBOR). RFC 7049, RFC Editor, 10 2013. URL `https://tools.ietf.org/html/rfc7049`.

[DGKR17] B. M. David, P. Gazi, A. Kiayias, and A. Russell. Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake protocol. *IACR Cryptology ePrint Archive*, 2017:573, 2017.

[DGNW20] M. Drijvers, S. Gorbunov, G. Neven, and H. Wee. Pixel: Multi-signatures for consensus. In S. Capkun and F. Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 2093–2110. USENIX Association, 2020. URL `https://www.usenix.org/conference/usenixsecurity20/presentation/drijvers`.

[FM-TR-2018-01] IOHK Formal Methods Team. Small Step Semantics for Cardano, IOHK Technical Report FM-TR-2018-01, 2018. URL `https://github.com/input-output-hk/cardano-chain/blob/master/specs/semantics/latex/small-step-semantics.tex`.

[Gol20] S. Goldberg. Verifiable Random Functions (VRFs), draft-irtf-cfrg-vrf-06. RFC draft, RFC Editor, 2 2020. URL `https://tools.ietf.org/html/draft-irtf-cfrg-vrf-06`.

[Jos17] S. Josefsson. Edwards-Curve Digital Signature Algorithm (EdDSA). RFC 8032, RFC Editor, 1 2017. URL `https://tools.ietf.org/html/rfc8032`.

[MMM01] T. Malkin, D. Micciancio, and S. Miner. Composition and efficiency tradeoffs for forward-secure digital signatures. Cryptology ePrint Archive, Report 2001/034, 2001. `https://eprint.iacr.org/2001/034`.

[MOR01] S. Micali, K. Ohta, and L. Reyzin. Accountable-subgroup multisignatures: Extended abstract. In *Proceedings of the 8th ACM Conference on Computer and Communications Security*, CCS '01, pages 245–254, New York, NY, USA, 2001. Association for Computing Machinery. doi:10.1145/501983.502017.

[MPSW19] G. Maxwell, A. Poelstra, Y. Seurin, and P. Wuille. Simple schnorr multi-signatures with applications to bitcoin. *Des. Codes Cryptogr.*, 87(9):2139–2164, 2019. doi:10.1007/s10623-019-00608-x.

[NRS20] J. Nick, T. Ruffing, and Y. Seurin. Musig2: Simple two-round schnorr multi-signatures. *IACR Cryptol. ePrint Arch.*, 2020:1261, 2020. URL `https://eprint.iacr.org/2020/1261`.

[Saa15] M.-J. Saarinen. The BLAKE2 cryptographic hash and message authentication code (MAC). RFC 7693, RFC Editor, 11 2015. URL `https://tools.ietf.org/html/rfc7693`.

[SL-D1] IOHK Formal Methods Team. Design Specification for Delegation and Incentives in Cardano, IOHK Deliverable SL-D1, 2018. URL `https://github.com/input-output-hk/cardano-ledger/releases/latest/download/shelley-delegation.pdf`.

[Wui12] P. Wuille. Hierarchical deterministic wallets, February 2012. URL `https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki`. BIP-32.

[Zah18] J. Zahnentferner. Chimeric ledgers: Translating and unifying utxo-based and account-based cryptocurrencies. *Cryptology ePrint Archive, Report 2018/262*, 2018. URL `https://eprint.iacr.org/2018/262`.

A Cryptographic Details

A.1 Warning About Re-serialization

It is a bad security practice to check signatures or hashes on re-serialized data, the original bytes must be preserved.

A.2 Hashing

We present the (informal) properties of cryptographically safe hash functions:

Preimage resistance It should be hard to find a message with a given hash value.

Second preimage resistance Given one message, it should be hard to find another message with the same hash value.

Collision resistance It should be hard to find two messages with the same hash value.

In practice, several cryptographic protocols use hash functions as random oracles. However, the properties of our scheme do not depend on it.

The hashing algorithm we use for all verification keys and multi-signature scripts is BLAKE2b-224[4]. Explicitly, this is the payment and stake credentials (Figure 6), the genesis keys and their delegates (Figure 12), stake pool verification keys (Figure 22), and VRF verification keys (Figure 21). The only property we require of this hash function is that of Second Preimage resistance. Given the hash of verification key or multi-signature script, it should be hard to find a different key or script with the same hash.

Everywhere else we use BLAKE2b-256. In this case we do require second preimage resistance and collision resistance. We use the output of these hash functions to produce signatures (see Figure 20), meaning that if an adversary can find two messages with the same hash, it could break the intended EUF-CMA (Existentially UnForgeable against a Chosen Plaintext Attack) property of the underlying signature scheme.

In the CDDL specification in Appendix D, hash28 refers to BLAKE2b-224 and and hash32 refers to BLAKE2b-256. BLAKE2 is specified in RFC 7693 [Saa15].

A.3 Addresses

The sign and verify functions from Figure 2 use Ed25519. See [Jos17].

A.4 KES

The sign_{ev} and verify_{ev} functions from Figure 3 use the iterated sum construction from Section 3.1 of [MMM01]. We allow up to 2^7 key evolutions, which is larger than the maximum number of evolutions allow by the spec, MaxKESEvo, which will be set to 90. See Figure 66.

A.5 VRF

The verifyVrf function from Figure 5 uses ECVRF-ED25519-SHA512-Elligator2 as described in the draft IETF specification [Gol20].

More generally, we expect the following properties from a Verifyable Random Function:

Full uniqueness For any fixed public VRF key and for *any* input seed, there is a unique VRF output that can be proved valid.

[4]Note that for the signature and verification algorithms, as well as the proving and verification algorithms of VRFs, we use the hash function as defined in the IETF standard (see Section A.3 and Section A.5).

Full collision resistance It should be computationally infeasible to find two distinct VRF inputs that have the same VRF output, even if the adversary knows the private key.

Pseudorandomness Assuming the public-private key pair has been generated honestly, the VRF hash output on any adversarially-chosen VRF input looks indistinguishable from random for a computationally bounded adversary who does not know the private key.

A.6 Abstract functions

- The transaction ID function txid from Figure 10 is implemented as the BLAKE2b-256 hash of the CBOR serialization of the transaction body. **Note** that we do **not** enforce canonical CBOR and that there are multiple valid serializations for any given transaction body. The transaction ID must be computed using the original bytes that were submitted to the network.
- The seed operation $x \star y$ from Figure 5 is implemented as the BLAKE2b-256 hash of the concatenation of x and y.
- The function $\mathsf{slotToSeed}$ from Figure 53 is implemented as the big-endian encoding of the slot number in 8 bytes.

A.7 Multi-Signatures

As presented in Figure 4, Shelley realizes multi-signatures in a native way, via a script-like DSL. One defines the conditions required to validate a multi-signature, and the script takes care of verifying the correctness of the request. It does so in a naïve way, i.e. checking every signature individually. For instance, if the requirement is to have n valid signatures out of some set $\mathcal{V}$ of public keys, the naïve script-based solution checks if: (i) the number of submitted signatures is greater or equal to n, (ii) the distinct verification keys are part of the set $\mathcal{V}$, and (iii) at least n signatures are valid. However, there are more efficient ways to achieve this using more advanced multi-signature schemes, that allow for aggregating both the signatures and the verification procedure. This results in less data to be stored on-chain, and a cheaper verification procedure. Several schemes provide these properties [BDN18, MPSW19, NRS20, DGNW20], and we are currently investigating which would be the best fit for the Cardano ecosystem. We formally introduce multi-signature schemes.

A multi-signature scheme [MOR01] is defined as a tuple of algorithms $\mathsf{MS} = (\mathsf{MS\text{-}Setup}, \mathsf{MS\text{-}KG}, \mathsf{MS\text{-}AVK}, \mathsf{MS\text{-}Sign}, \mathsf{MS\text{-}ASign}, \mathsf{MS\text{-}Verify})$ such that $\Pi \leftarrow \mathsf{MS\text{-}Setup}(1^k)$ generates public parameters—where k is the security parameter. Given the public parameters, one can generate a verification-signing key pair calling, $(\mathsf{vk}, \mathsf{sk}) \leftarrow \mathsf{MS\text{-}KG}(\Pi)$. A multi-signature scheme provides a signing and an aggregate functionality. Mainly

- $\sigma \leftarrow \mathsf{MS\text{-}Sign}(\Pi, \mathsf{sk}, m)$, produces a signature, σ, over a message m using signing key sk, and
- $\tilde{\sigma} \leftarrow \mathsf{MS\text{-}ASig}(\Pi, m, \mathcal{V}, \mathcal{S})$, where given a message m, a set of signatures, $\mathcal{S}$, over the message m, and the corresponding set of verification keys, $\mathcal{V}$, aggregates all signatures into a single, aggregate signature $\tilde{\sigma}$.

To generate an aggregate verification key, the verifier calls the function $\mathsf{avk} \leftarrow \mathsf{MS\text{-}AVK}(\Pi, \mathcal{V})$, which given input a set of verification keys, $\mathcal{V}$, returns an aggregate verification key that can be used for the verification of the aggregate siganture: $\mathsf{MS\text{-}Verify}(\Pi, \mathsf{avk}, m, \tilde{\sigma}) \in \{\mathsf{true}, \mathsf{false}\}$.

Note the distinction between multi-signature schemes (as described above) and multi-signatures as defined in Figure 4. In Figure 4 we allow scripts to consider valid the $\mathsf{RequireAllOf}$, $\mathsf{RequireAnyOf}$ or $\mathsf{RequireMOfN}$ typed signatures. In the definition above, given a set of public