DEER Network

Publicchain audit report

CHAINLION

NO.0C0022071800001

JULY 18, 2022

CATALOGUE

1. Project overview

This report is prepared for deer network public chain. The purpose of this report is to explore the security vulnerabilities and non-standard coding problems in the public chain through the security audit of the public chain source code. The audit cycle of this project is from July 14, 2022 to July 18, 2022. This audit mainly involves the following test methods:

White box test

Conduct security audit on the source code of the public chain, and check the security problems of the coding specification, public chain data layer, network layer, consensus layer, incentive layer, contract layer and application layer.

Grey box test

Deploy the public chain test node locally and conduct fuzzy test to check whether the RPC interface can be called normally, RPC interface permission allocation, RPC interface use permission, RPC interface security design, RPC interface and business interaction, etc

Black box test

Conduct security test attack on the public chain from the perspective of the attacker in combination with the black-and-white box test method to check whether there are exploitable vulnerabilities.

This audit report is subject to the latest public chain code provided by the current project party, and does not include the newly added business logic function modules after the public chain upgrade, new attack methods in the future, and web front-end security and server-side security.

2. Project introduction

The Deer Network is comprised of a vast public chain of globally shared metaverse infrastructure nodes that have the ability to self-repair by interconnecting decentralized storage and computational resources. The Deer Network's Substrate framework is based on the Polkadot ecosystem and is a digital encryption application layer based on both the proof-of-work (PoW) as well as the Nominated Proof of Stake (NPoS) protocols in addition to a new generation of blockchain technology which supports decentralized storage and computation.

The network itself is highly secure, energy-efficient, fair and open. Built upon its decentralized storage network, its condensed consensus base is routed back to its decentralized storage and computing channels via NPoS. This means that it is simple for anyone to contribute and participate in the construction of the Deer Network’s decentralized file system by using any storage devices that they are not currently using for other purposes. Users can thus use their own hardware to contribute to the

efficient, fair, safe and low-cost accessibility of data by providing processing resources in this way.

The flexibility of the PoW protocol means that the Deer Network's efficient design combines not only consensus cohesion + decentralized storage, but also consensus cohesion + decentralized computing. With decentralized storage as its primary characteristic, the Deer Network technically provides the possibility for a full-stack seamless transition of the incentive layer (consensus) + network layer + durability(storage) + application layer (computing).

Official website： https://deernetwork.org

3. Audit summary

3.1. Audit scope

The types and scope of this safety audit mainly include the following aspects：

Coding specification

Data layer security

Network layer security

Consensus layer security

Incentive layer safety

Application Layer Security

Third party dependence on Security

3.2. Audit catalogue

The core code directory involved in this public chain security audit is as follows:

```
├──.github
│   ├──ISSUE_TEMPLATE
│   └──workflows
├──node
│   ├──cli
│   │   ├──bin
│   │   ├──res
│   │   └──src
│   ├──executor
│   │   └──src
│   ├──inspect
│   │   └──src
│   ├──primitives
│   │   └──src
│   ├──rpc
│   │   └──src
│   └──runtime
│       └──src
├──pallets
│   ├──bridge
│   │   └──src
│   ├──bridge-transfer
│   │   └──src
│   ├──nft
│   │   ├──rpc
│   │   │   ├──runtime-api
│   │   │   │   └──src
│   │   │   └──src
│   │   └──src
│   ├──nft-auction
│   │   └──src
```

```
│   ├──nft-order
│   │   └──src
│   └──storage
│       ├──rpc
│       │   ├──runtime-api
│       │   │   └──src
│       │   └──src
│       └──src
├──scripts
├──substrate
└──vendors
    ├──ring
    │   ├──crypto
    │   │   ├──chacha
    │   │   │   └──asm
    │   │   ├──cipher_extra
    │   │   │   ├──asm
    │   │   │   └──test
    │   │   ├──curve25519
    │   │   │   └──asm
    │   │   ├──fipsmodule
    │   │   │   ├──aes
    │   │   │   │   └──asm
    │   │   │   ├──bn
    │   │   │   │   └──asm
    │   │   │   ├──ec
    │   │   │   │   └──asm
    │   │   │   ├──ecdsa
    │   │   │   ├──modes
    │   │   │   │   └──asm
    │   │   │   └──sha
    │   │   │       └──asm
    │   │   ├──limbs
    │   │   ├──perlasm
    │   │   └──poly1305
    │   │       └──asm
```

```
│   ├── doc
│   ├── include
│   │   └── GFp
│   ├── mk
│   ├── pregenerate_asm
│   ├── src
│   │   ├── aead
│   │   ├── arithmetic
│   │   ├── data
│   │   ├── digest
│   │   ├── ec
│   │   │   ├── curve25519
│   │   │   │   └── ed25519
│   │   │   └── suite_b
│   │   │       ├── ecdsa
│   │   │       └── ops
│   │   ├── io
│   │   ├── polyfill
│   │   └── rsa
│   ├── tests
│   ├── third_party
│   │   ├── fiat
│   │   └── NIST
│   │       └── SHAVS
│   └── util
│       └── ar
│           └── testdata
│               ├── linux
│               ├── mac
│               ├── sample
│               └── windows
└── webpki
    ├── mk
    ├── src
    │   └── data
    └── tests
```

```
│   ├──ed25519
│   ├──misc
│   └──netflix
└──third-party
    └──chromium
        └──data
            └──verify_signed_data
                └──ours
```

3.3. Third party Library

Futures 0.3.16
hex-literal 0.3.4
log 0.4.8
rand 0.7.2
structopt 0.3.25
parking_lot 0.11.1
futures 0.3.16
tempfile 3.1.0
assert_cmd 2.0.2
nix 0.23
serde_json 1.0
regex 1
platforms 2.0
async-std 1.10.0
soketto 0.4.2
criterion 0.3.5
tokio 1.13
jsonrpsee-ws-client 0.4.1
wait-timeout 0.2
structopt 0.3.25
scale-info 1.0
blake2-rfc 0.2.18
codec 2.2.0
scale-info 1.0
hex 0.4
hex-literal 0.3.4
enumflags2 0.6.3

scale-info 1.0 base64 0.13.0 untrusted 0.7.0 winapi 0.3.7 wasm-bindgen-test 0.2.48 ……

4. Audit results

4.1. Vulnerability statistics

The audit results are as follows:

Entry type	Specific description
Serious vulnerability	0
High risk vulnerability	0
Moderate risk	0
Low risk vulnerability	0

Security vulnerability rating description:

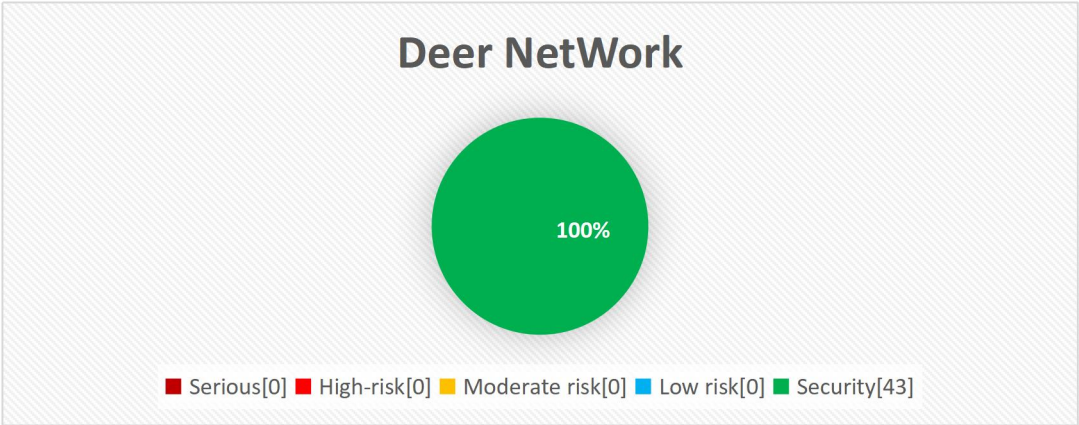
1) **Serious vulnerability:** It can cause security risks such as chain bifurcation, transaction forgery, P2P node pollution, data tampering, transaction replay attacks through certain attack techniques.

2) **High risk vulnerability:** It can lead to security risks such as node program crash, private key disclosure, Web UI authentication bypass, and node sensitive information disclosure.

3) **Moderate risk：** Security problems caused by unreasonable business logic design, such as token allocation caused by defective incentive mechanism design, encrypted storage of sensitive information, lack of authentication mechanism when adding, deleting, modifying and checking accounts through RPC, and other security risks.

4) **Low risk vulnerability：** Security vulnerabilities that require special conditions to trigger, such as obtaining RPC authentication information and sensitive information after interacting with nodes.

4.2. Vulnerability distribution



5. Audit content

5.1. Coding specification

Coding specification is a basic skill that developers must master. On the one hand, standardized coding can avoid some common security problems, on the other hand, it can improve the readability of code, and provide convenience for later code reconstruction, code update, code debugging and code analysis.

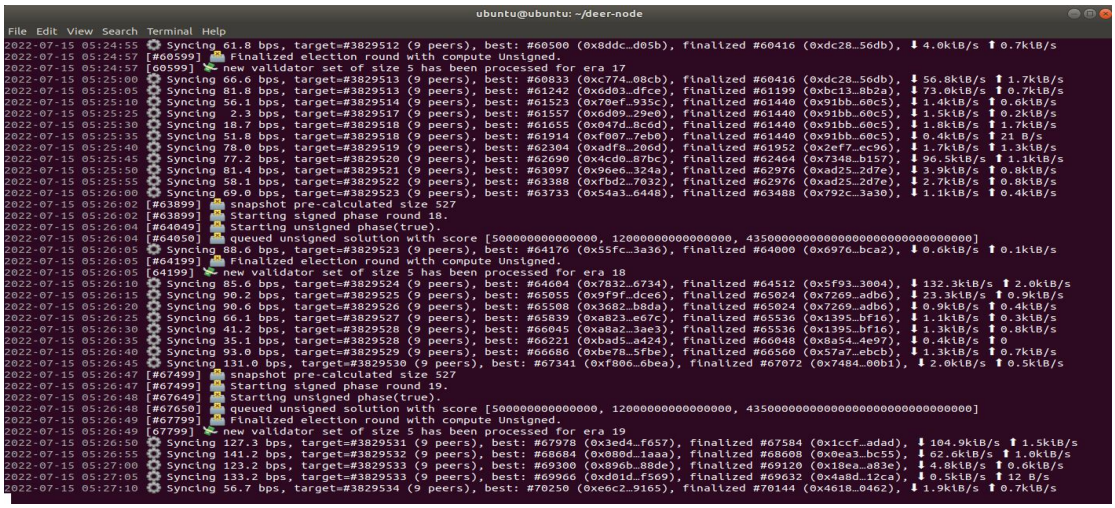
5.1.1. Sensitive information log printing 【security】

Audit description： Check whether there is user authentication information entered by the user through the interactive end (web and CLI) in the public chain source code, such as Web UI unlock authentication password, private key and other information, which is printed in the form of log.

Risk hazard： The leakage of sensitive information may endanger user assets in serious cases.

Audit process： Check whether there is sensitive information printed in the public chain source code to the log, or check whether there is sensitive information in the log information printed after the node runs.

Audit results: After audit, no sensitive information printing operation was found in the public chain source code, and no relevant sensitive log records were found in the console after the node was running.



Safety advice：NONE.

5.1.2. Unhandled exception information【security】

Audit description： Check whether the exception information is handled in the public chain source code. For example, when the user enters illegal data information, whether all kinds of special conditions of parameters are considered and the exception information is handled, instead of constantly throwing up or not doing any exception handling.

Risk hazard： Sensitive information is leaked, and the node collapses.

Audit process：Check the processing logic design for abnormal conditions in the

public chain source code.

Audit results： After audit, there is no such safety problem.

Safety advice：NONE.

5.1.3. Variable function definition declaration【security】

Audit description： Check whether the declaration of variable functions in the public chain source code is simple and readable.

Risk hazard： Security issues such as variable coverage and null pointer

Audit process: Check the variable function declaration specification in the public chain source code.

Audit results： After audit, there is no such safety problem.

Safety advice：NONE.

5.2. Data layer security

The data layer is the underlying technology of blockchain, which mainly includes two functions: data storage, implementation and security of accounts and transactions.

5.2.1. Block data structure design【security】

Audit description： Check whether the block data structure design is

reasonable in the block generation process.

Risk hazard： Block forgery, wrong data type and other security issues.

Audit process： Check whether the block data structure design is reasonable in the block generation process.

Audit results： After audit, there is no such safety problem.

```
pub struct Block<Header, Extrinsic: MaybeSerialize> {
	/// The block header.
	pub header: Header,
	/// The accompanying extrinsics.
	pub extrinsics: Vec<Extrinsic>,
}
pub struct Header<Number: Copy + Into<U256> + TryFrom<U256>, Hash: HashT> {
	/// The parent hash.
	pub parent_hash: Hash::Output,
	/// The block number.
	#[cfg_attr(
		feature = "std",
		serde(serialize_with = "serialize_number", deserialize_with = "deserialize_number")
	)]
	#[codec(compact)]
	pub number: Number,
	/// The state trie merkle root
	pub state_root: Hash::Output,
	/// The merkle root of the extrinsics.
	pub extrinsics_root: Hash::Output,
	/// A chain-specific digest of data useful for light clients or referencing auxiliary data.
	pub digest: Digest,
}
```

Safety advice：NONE.

5.2.2. Block processing resource restrictions 【security】

Audit description： Check whether the resource restrictions such as orphan block pool, verification calculation and hard disk addressing are reasonable.

Risk hazard：CPU resource consumption leads to security problems such as node crash

Audit process： Check whether the resource restrictions such as orphan block pool, verification calculation and hard disk addressing are reasonable.

Audit results： After audit, there is no such safety problem.

Safety advice：NONE.

5.2.3. Merkle Tree Root 【security】

Audit description： Check whether the update, construction, search and other designs of Merkle tree root are reasonable.

Risk hazard： Double flower attack, chain bifurcation and other security issues.

Audit process： Check whether the update, construction, search and other designs of Merkle tree root are reasonable.

Audit results： After audit, there is no such safety problem.

```
/// Generate trie from given `key_values`.
///
```

```
/// Will fill your database `db` with trie data from `key_values` and
/// return root.
pub fn generate_trie(
	db: Arc<dyn KeyValueDB>,
	key_values: impl IntoIterator<Item = (Vec<u8>, Vec<u8>)>,
) -> Hash {
	let mut root = Hash::default();

	let (db, overlay) = {
		let mut overlay = HashMap::new();
		overlay.insert(

	hex::decode("03170a2e7597b7b7e3d84c05391d139a62b157e78786d8c082f29dcf4c111314
")
				.expect("null key is valid"),
			Some(vec![0]),
		);
		let mut trie = SimpleTrie { db, overlay: &mut overlay };
		{
			let mut trie_db = TrieDBMutV1::<crate::simple_trie::Hasher>::new(&mut trie,
&mut root);
			for (key, value) in key_values {
				trie_db.insert(&key, &value).expect("trie insertion failed");
			}

			trie_db.commit();
		}
		(trie.db, overlay)
	};

	let mut transaction = db.transaction();
	for (key, value) in overlay.into_iter() {
		match value {
			Some(value) => transaction.put(0, &key[..], &value[..]),
			None => transaction.delete(0, &key[..]),
		}
```

```
        }
        db.write(transaction).expect("Failed to write transaction");

        root
}
impl<'a> HashDB<Hasher, DBValue> for SimpleTrie<'a> {
        fn get(&self, key: &Hash, prefix: Prefix) -> Option<DBValue> {
                let key = sp_trie::prefixed_key::<Hasher>(key, prefix);
                if let Some(value) = self.overlay.get(&key) {
                        return value.clone()
                }
                self.db.get(0, &key).expect("Database backend error")
        }

        fn contains(&self, hash: &Hash, prefix: Prefix) -> bool {
                self.get(hash, prefix).is_some()
        }

        fn insert(&mut self, prefix: Prefix, value: &[u8]) -> Hash {
                let key = Hasher::hash(value);
                self.emplace(key, prefix, value.to_vec());
                key
        }
        fn emplace(&mut self, key: Hash, prefix: Prefix, value: DBValue) {
                let key = sp_trie::prefixed_key::<Hasher>(&key, prefix);
                self.overlay.insert(key, Some(value));
        }
        fn remove(&mut self, key: &Hash, prefix: Prefix) {
                let key = sp_trie::prefixed_key::<Hasher>(key, prefix);
                self.overlay.insert(key, None);
        }
}
```

Safety advice：NONE.

5.2.4. Generation and use of random numbers【security】

Audit description： Check whether the key random number generation algorithm is reasonable, and test whether the real probability of key random number generation meets the requirements (normal uniform distribution).

Risk hazard： Security risks such as disclosure of sensitive information and arbitrary issuance of transactions.

Audit process： Check whether the business logic design related to random numbers in the public chain source code is reasonable.

Audit results： After audit, the generation of random numbers is reasonable, and there is no such security problem.

```
/// Generates a random scalar in the range [1, n).
pub fn random_scalar(
        ops: &PrivateKeyOps,
        rng: &dyn rand::SecureRandom,
) -> Result<Scalar, error::Unspecified> {
        let num_limbs = ops.common.num_limbs;
        let mut bytes = [0; ec::SCALAR_MAX_BYTES];
        let bytes = &mut bytes[..(num_limbs * LIMB_BYTES)];
        generate_private_scalar_bytes(ops, rng, bytes)?;
        scalar_from_big_endian_bytes(ops, bytes)
}
pub fn generate_private_scalar_bytes(
        ops: &PrivateKeyOps,
        rng: &dyn rand::SecureRandom,
        out: &mut [u8],
```

```
) -> Result<(), error::Unspecified> {
    // [NSA Suite B Implementer's Guide to ECDSA] Appendix A.1.2, and
    // [NSA Suite B Implementer's Guide to NIST SP 800-56A] Appendix B.2,
    // "Key Pair Generation by Testing Candidates".
    //
    // [NSA Suite B Implementer's Guide to ECDSA]: doc/ecdsa.pdf.
    // [NSA Suite B Implementer's Guide to NIST SP 800-56A]: doc/ecdh.pdf.

    // TODO: The NSA guide also suggests, in appendix B.1, another mechanism
    // that would avoid the need to use `rng.fill()` more than once. It works
    // by generating an extra 64 bits of random bytes and then reducing the
    // output (mod n). Supposedly, this removes enough of the bias towards
    // small values from the modular reduction, but it isn't obvious that it is
    // sufficient. TODO: Figure out what we can do to mitigate the bias issue
    // and switch to the other mechanism.

    let candidate = out;

    // XXX: The value 100 was chosen to match OpenSSL due to uncertainty of
    // what specific value would be better, but it seems bad to try 100 times.
    for _ in 0..100 {
        // NSA Guide Steps 1, 2, and 3.
        //
        // Since we calculate the length ourselves, it is pointless to check
        // it, since we can only check it by doing the same calculation.

        // NSA Guide Step 4.
        //
        // The requirement that the random number generator has the
        // requested security strength is delegated to `rng`.
        rng.fill(candidate)?;

        // NSA Guide Steps 5, 6, and 7.
        if check_scalar_big_endian_bytes(ops, candidate).is_err() {
            continue;
        }
```

```
            // NSA Guide Step 8 is done in `public_from_private()`.

            // NSA Guide Step 9.
            return Ok(());
        }
        Err(error::Unspecified)
}
pub fn check_scalar_big_endian_bytes(
        ops: &PrivateKeyOps,
        bytes: &[u8],
) -> Result<(), error::Unspecified> {
        debug_assert_eq!(bytes.len(), ops.common.num_limbs * LIMB_BYTES);
        scalar_from_big_endian_bytes(ops, bytes).map(|_| ())
}
// Parses a fixed-length (zero-padded) big-endian-encoded scalar in the range
// [1, n). This is constant-time with respect to the actual value *only if* the
// value is actually in range. In other words, this won't leak anything about a
// valid value, but it might leak small amounts of information about an invalid
// value (which constraint it failed).
pub fn scalar_from_big_endian_bytes(
        ops: &PrivateKeyOps,
        bytes: &[u8],
) -> Result<Scalar, error::Unspecified> {
        // [NSA Suite B Implementer's Guide to ECDSA] Appendix A.1.2, and
        // [NSA Suite B Implementer's Guide to NIST SP 800-56A] Appendix B.2,
        // "Key Pair Generation by Testing Candidates".
        // [NSA Suite B Implementer's Guide to ECDSA]: doc/ecdsa.pdf.
        // [NSA Suite B Implementer's Guide to NIST SP 800-56A]: doc/ecdh.pdf.
        // Steps 5, 6, and 7.
        // XXX: The NSA guide says that we should verify that the random scalar is
        // in the range [0, n - 1) and then add one to it so that it is in the range
        // [1, n). Instead, we verify that the scalar is in the range [1, n). This
        // way, we avoid needing to compute or store the value (n - 1), we avoid the
        // need to implement a function to add one to a scalar, and we avoid needing
        // to convert the scalar back into an array of bytes.
```

```
        scalar_parse_big_endian_fixed_consttime(ops.common, untrusted::Input::from(bytes))
}
#[inline]
pub fn scalar_parse_big_endian_fixed_consttime(
        ops: &CommonOps,
        bytes: untrusted::Input,
) -> Result<Scalar, error::Unspecified> {
        parse_big_endian_fixed_consttime(ops,                bytes,                AllowZero::No,
&ops.n.limbs[..ops.num_limbs])
}
fn parse_big_endian_fixed_consttime<M>(
        ops: &CommonOps,
        bytes: untrusted::Input,
        allow_zero: AllowZero,
        max_exclusive: &[Limb],
) -> Result<elem::Elem<M, Unencoded>, error::Unspecified> {
        if bytes.len() != ops.num_limbs * LIMB_BYTES {
                return Err(error::Unspecified);
        }
        let mut r = elem::Elem::zero();
        parse_big_endian_in_range_and_pad_consttime(
                bytes,
                allow_zero,
                max_exclusive,
                &mut r.limbs[..ops.num_limbs],
        )?;
        Ok(r)
}
/// Parses `input` into `result`, verifies that the value is less than
/// `max_exclusive`, and pads `result` with zeros to its length. If `allow_zero`
/// is not `AllowZero::Yes`, zero values are rejected.
/// This attempts to be constant-time with respect to the actual value *only if*
/// the value is actually in range. In other words, this won't leak anything
/// about a valid value, but it might leak small amounts of information about an
/// invalid value (which constraint it failed).
pub fn parse_big_endian_in_range_and_pad_consttime(
```

```
    input: untrusted::Input,
    allow_zero: AllowZero,
    max_exclusive: &[Limb],
    result: &mut [Limb],
) -> Result<(), error::Unspecified> {
    parse_big_endian_and_pad_consttime(input, result)?;
    if limbs_less_than_limbs_consttime(&result, max_exclusive) != LimbMask::True {
        return Err(error::Unspecified);
    }
    if allow_zero != AllowZero::Yes {
        if limbs_are_zero_constant_time(&result) != LimbMask::False {
            return Err(error::Unspecified);
        }
    }
    Ok(())
}
```

Safety advice：NONE.

5.2.5. Implementation / use of cryptographic algorithm 【security】

Audit description： Check whether the cryptographic use or implementation of signature, hash and verification is reasonable.

Risk hazard： Security issues such as transaction forgery and chain forking.

Audit process：Check whether the cryptographic algorithms used in the signing, hashing and verification of the public chain source code meet the security requirements.

Audit results： After audit, the public chain of deer network uses sha256

encryption algorithm and ECDSA elliptic curve encryption algorithm, which is relatively safe.

Safety advice：NONE.

5.2.6. Private key / mnemonic generation algorithm 【security】

Audit description： Check whether the logic of the private key generation algorithm is reasonable, and test whether the complexity bottleneck meets the expectation.

Risk hazard： Leakage of sensitive information, arbitrary issuance of transactions, affecting asset security, etc.

Audit process：Check whether the cryptographic algorithms used in the signing, hashing and verification of the public chain source code meet the security requirements.

Audit results：After audit, the public chain of deer network uses ed25519 elliptic curve encryption algorithm and sr25519 encryption algorithm, which can meet the security requirements.

```
/// An Ed25519 key pair, for signing.
pub struct Ed25519KeyPair {
    // RFC 8032 Section 5.1.6 calls this *s*.
    private_scalar: Scalar,
```

```
        // RFC 8032 Section 5.1.6 calls this *prefix*.
        private_prefix: Prefix,

        // RFC 8032 Section 5.1.5 calls this *A*.
        public_key: PublicKey,
}
impl Ed25519KeyPair {
        /// Generates a new key pair and returns the key pair serialized as a
        /// PKCS#8 document.
        ///
        /// The PKCS#8 document will be a v2 `OneAsymmetricKey` with the public key,
        /// as described in [RFC 5958 Section 2]. See also
        /// https://tools.ietf.org/html/draft-ietf-curdle-pkix-04.
        ///
        /// [RFC 5958 Section 2]: https://tools.ietf.org/html/rfc5958#section-2
        pub fn generate_pkcs8(
                rng: &dyn rand::SecureRandom,
        ) -> Result<pkcs8::Document, error::Unspecified> {
                let seed: [u8; SEED_LEN] = rand::generate(rng)?.expose();
                let key_pair = Self::from_seed_(&seed);
                Ok(pkcs8::wrap_key(
                        &PKCS8_TEMPLATE,
                        &seed[..],
                        key_pair.public_key().as_ref(),
                ))
        }

        /// Constructs an Ed25519 key pair by parsing an unencrypted PKCS#8 v2
        /// Ed25519 private key.
        ///
        /// `openssl genpkey -algorithm ED25519` generates PKCS# v1 keys, which
        /// require the use of `Ed25519KeyPair::from_pkcs8_maybe_unchecked()`
        /// instead of `Ed25519KeyPair::from_pkcs8()`.
        ///
        /// The input must be in PKCS#8 v2 format, and in particular it must contain
        /// the public key in addition to the private key. `from_pkcs8()` will
```

```
/// verify that the public key and the private key are consistent with each
/// other.
///
/// If you need to parse PKCS#8 v1 files (without the public key) then use
/// `Ed25519KeyPair::from_pkcs8_maybe_unchecked()` instead.
pub fn from_pkcs8(pkcs8: &[u8]) -> Result<Self, error::KeyRejected> {
    let (seed, public_key) =
        unwrap_pkcs8(pkcs8::Version::V2Only, untrusted::Input::from(pkcs8))?;
    Self::from_seed_and_public_key(
        seed.as_slice_less_safe(),
        public_key.unwrap().as_slice_less_safe(),
    )
}

/// Constructs an Ed25519 key pair by parsing an unencrypted PKCS#8 v1 or v2
/// Ed25519 private key.
///
/// `openssl genpkey -algorithm ED25519` generates PKCS# v1 keys.
///
/// It is recommended to use `Ed25519KeyPair::from_pkcs8()`, which accepts
/// only PKCS#8 v2 files that contain the public key.
/// `from_pkcs8_maybe_unchecked()` parses PKCS#2 files exactly like
/// `from_pkcs8()`. It also accepts v1 files. PKCS#8 v1 files do not contain
/// the public key, so when a v1 file is parsed the public key will be
/// computed from the private key, and there will be no consistency check
/// between the public key and the private key.
///
/// PKCS#8 v2 files are parsed exactly like `Ed25519KeyPair::from_pkcs8()`.
pub fn from_pkcs8_maybe_unchecked(pkcs8: &[u8]) -> Result<Self, error::KeyRejected> {
    let (seed, public_key) =
        unwrap_pkcs8(pkcs8::Version::V1OrV2, untrusted::Input::from(pkcs8))?;
    if let Some(public_key) = public_key {
        Self::from_seed_and_public_key(
            seed.as_slice_less_safe(),
            public_key.as_slice_less_safe(),
        )
```

```
        } else {
            Self::from_seed_unchecked(seed.as_slice_less_safe())
        }
    }

    /// Constructs an Ed25519 key pair from the private key seed `seed` and its
    /// public key `public_key`.
    ///
    /// It is recommended to use `Ed25519KeyPair::from_pkcs8()` instead.
    ///
    /// The private and public keys will be verified to be consistent with each
    /// other. This helps avoid misuse of the key (e.g. accidentally swapping
    /// the private key and public key, or using the wrong private key for the
    /// public key). This also detects any corruption of the public or private
    /// key.
    pub fn from_seed_and_public_key(
        seed: &[u8],
        public_key: &[u8],
    ) -> Result<Self, error::KeyRejected> {
        let pair = Self::from_seed_unchecked(seed)?;

        // This implicitly verifies that `public_key` is the right length.
        // XXX: This rejects ~18 keys when they are partially reduced, though
        // those keys are virtually impossible to find.
        if public_key != pair.public_key.as_ref() {
            let err = if public_key.len() != pair.public_key.as_ref().len() {
                error::KeyRejected::invalid_encoding()
            } else {
                error::KeyRejected::inconsistent_components()
            };
            return Err(err);
        }

        Ok(pair)
    }
```

```
/// Constructs a Ed25519 key pair from the private key seed `seed`.
///
/// It is recommended to use `Ed25519KeyPair::from_pkcs8()` instead. When
/// that is not practical, it is recommended to use
/// `Ed25519KeyPair::from_seed_and_public_key()` instead.
///
/// Since the public key is not given, the public key will be computed from
/// the private key. It is not possible to detect misuse or corruption of
/// the private key since the public key isn't given as input.
pub fn from_seed_unchecked(seed: &[u8]) -> Result<Self, error::KeyRejected> {
    let seed = seed
        .try_into()
        .map_err(|_| error::KeyRejected::invalid_encoding())?;
    Ok(Self::from_seed_(seed))
}

fn from_seed_(seed: &Seed) -> Self {
    let h = digest::digest(&digest::SHA512, seed);
    let (private_scalar, private_prefix) = h.as_ref().split_at(SCALAR_LEN);

    let private_scalar =
        MaskedScalar::from_bytes_masked(private_scalar.try_into().unwrap()).into();

    let mut a = ExtPoint::new_at_infinity();
    unsafe {
        GFp_x25519_ge_scalarmult_base(&mut a, &private_scalar);
    }

    Self {
        private_scalar,
        private_prefix: private_prefix.try_into().unwrap(),
        public_key: PublicKey(a.into_encoded_point()),
    }
}

/// Returns the signature of the message `msg`.
```

```
    pub fn sign(&self, msg: &[u8]) -> signature::Signature {
        signature::Signature::new(|signature_bytes| {
            extern "C" {
                fn GFp_x25519_sc_muladd(
                    s: &mut [u8; SCALAR_LEN],
                    a: &Scalar,
                    b: &Scalar,
                    c: &Scalar,
                );
            }

            let (signature_bytes, _unused) = signature_bytes.split_at_mut(ELEM_LEN +
SCALAR_LEN);
            let (signature_r, signature_s) = signature_bytes.split_at_mut(ELEM_LEN);
            let nonce = {
                let mut ctx = digest::Context::new(&digest::SHA512);
                ctx.update(&self.private_prefix);
                ctx.update(msg);
                ctx.finish()
            };
            let nonce = Scalar::from_sha512_digest_reduced(nonce);

            let mut r = ExtPoint::new_at_infinity();
            unsafe {
                GFp_x25519_ge_scalarmult_base(&mut r, &nonce);
            }
            signature_r.copy_from_slice(&r.into_encoded_point());
            let hram_digest = eddsa_digest(signature_r, &self.public_key.as_ref(), msg);
            let hram = Scalar::from_sha512_digest_reduced(hram_digest);
            unsafe {
                GFp_x25519_sc_muladd(
                    signature_s.try_into().unwrap(),
                    &hram,
                    &self.private_scalar,
                    &nonce,
                );
```

```
                }

                SIGNATURE_LEN
            })
        }
}

impl signature::KeyPair for Ed25519KeyPair {
        type PublicKey = PublicKey;

        fn public_key(&self) -> &Self::PublicKey {
                &self.public_key
        }
}

#[derive(Clone, Copy)]
pub struct PublicKey([u8; ED25519_PUBLIC_KEY_LEN]);

impl AsRef<[u8]> for PublicKey {
        fn as_ref(&self) -> &[u8] {
                self.0.as_ref()
        }
}
type Seed = [u8; SEED_LEN];
const SEED_LEN: usize = 32;

static PKCS8_TEMPLATE: pkcs8::Template = pkcs8::Template {
        bytes: include_bytes!("ed25519_pkcs8_v2_template.der"),
        alg_id_range: core::ops::Range { start: 7, end: 12 },
        curve_id_index: 0,
        private_key_index: 0x10,
};
/// Verification of [Ed25519] signatures.
///
/// Ed25519 uses SHA-512 as the digest algorithm.
///
```

```
/// [Ed25519]: https://ed25519.cr.yp.to/
pub static ED25519: EdDSAParameters = EdDSAParameters {};

impl signature::VerificationAlgorithm for EdDSAParameters {
    fn verify(
        &self,
        public_key: untrusted::Input,
        msg: untrusted::Input,
        signature: untrusted::Input,
    ) -> Result<(), error::Unspecified> {
        let public_key: &[u8; ELEM_LEN] = public_key.as_slice_less_safe().try_into()?;
        let (signature_r, signature_s) = signature.read_all(error::Unspecified, |input| {
            let signature_r: &[u8; ELEM_LEN] = input
                .read_bytes(ELEM_LEN)?
                .as_slice_less_safe()
                .try_into()?;
            let signature_s: &[u8; SCALAR_LEN] = input
                .read_bytes(SCALAR_LEN)?
                .as_slice_less_safe()
                .try_into()?;
            Ok((signature_r, signature_s))
        })?;
        let signature_s = Scalar::from_bytes_checked(*signature_s)?;

        let mut a = ExtPoint::from_encoded_point_vartime(public_key)?;
        a.invert_vartime();
        let h_digest = eddsa_digest(signature_r, public_key, msg.as_slice_less_safe());
        let h = Scalar::from_sha512_digest_reduced(h_digest);

        let mut r = Point::new_at_infinity();
        unsafe { GFp_x25519_ge_double_scalarmult_vartime(&mut r, &h, &a, &signature_s) };
        let r_check = r.into_encoded_point();
        if *signature_r != r_check {
            return Err(error::Unspecified);
        }
        Ok(())
```

```
    }
}
```

Safety advice：NONE.

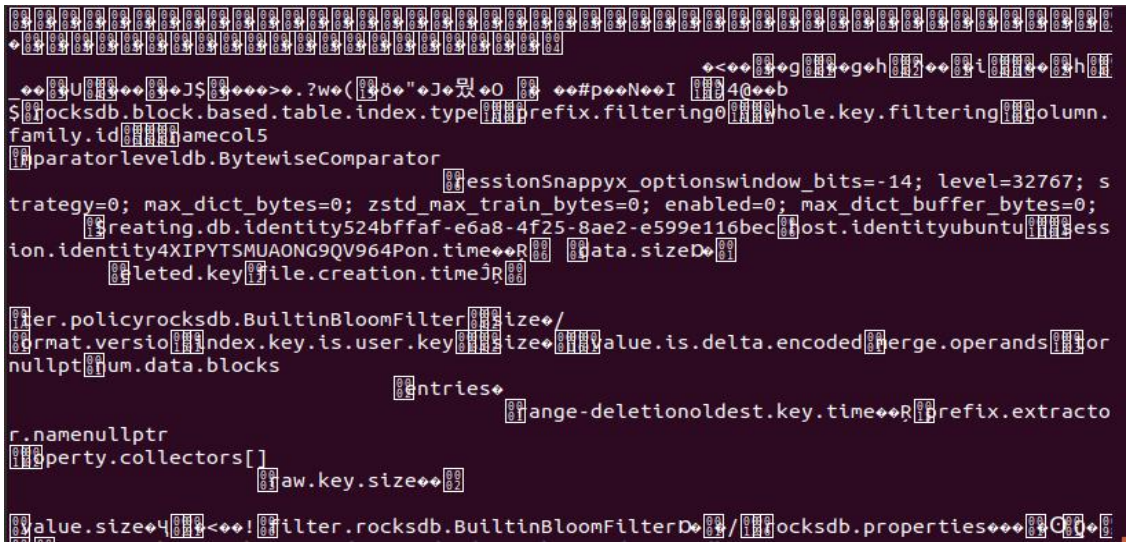
5.2.7. Private key / mnemonic plaintext storage【security】

Audit description： Check the storage mode of the private key and whether the file is encrypted or can be decrypted.

Risk hazard ： Leakage of sensitive information, arbitrary issuance of transactions, affecting asset security, etc.

Audit process：Check whether the private key / mnemonic is stored in clear text.

Audit results： After audit, the private key / mnemonic is encrypted and stored, and there is no such security problem.



Safety advice：NONE.

5.2.8. Private key / mnemonic usage trace 【security】

Audit description： Check whether the traces of private key / mnemonic words in code logic are cleaned up in time after use, and whether traces that can leak information can be found in logs and memory dumps after sensitive data is used.

Risk hazard： Cooperating with other vulnerabilities can lead to private key disclosure, affect asset security, and in serious cases, lead to chain forking and other problems.

Audit process： Check whether the use of private key / mnemonic words in the public chain code conforms to the security operation specifications.

Audit results： After audit, there is no such safety problem.

Safety advice：NONE.

5.3. Network layer security

The network layer encapsulates the networking mode, message propagation protocol, data verification mechanism and other elements of the blockchain system. By designing specific propagation protocols and data verification mechanisms, each node in the blockchain system can participate in the verification and accounting process of block data, and the block data can be recorded in the blockchain only after it has been verified by most nodes in the whole network.

5.3.1. Node discovery algorithm 【security】

Audit description： Check whether the node discovery algorithm is balanced and unpredictable, for example, the distance algorithm is unbalanced.

Risk hazard： Node pollution and other safety risks.

Audit process： Check whether the node discovery algorithm is balanced and unpredictable.

Audit results： After audit, there is no such safety problem.

Safety advice：NONE.

5.3.2. Node communication protocol 【security】

Audit description： Check whether there is a design / implementation layer problem in the communication protocol between nodes, and test the node's response to abnormal packets by fuzzy testing.

Risk hazard： Security risks such as transaction propagation and node pollution.

Audit process： Check whether there is a design / implementation layer problem in the communication protocol between nodes, and test the node's response to abnormal packets by fuzzy testing.

Audit results： After audit, deernetwork uses gossip protocol, and there is no such security problem.

Safety advice：NONE.

5.3.3. Communication flow limit 【security】

Audit description： Check the restrictions on the number of connections and packet size in various P2P protocols in the code logic, try to establish a large number of connections with the target node, and construct and send packets of various volumes.

Risk hazard： Node DoS attack and other security risks

Audit process: Check the restrictions on the number of connections and packet size in various P2P protocols in the code logic, try to establish a large number of connections with the target node, and construct and send packets of various volumes.

Audit results: According to the audit, deer network has limited the communication flow, so there is no such security problem.

Safety advice: NONE.

5.3.4. Node penalty mechanism 【security】

Audit description: Check whether the design of node punishment mechanism is reasonable.

Risk hazard: Security issues such as solar eclipse attack, asset double blossom, chain bifurcation, etc.

Audit process: Check whether the design of node punishment mechanism in public chain code is reasonable.

Audit results: After audit, there is no such safety problem.

Safety advice: NONE.

5.3.5. Sensitive information differentiation response 【security】

Audit description： Check the response data logic of various responses, and test whether there are different responses to sensitive information in various responses.

Risk hazard： Disclosure of sensitive information.

Audit process： Check whether the P2P connection response function in the public chain code only responds to the MSG protocol that meets the standard.

Audit results： After audit, there is no such safety problem.

Safety advice：NONE.

5.4. Consensus layer security

Consensus mechanism is the cornerstone of building trust in blockchains. Different types of blockchains will choose different consensus algorithms or adopt a combination of multiple consensus algorithms according to their business design needs. The content of the consensus mainly covers the standardization of account books, the certainty and execution results of transactions, the non dual uniqueness of transactions, the completeness of transaction sequence, and other information to ensure the safe and stable operation of the network.

5.4.1. Block generation logic【security】

Audit description： Block is the smallest unit of data storage in the blockchain. Each block is composed of "block header" and "block subject". The block subject is used to store transaction information, and the block header is composed of version number, parent block hash value, Merkle root, timestamp, difficulty value, random number, etc. whether the generation process of the block conforms to the consensus design directly determines the effectiveness of the block.

Risk hazard： Block forgery, chain forking, double flower attack and other

security issues.

Audit process：Check whether the block generation logic design is reasonable.

Audit results： After audit, there is no such safety problem.

```
fn block_production(c: &mut Criterion) {
	sp_tracing::try_init_simple();

	let runtime = tokio::runtime::Runtime::new().expect("creating tokio runtime doesn't fail; qed");
	let tokio_handle = runtime.handle().clone();

	let node = new_node(tokio_handle.clone());
	let client = &*node.client;

	// Buliding the very first block is around ~30x slower than any subsequent one,
	// so let's make sure it's built and imported before we benchmark anything.
	let mut block_builder = client.new_block(Default::default()).unwrap();
	block_builder.push(extrinsic_set_time(1)).unwrap();
	import_block(client, block_builder.build().unwrap());

	let (max_transfer_count, extrinsics) = prepare_benchmark(&client);
	log::info!("Maximum transfer count: {}", max_transfer_count);

	let mut group = c.benchmark_group("Block production");

	group.sample_size(10);
	group.throughput(Throughput::Elements(max_transfer_count as u64));

	let block_id = BlockId::Hash(client.chain_info().best_hash);

	group.bench_function(format!("{} transfers (no proof)", max_transfer_count), |b| {
		b.iter_batched(
			|| extrinsics.clone(),
			|extrinsics| {
```

```
                        let mut block_builder =
                            client.new_block_at(&block_id,                    Default::default(),
RecordProof::No).unwrap();
                        for extrinsic in extrinsics {
                            block_builder.push(extrinsic).unwrap();
                        }
                        block_builder.build().unwrap()
                    },
                    BatchSize::SmallInput,
                )
        });

        group.bench_function(format!("{} transfers (with proof)", max_transfer_count), |b| {
            b.iter_batched(
                || extrinsics.clone(),
                |extrinsics| {
                    let mut block_builder =
                        client.new_block_at(&block_id,                    Default::default(),
RecordProof::Yes).unwrap();
                    for extrinsic in extrinsics {
                        block_builder.push(extrinsic).unwrap();
                    }
                    block_builder.build().unwrap()
                },
                BatchSize::SmallInput,
            )
        });
}
fn new_node(tokio_handle: Handle) -> node_cli::service::NewFullBase {
        let base_path = BasePath::new_temp_dir().expect("Creates base path");
        let root = base_path.path().to_path_buf();

        let network_config = NetworkConfiguration::new(
            Sr25519Keyring::Alice.to_seed(),
            "network/test/0.1",
            Default::default(),
```

```
        None,
    );
    let spec = Box::new(node_cli::chain_spec::development_config());

    let config = Configuration {
        impl_name: "BenchmarkImpl".into(),
        impl_version: "1.0".into(),
        role: Role::Authority,
        tokio_handle,
        transaction_pool: TransactionPoolOptions {
            ready: PoolLimit { count: 100_000, total_bytes: 100 * 1024 * 1024 },
            future: PoolLimit { count: 100_000, total_bytes: 100 * 1024 * 1024 },
            reject_future_transactions: false,
            ban_time: Duration::from_secs(30 * 60),
        },
        network: network_config,
        keystore: KeystoreConfig::InMemory,
        keystore_remote: Default::default(),
        database: DatabaseSource::RocksDb { path: root.join("db"), cache_size: 128 },
        state_cache_size: 67108864,
        state_cache_child_ratio: None,
        state_pruning: Some(PruningMode::ArchiveAll),
        keep_blocks: KeepBlocks::All,
        chain_spec: spec,
        wasm_method: WasmExecutionMethod::Interpreted,
        // NOTE: we enforce the use of the native runtime to make the errors more debuggable
        execution_strategies: ExecutionStrategies {
            syncing: sc_client_api::ExecutionStrategy::NativeWhenPossible,
            importing: sc_client_api::ExecutionStrategy::NativeWhenPossible,
            block_construction: sc_client_api::ExecutionStrategy::NativeWhenPossible,
            offchain_worker: sc_client_api::ExecutionStrategy::NativeWhenPossible,
            other: sc_client_api::ExecutionStrategy::NativeWhenPossible,
        },
        rpc_http: None,
        rpc_ws: None,
        rpc_ipc: None,
```

```
            rpc_ws_max_connections: None,
            rpc_cors: None,
            rpc_methods: Default::default(),
            rpc_max_payload: None,
            rpc_max_request_size: None,
            rpc_max_response_size: None,
            rpc_id_provider: None,
            rpc_max_subs_per_conn: None,
            ws_max_out_buffer_capacity: None,
            prometheus_config: None,
            telemetry_endpoints: None,
            default_heap_pages: None,
            offchain_worker: OffchainWorkerConfig { enabled: true, indexing_enabled: false },
            force_authoring: false,
            disable_grandpa: false,
            dev_key_seed: Some(Sr25519Keyring::Alice.to_seed()),
            tracing_targets: None,
            tracing_receiver: Default::default(),
            max_runtime_instances: 8,
            runtime_cache_size: 2,
            announce_block: true,
            base_path: Some(base_path),
            informant_output_format: Default::default(),
            wasm_runtime_overrides: None,
        };
        node_cli::service::new_full_base(config, false, |_, _| ()).expect("Creates node")
}
        fn new_block(
            &self,
            inherent_digests: Digest,
        ) -> sp_blockchain::Result<sc_block_builder::BlockBuilder<Block, Self, B>> {
            let info = self.chain_info();
            sc_block_builder::BlockBuilder::new(
                self,
                info.best_hash,
                info.best_number,
```

```
                RecordProof::No,
                inherent_digests,
                &self.backend,
            )
        }
        /// Get blockchain info.
        pub fn chain_info(&self) -> blockchain::Info<Block> {
            self.backend.blockchain().info()
        }
fn import_block(
        mut client: &FullClient,
        built: BuiltBlock<
            node_primitives::Block,
            <FullClient as sp_api::CallApiAt<node_primitives::Block>>::StateBackend,
        >,
) {
        let mut params = BlockImportParams::new(BlockOrigin::File, built.block.header);
        params.state_action =
        StateAction::ApplyChanges(sc_consensus::StorageChanges::Changes(built.storage_changes))
;
        params.fork_choice = Some(ForkChoiceStrategy::LongestChain);
        futures::executor::block_on(client.import_block(params, Default::default()))
            .expect("importing a block doesn't fail");
}
```

Safety advice：NONE.

5.4.2. Block signature verification 【security】

Audit description： Check whether the block signature verification logic design has design defects.

Risk hazard： Block forgery, data tampering, chain forking and other security

issues.

Audit process： Check the business logic design related to block legitimacy verification.

Audit results：After audit, there is no such safety problem.

```
fn verify<Pair>(sig_data: Vec<u8>, message: Vec<u8>, uri: &str) -> error::Result<()>
where
	Pair: sp_core::Pair,
	Pair::Signature: for<'a> TryFrom<&'a [u8]>,
{
	let signature =
		Pair::Signature::try_from(&sig_data).map_err(|_|
error::Error::SignatureFormatInvalid)?;

	let pubkey = if let Ok(pubkey_vec) = hex::decode(uri) {
		Pair::Public::from_slice(pubkey_vec.as_slice())
			.map_err(|_| error::Error::KeyFormatInvalid)?
	} else {
		Pair::Public::from_string(uri)?
	};

	if Pair::verify(&signature, &message, &pubkey) {
		println!("Signature verifies correctly.");
	} else {
		return Err(error::Error::SignatureInvalid)
	}

	Ok(())
}
```

Safety advice：NONE.

5.4.3. Block verification check 【security】

Audit description： After the block is generated, it needs to go through the legitimacy check to determine the effectiveness of the block, and then add the block to the blockchain. The legitimacy check of the block directly affects the security of the blockchain system.

Risk hazard： Block forgery, chain forking, data tampering and other security issues.

Audit process： Check the business logic design related to block legitimacy verification。

Audit results： After audit, there is no such safety problem.

```
/// check a header has been signed by the right key. If the slot is too far in the future, an error
/// will be returned. If it's successful, returns the pre-header and the digest item
/// containing the seal.
///
/// This digest item will always return `Some` when used with `as_aura_seal`.
fn check_header<C, B: BlockT, P: Pair>(
	client: &C,
	slot_now: Slot,
	mut header: B::Header,
	hash: B::Hash,
	authorities: &[AuthorityId<P>],
	check_for_equivocation: CheckForEquivocation,
) -> Result<CheckedHeader<B::Header, (Slot, DigestItem)>, Error<B>>
where
	P::Signature: Codec,
```

```
    C: sc_client_api::backend::AuxStore,
    P::Public: Encode + Decode + PartialEq + Clone,
{
    let seal = header.digest_mut().pop().ok_or(Error::HeaderUnsealed(hash))?;

    let sig = seal.as_aura_seal().ok_or_else(|| aura_err(Error::HeaderBadSeal(hash)))?;

    let slot = find_pre_digest::<B, P::Signature>(&header)?;

    if slot > slot_now {
        header.digest_mut().push(seal);
        Ok(CheckedHeader::Deferred(header, slot))
    } else {
        // check the signature is valid under the expected authority and
        // chain state.
        let expected_author =
            slot_author::<P>(slot, authorities).ok_or(Error::SlotAuthorNotFound)?;

        let pre_hash = header.hash();

        if P::verify(&sig, pre_hash.as_ref(), expected_author) {
            if check_for_equivocation.check_for_equivocation() {
                if let Some(equivocation_proof) =
                    check_equivocation(client, slot_now, slot, &header, expected_author)
                        .map_err(Error::Client)?
                {
                    info!(
                        target: "aura",
                        "Slot author is equivocating at slot {} with headers {:?} and {:?}",
                        slot,
                        equivocation_proof.first_header.hash(),
                        equivocation_proof.second_header.hash(),
                    );
                }
            }
```

```
                    Ok(CheckedHeader::Checked(header, (slot, seal)))
                } else {
                    Err(Error::BadSignature(hash))
                }
            }
        }
        fn digest_mut(&mut self) -> &mut Digest {
            #[cfg(feature = "std")]
            log::debug!(target: "header", "Retrieving mutable reference to digest");
            &mut self.digest
        }
/// Checks if the header is an equivocation and returns the proof in that case.
///
/// Note: it detects equivocations only when slot_now - slot <= MAX_SLOT_CAPACITY.
pub fn check_equivocation<C, H, P>(
        backend: &C,
        slot_now: Slot,
        slot: Slot,
        header: &H,
        signer: &P,
) -> ClientResult<Option<EquivocationProof<H, P>>>
where
        H: Header,
        C: AuxStore,
        P: Clone + Encode + Decode + PartialEq,
{
        // We don't check equivocations for old headers out of our capacity.
        if slot_now.saturating_sub(*slot) > MAX_SLOT_CAPACITY {
            return Ok(None)
        }

        // Key for this slot.
        let mut curr_slot_key = SLOT_HEADER_MAP_KEY.to_vec();
        slot.using_encoded(|s| curr_slot_key.extend(s));

        // Get headers of this slot.
```

```
	let mut headers_with_sig =
		load_decode::<_, Vec<(H, P)>>(backend, &curr_slot_key[..])?.unwrap_or_else(Vec::new);

	// Get first slot saved.
	let slot_header_start = SLOT_HEADER_START.to_vec();
	let first_saved_slot = load_decode::<_, Slot>(backend, &slot_header_start[..])?.unwrap_or(slot);

	if slot_now < first_saved_slot {
		// The code below assumes that slots will be visited sequentially.
		return Ok(None)
	}

	for (prev_header, prev_signer) in headers_with_sig.iter() {
		// A proof of equivocation consists of two headers:
		// 1) signed by the same voter,
		if prev_signer == signer {
			// 2) with different hash
			if header.hash() != prev_header.hash() {
				return Ok(Some(EquivocationProof {
					slot,
					offender: signer.clone(),
					first_header: prev_header.clone(),
					second_header: header.clone(),
				}))
			} else {
				// We don't need to continue in case of duplicated header,
				// since it's already saved and a possible equivocation
				// would have been detected before.
				return Ok(None)
			}
		}
	}

	let mut keys_to_delete = vec![];
```

```
        let mut new_first_saved_slot = first_saved_slot;

        if *slot_now - *first_saved_slot >= PRUNING_BOUND {
            let prefix = SLOT_HEADER_MAP_KEY.to_vec();
            new_first_saved_slot = slot_now.saturating_sub(MAX_SLOT_CAPACITY);

            for s in u64::from(first_saved_slot)..new_first_saved_slot.into() {
                let mut p = prefix.clone();
                s.using_encoded(|s| p.extend(s));
                keys_to_delete.push(p);
            }
        }

        headers_with_sig.push((header.clone(), signer.clone()));

        backend.insert_aux(
            &[
                (&curr_slot_key[..], headers_with_sig.encode().as_slice()),
                (&slot_header_start[..], new_first_saved_slot.encode().as_slice()),
            ],
            &keys_to_delete.iter().map(|k| &k[..]).collect::<Vec<&[u8]>>()[..],
        )?;

        Ok(None)
    }
    /// Check a header has been signed by the right key. If the slot is too far in
    /// the future, an error will be returned. If successful, returns the pre-header
    /// and the digest item containing the seal.
    ///
    /// The seal must be the last digest.  Otherwise, the whole header is considered
    /// unsigned.  This is required for security and must not be changed.
    ///
    /// This digest item will always return `Some` when used with `as_babe_pre_digest`.
    ///
    /// The given header can either be from a primary or secondary slot assignment,
    /// with each having different validation logic.
```

```
pub(super) fn check_header<B: BlockT + Sized>(
	params: VerificationParams<B>,
) -> Result<CheckedHeader<B::Header, VerifiedHeaderInfo>, Error<B>> {
	let VerificationParams { mut header, pre_digest, slot_now, epoch } = params;

	let authorities = &epoch.authorities;
	let pre_digest = pre_digest.map(Ok).unwrap_or_else(|| find_pre_digest::<B>(&header))?;

	trace!(target: "babe", "Checking header");
	let seal = header
		.digest_mut()
		.pop()
		.ok_or_else(|| babe_err(Error::HeaderUnsealed(header.hash())))?;

	let sig = seal
		.as_babe_seal()
		.ok_or_else(|| babe_err(Error::HeaderBadSeal(header.hash())))?;

	// the pre-hash of the header doesn't include the seal
	// and that's what we sign
	let pre_hash = header.hash();

	if pre_digest.slot() > slot_now {
		header.digest_mut().push(seal);
		return Ok(CheckedHeader::Deferred(header, pre_digest.slot()))
	}

	let author = match authorities.get(pre_digest.authority_index() as usize) {
		Some(author) => author.0.clone(),
		None => return Err(babe_err(Error::SlotAuthorNotFound)),
	};

	match &pre_digest {
		PreDigest::Primary(primary) => {
			debug!(target: "babe",
				"Verifying primary block #{} at slot: {}",
```

```
                header.number(),
                primary.slot,
            );

            check_primary_header::<B>(pre_hash, primary, sig, epoch, epoch.config.c)?;
        },
        PreDigest::SecondaryPlain(secondary)
            if epoch.config.allowed_slots.is_secondary_plain_slots_allowed() =>
        {
            debug!(target: "babe",
                "Verifying secondary plain block #{} at slot: {}",
                header.number(),
                secondary.slot,
            );

            check_secondary_plain_header::<B>(pre_hash, secondary, sig, epoch)?;
        },
        PreDigest::SecondaryVRF(secondary)
            if epoch.config.allowed_slots.is_secondary_vrf_slots_allowed() =>
        {
            debug!(target: "babe",
                "Verifying secondary VRF block #{} at slot: {}",
                header.number(),
                secondary.slot,
            );

            check_secondary_vrf_header::<B>(pre_hash, secondary, sig, epoch)?;
        },
        _ => return Err(babe_err(Error::SecondarySlotAssignmentsDisabled)),
    }

    let info = VerifiedHeaderInfo {
        pre_digest: CompatibleDigestItem::babe_pre_digest(pre_digest),
        seal,
        author,
    };
```

```
        Ok(CheckedHeader::Checked(header, info))
}

pub(super) struct VerifiedHeaderInfo {
        pub(super) pre_digest: DigestItem,
        pub(super) seal: DigestItem,
        pub(super) author: AuthorityId,
}

/// Check a primary slot proposal header. We validate that the given header is
/// properly signed by the expected authority, and that the contained VRF proof
/// is valid. Additionally, the weight of this block must increase compared to
/// its parent since it is a primary block.
fn check_primary_header<B: BlockT + Sized>(
        pre_hash: B::Hash,
        pre_digest: &PrimaryPreDigest,
        signature: AuthoritySignature,
        epoch: &Epoch,
        c: (u64, u64),
) -> Result<(), Error<B>> {
        let author = &epoch.authorities[pre_digest.authority_index as usize].0;

        if AuthorityPair::verify(&signature, pre_hash, author) {
                let (inout, _) = {
                        let transcript = make_transcript(&epoch.randomness, pre_digest.slot,
epoch.epoch_index);

                        schnorrkel::PublicKey::from_bytes(author.as_slice())
                                .and_then(|p| {
                                        p.vrf_verify(transcript, &pre_digest.vrf_output, &pre_digest.vrf_proof)
                                })
                                .map_err(|s| babe_err(Error::VRFVerificationFailed(s)))?
                };

                let threshold =
                        calculate_primary_threshold(c, &epoch.authorities, pre_digest.authority_index as
```

```
usize);

            if !check_primary_threshold(&inout, threshold) {
                return Err(babe_err(Error::VRFVerificationOfBlockFailed(author.clone(),
threshold)))
            }

            Ok(())
        } else {
            Err(babe_err(Error::BadSignature(pre_hash)))
        }
}

/// Check a secondary slot proposal header. We validate that the given header is
/// properly signed by the expected authority, which we have a deterministic way
/// of computing. Additionally, the weight of this block must stay the same
/// compared to its parent since it is a secondary block.
fn check_secondary_plain_header<B: BlockT>(
        pre_hash: B::Hash,
        pre_digest: &SecondaryPlainPreDigest,
        signature: AuthoritySignature,
        epoch: &Epoch,
) -> Result<(), Error<B>> {
        // check the signature is valid under the expected authority and
        // chain state.
        let expected_author =
            secondary_slot_author(pre_digest.slot, &epoch.authorities, epoch.randomness)
                .ok_or(Error::NoSecondaryAuthorExpected)?;

        let author = &epoch.authorities[pre_digest.authority_index as usize].0;

        if expected_author != author {
            return Err(Error::InvalidAuthor(expected_author.clone(), author.clone()))
        }

        if AuthorityPair::verify(&signature, pre_hash.as_ref(), author) {
```

```
            Ok(())
        } else {
            Err(Error::BadSignature(pre_hash))
        }
}

/// Check a secondary VRF slot proposal header.
fn check_secondary_vrf_header<B: BlockT>(
        pre_hash: B::Hash,
        pre_digest: &SecondaryVRFPreDigest,
        signature: AuthoritySignature,
        epoch: &Epoch,
) -> Result<(), Error<B>> {
        // check the signature is valid under the expected authority and
        // chain state.
        let expected_author =
            secondary_slot_author(pre_digest.slot, &epoch.authorities, epoch.randomness)
                .ok_or(Error::NoSecondaryAuthorExpected)?;

        let author = &epoch.authorities[pre_digest.authority_index as usize].0;

        if expected_author != author {
            return Err(Error::InvalidAuthor(expected_author.clone(), author.clone()))
        }

        if AuthorityPair::verify(&signature, pre_hash.as_ref(), author) {
            let transcript = make_transcript(&epoch.randomness, pre_digest.slot,
epoch.epoch_index);
            schnorrkel::PublicKey::from_bytes(author.as_slice())
                .and_then(|p| p.vrf_verify(transcript, &pre_digest.vrf_output,
&pre_digest.vrf_proof))
                .map_err(|s| babe_err(Error::VRFVerificationFailed(s)))?;
            Ok(())
        } else {
            Err(Error::BadSignature(pre_hash))
        }
```

```
}
```

Safety advice：NONE.

5.4.4. Transaction pooling logic【security】

Audit description： Check whether the logic design of transaction pooling is reasonable.

Risk hazard： It depends on the specific situation

Audit process： Audit the logic design of transaction pooling, and check whether there is a limit on the maximum number of transactions and the process of adding transactions.

Audit results： After audit, there is no such safety problem.

```
pub struct PoolStatus {
    /// Number of transactions in the ready queue.
    pub ready: usize,
    /// Sum of bytes of ready transaction encodings.
    pub ready_bytes: usize,
    /// Number of transactions in the future queue.
    pub future: usize,
    /// Sum of bytes of ready transaction encodings.
    pub future_bytes: usize,
}
/// Transaction pool interface.
pub trait TransactionPool: Send + Sync {
    /// Block type.
    type Block: BlockT;
    /// Transaction hash type.
    type Hash: Hash + Eq + Member + Serialize + DeserializeOwned;
```

```
/// In-pool transaction type.
type InPoolTransaction: InPoolTransaction<
	Transaction = TransactionFor<Self>,
	Hash = TxHash<Self>,
>;
/// Error type.
type Error: From<crate::error::Error> + crate::error::IntoPoolError;

// *** RPC

/// Returns a future that imports a bunch of unverified transactions to the pool.
fn submit_at(
	&self,
	at: &BlockId<Self::Block>,
	source: TransactionSource,
	xts: Vec<TransactionFor<Self>>,
) -> PoolFuture<Vec<Result<TxHash<Self>, Self::Error>>, Self::Error>;

/// Returns a future that imports one unverified transaction to the pool.
fn submit_one(
	&self,
	at: &BlockId<Self::Block>,
	source: TransactionSource,
	xt: TransactionFor<Self>,
) -> PoolFuture<TxHash<Self>, Self::Error>;

/// Returns a future that import a single transaction and starts to watch their progress in the
/// pool.
fn submit_and_watch(
	&self,
	at: &BlockId<Self::Block>,
	source: TransactionSource,
	xt: TransactionFor<Self>,
) -> PoolFuture<Pin<Box<TransactionStatusStreamFor<Self>>>, Self::Error>;

// *** Block production / Networking
```

```
	/// Get an iterator for ready transactions ordered by priority.
	///
	/// Guarantees to return only when transaction pool got updated at `at` block.
	/// Guarantees to return immediately when `None` is passed.
	fn ready_at(
		&self,
		at: NumberFor<Self::Block>,
	) -> Pin<
		Box<
			dyn Future<
				Output = Box<dyn ReadyTransactions<Item = Arc<Self::InPoolTransaction>> + Send>,
			> + Send,
		>,
	>;

	/// Get an iterator for ready transactions ordered by priority.
	fn ready(&self) -> Box<dyn ReadyTransactions<Item = Arc<Self::InPoolTransaction>> + Send>;

	// *** Block production
	/// Remove transactions identified by given hashes (and dependent transactions) from the pool.
	fn remove_invalid(&self, hashes: &[TxHash<Self>]) -> Vec<Arc<Self::InPoolTransaction>>;

	// *** logging
	/// Returns pool status.
	fn status(&self) -> PoolStatus;

	// *** logging / RPC / networking
	/// Return an event stream of transactions imported to the pool.
	fn import_notification_stream(&self) -> ImportNotificationStream<TxHash<Self>>;

	// *** networking
	/// Notify the pool about transactions broadcast.
	fn on_broadcasted(&self, propagations: HashMap<TxHash<Self>, Vec<String>>);
```

```
        /// Returns transaction hash
        fn hash_of(&self, xt: &TransactionFor<Self>) -> TxHash<Self>;

        /// Return specific ready transaction by hash, if there is one.
        fn ready_transaction(&self, hash: &TxHash<Self>) -> Option<Arc<Self::InPoolTransaction>>;
}
impl<TPool: LocalTransactionPool> OffchainSubmitTransaction<TPool::Block> for TPool {
        fn submit_at(
                &self,
                at: &BlockId<TPool::Block>,
                extrinsic: <TPool::Block as BlockT>::Extrinsic,
        ) -> Result<(), ()> {
                log::debug!(
                        target: "txpool",
                        "(offchain call) Submitting a transaction to the pool: {:?}",
                        extrinsic
                );

                let result = self.submit_local(at, extrinsic);

                result.map(|_| ()).map_err(|e| {
                        log::warn!(
                                target: "txpool",
                                "(offchain call) Error submitting a transaction to the pool: {}",
                                e
                        )
                })
        }
}
```

Safety advice：NONE.

5.4.5. Transaction signature logic【security】

Audit description： Check whether the signature content items and verification logic of various types of transactions are sufficient.

Risk hazard： Replay of same chain or cross chain transactions, forge legal transactions, and affect asset safety

Audit process： Check whether the signature content items and verification logic of various types of transactions are sufficient.

Audit results： After audit, there is no such safety problem.

```
/// Sign transaction with keypair from this keyring.
pub fn sign(
    &self,
    xt: CheckedExtrinsic,
    spec_version: u32,
    tx_version: u32,
    genesis_hash: [u8; 32],
) -> UncheckedExtrinsic {
    match xt.signed {
        Some((signed, extra)) => {
            let payload = (
                xt.function,
                extra.clone(),
                spec_version,
                tx_version,
                genesis_hash,
                genesis_hash,
            );
            let key = self.accounts.get(&signed).expect("Account id not found in
```

```
keyring");
                        let signature = payload.using_encoded(|b| {
                            if b.len() > 256 {
                                key.sign(&sp_io::hashing::blake2_256(b))
                            } else {
                                key.sign(b)
                            }
                        });
                        UncheckedExtrinsic {
                            signature: Some((sp_runtime::MultiAddress::Id(signed), signature,
extra)),
                            function: payload.0,
                        }
                    },
                    None => UncheckedExtrinsic { signature: None, function: xt.function },
                }
            }
fn sign<P: sp_core::Pair>(
    suri: &str,
    password: Option<SecretString>,
    message: Vec<u8>,
) -> error::Result<String> {
    let pair = utils::pair_from_suri::<P>(suri, password)?;
    Ok(hex::encode(pair.sign(&message)))
}
```

Safety advice：NONE.

5.4.6. Transaction verification logic 【security】

Audit description： Audit the verification logic of packaged transactions and check whether there are logical design defects.

Risk hazard： Forging legal transactions, affecting asset security

Audit process： Audit the verification logic of packaged transactions and check whether there are logical design defects.

Audit results： After audit, the transaction verification logic design is reasonable, and there is no such security problem.

```
/// Check a given signed transaction for validity. This doesn't execute any
/// side-effects; it merely checks whether the transaction would panic if it were included or
/// not.
///
/// Changes made to storage should be discarded.
pub fn validate_transaction(
	source: TransactionSource,
	uxt: Block::Extrinsic,
	block_hash: Block::Hash,
) -> TransactionValidity {
	sp_io::init_tracing();
	use sp_tracing::{enter_span, within_span};

	<frame_system::Pallet<System>>::initialize(
		&(frame_system::Pallet::<System>::block_number() + One::one()),
		&block_hash,
		&Default::default(),
	);

	enter_span! { sp_tracing::Level::TRACE, "validate_transaction" };

	let encoded_len = within_span! { sp_tracing::Level::TRACE, "using_encoded";
		uxt.using_encoded(|d| d.len())
	};

	let xt = within_span! { sp_tracing::Level::TRACE, "check";
		uxt.check(&Default::default())
	}?;
```

```
        let dispatch_info = within_span! { sp_tracing::Level::TRACE, "dispatch_info";
            xt.get_dispatch_info()
        };

        within_span! {
            sp_tracing::Level::TRACE, "validate";
            xt.validate::<UnsignedValidator>(source, &dispatch_info, encoded_len)
        }
    }
```

Safety advice：NONE.

5.4.7. Consensus mechanism design 【security】

Audit description： Check whether there are design defects in the design of the public chain consensus mechanism. For the new type of consensus mechanism, check whether there are security risks in the design according to its documents, such as missing transactions / unverified transactions / forged proof of computing power / centralization, etc.

Risk hazard： Chain forking, block forgery, transaction tampering and other security issues.

Audit process: Check the public chain consensus design for potential safety hazards according to the official documents of the public chain project.

Audit results： According to the audit, the deer network public chain uses proof of work (POW) and proof of designated interests (NPOs) protocols, and is developed based on the substrate blockchain development framework, so there is no such security problem.

```
/// Algorithm used for proof of work.
pub trait PowAlgorithm<B: BlockT> {
	/// Difficulty for the algorithm.
	type Difficulty: TotalDifficulty + Default + Encode + Decode + Ord + Clone + Copy;

	/// Get the next block's difficulty.
	///
	/// This function will be called twice during the import process, so the implementation
	/// should be properly cached.
	fn difficulty(&self, parent: B::Hash) -> Result<Self::Difficulty, Error<B>>;
	/// Verify that the seal is valid against given pre hash when parent block is not yet imported.
	///
	/// None means that preliminary verify is not available for this algorithm.
	fn preliminary_verify(
		&self,
		_pre_hash: &B::Hash,
		_seal: &Seal,
	) -> Result<Option<bool>, Error<B>> {
		Ok(None)
	}
	/// Break a fork choice tie.
	///
	/// By default this chooses the earliest block seen. Using uniform tie
	/// breaking algorithms will help to protect against selfish mining.
	///
	/// Returns if the new seal should be considered best block.
	fn break_tie(&self, _own_seal: &Seal, _new_seal: &Seal) -> bool {
		false
```

```
	}
	/// Verify that the difficulty is valid against given seal.
	fn verify(
		&self,
		parent: &BlockId<B>,
		pre_hash: &B::Hash,
		pre_digest: Option<&[u8]>,
		seal: &Seal,
		difficulty: Self::Difficulty,
	) -> Result<bool, Error<B>>;
}

/// A block importer for PoW.
pub struct PowBlockImport<B: BlockT, I, C, S, Algorithm, CAW, CIDP> {
	algorithm: Algorithm,
	inner: I,
	select_chain: S,
	client: Arc<C>,
	create_inherent_data_providers: Arc<CIDP>,
	check_inherents_after: <<B as BlockT>::Header as HeaderT>::Number,
	can_author_with: CAW,
}

impl<B: BlockT, I: Clone, C, S: Clone, Algorithm: Clone, CAW: Clone, CIDP> Clone
	for PowBlockImport<B, I, C, S, Algorithm, CAW, CIDP>
{
	fn clone(&self) -> Self {
		Self {
			algorithm: self.algorithm.clone(),
			inner: self.inner.clone(),
			select_chain: self.select_chain.clone(),
			client: self.client.clone(),
			create_inherent_data_providers: self.create_inherent_data_providers.clone(),
			check_inherents_after: self.check_inherents_after,
			can_author_with: self.can_author_with.clone(),
		}
```

```
    }
}

impl<B, I, C, S, Algorithm, CAW, CIDP> PowBlockImport<B, I, C, S, Algorithm, CAW, CIDP>
where
    B: BlockT,
    I: BlockImport<B, Transaction = sp_api::TransactionFor<C, B>> + Send + Sync,
    I::Error: Into<ConsensusError>,
    C: ProvideRuntimeApi<B> + Send + Sync + HeaderBackend<B> + AuxStore + BlockOf,
    C::Api: BlockBuilderApi<B>,
    Algorithm: PowAlgorithm<B>,
    CAW: CanAuthorWith<B>,
    CIDP: CreateInherentDataProviders<B, ()>,
{
    /// Create a new block import suitable to be used in PoW
    pub fn new(
        inner: I,
        client: Arc<C>,
        algorithm: Algorithm,
        check_inherents_after: <<B as BlockT>::Header as HeaderT>::Number,
        select_chain: S,
        create_inherent_data_providers: CIDP,
        can_author_with: CAW,
    ) -> Self {
        Self {
            inner,
            client,
            algorithm,
            check_inherents_after,
            select_chain,
            create_inherent_data_providers: Arc::new(create_inherent_data_providers),
            can_author_with,
        }
    }

    async fn check_inherents(
```

```
        &self,
        block: B,
        block_id: BlockId<B>,
        inherent_data_providers: CIDP::InherentDataProviders,
        execution_context: ExecutionContext,
    ) -> Result<(), Error<B>> {
        if *block.header().number() < self.check_inherents_after {
            return Ok(())
        }

        if let Err(e) = self.can_author_with.can_author_with(&block_id) {
            debug!(
                target: "pow",
                "Skipping `check_inherents` as authoring version is not compatible: {}",
                e,
            );

            return Ok(())
        }

        let inherent_data = inherent_data_providers
            .create_inherent_data()
            .map_err(|e| Error::CreateInherents(e))?;

        let inherent_res = self
            .client
            .runtime_api()
            .check_inherents_with_context(&block_id, execution_context, block,
inherent_data)
            .map_err(|e| Error::Client(e.into()))?;

        if !inherent_res.ok() {
            for (identifier, error) in inherent_res.into_errors() {
                match inherent_data_providers.try_handle_error(&identifier, &error).await {
                    Some(res) => res.map_err(Error::CheckInherents)?,
                    None => return Err(Error::CheckInherentsUnknownError(identifier)),
```

```
                        }
                    }
                }

                Ok(())
        }
}

#[async_trait::async_trait]
impl<B, I, C, S, Algorithm, CAW, CIDP> BlockImport<B>
        for PowBlockImport<B, I, C, S, Algorithm, CAW, CIDP>
where
        B: BlockT,
        I: BlockImport<B, Transaction = sp_api::TransactionFor<C, B>> + Send + Sync,
        I::Error: Into<ConsensusError>,
        S: SelectChain<B>,
        C: ProvideRuntimeApi<B> + Send + Sync + HeaderBackend<B> + AuxStore + BlockOf,
        C::Api: BlockBuilderApi<B>,
        Algorithm: PowAlgorithm<B> + Send + Sync,
        Algorithm::Difficulty: 'static + Send,
        CAW: CanAuthorWith<B> + Send + Sync,
        CIDP: CreateInherentDataProviders<B, ()> + Send + Sync,
{
        type Error = ConsensusError;
        type Transaction = sp_api::TransactionFor<C, B>;

        async fn check_block(
                &mut self,
                block: BlockCheckParams<B>,
        ) -> Result<ImportResult, Self::Error> {
                self.inner.check_block(block).await.map_err(Into::into)
        }

        async fn import_block(
                &mut self,
                mut block: BlockImportParams<B, Self::Transaction>,
```

```
        new_cache: HashMap<CacheKeyId, Vec<u8>>,
    ) -> Result<ImportResult, Self::Error> {
        let best_header = self
            .select_chain
            .best_chain()
            .await
            .map_err(|e| format!("Fetch best chain failed via select chain: {}", e))?;
        let best_hash = best_header.hash();

        let parent_hash = *block.header.parent_hash();
        let best_aux = PowAux::read::<_, B>(self.client.as_ref(), &best_hash)?;
        let mut aux = PowAux::read::<_, B>(self.client.as_ref(), &parent_hash)?;

        if let Some(inner_body) = block.body.take() {
            let check_block = B::new(block.header.clone(), inner_body);

            self.check_inherents(
                check_block.clone(),
                BlockId::Hash(parent_hash),
                self.create_inherent_data_providers
                    .create_inherent_data_providers(parent_hash, ())
                    .await?,
                block.origin.into(),
            )
            .await?;

            block.body = Some(check_block.deconstruct().1);
        }

        let inner_seal = fetch_seal::<B>(block.post_digests.last(), block.header.hash())?;

        let intermediate =

    block.take_intermediate::<PowIntermediate<Algorithm::Difficulty>>(INTERMEDIATE_KEY)?;

        let difficulty = match intermediate.difficulty {
```

```
        Some(difficulty) => difficulty,
        None => self.algorithm.difficulty(parent_hash)?,
    };

    let pre_hash = block.header.hash();
    let pre_digest = find_pre_digest::<B>(&block.header)?;
    if !self.algorithm.verify(
        &BlockId::hash(parent_hash),
        &pre_hash,
        pre_digest.as_ref().map(|v| &v[..]),
        &inner_seal,
        difficulty,
    )? {
        return Err(Error::<B>::InvalidSeal.into())
    }

    aux.difficulty = difficulty;
    aux.total_difficulty.increment(difficulty);

    let key = aux_key(&block.post_hash());
    block.auxiliary.push((key, Some(aux.encode())));
    if block.fork_choice.is_none() {
        block.fork_choice = Some(ForkChoiceStrategy::Custom(
            match aux.total_difficulty.cmp(&best_aux.total_difficulty) {
                Ordering::Less => false,
                Ordering::Greater => true,
                Ordering::Equal => {
                    let best_inner_seal =
                        fetch_seal::<B>(best_header.digest().logs.last(), best_hash)?;

                    self.algorithm.break_tie(&best_inner_seal, &inner_seal)
                },
            },
        ));
    }
```

```
            self.inner.import_block(block, new_cache).await.map_err(Into::into)
        }
}
```

Safety advice：NONE.

5.4.8. Consensus verification implementation【security】

Audit description： Check whether the consensus such as pow/poc can be used to construct legal blocks at less than the expected cost..

Risk hazard： Counterfeiting and centralization of computing power can lead to 51% attacks, chain forking and other problems.

Audit process： Audit the consensus verification implementation logic of the public chain.

Audit results： After audit, there is no such safety problem.

```
/// A verifier for PoW blocks.
pub struct PowVerifier<B: BlockT, Algorithm> {
        algorithm: Algorithm,
        _marker: PhantomData<B>,
}

impl<B: BlockT, Algorithm> PowVerifier<B, Algorithm> {
        pub fn new(algorithm: Algorithm) -> Self {
            Self { algorithm, _marker: PhantomData }
        }

        fn check_header(&self, mut header: B::Header) -> Result<(B::Header, DigestItem), Error<B>>
        where
            Algorithm: PowAlgorithm<B>,
```

```
    {
        let hash = header.hash();

        let (seal, inner_seal) = match header.digest_mut().pop() {
            Some(DigestItem::Seal(id, seal)) =>
                if id == POW_ENGINE_ID {
                    (DigestItem::Seal(id, seal.clone()), seal)
                } else {
                    return Err(Error::WrongEngine(id))
                },
            _ => return Err(Error::HeaderUnsealed(hash)),
        };

        let pre_hash = header.hash();

        if !self.algorithm.preliminary_verify(&pre_hash, &inner_seal)?.unwrap_or(true) {
            return Err(Error::FailedPreliminaryVerify)
        }

        Ok((header, seal))
    }
}

#[async_trait::async_trait]
impl<B: BlockT, Algorithm> Verifier<B> for PowVerifier<B, Algorithm>
where
    Algorithm: PowAlgorithm<B> + Send + Sync,
    Algorithm::Difficulty: 'static + Send,
{
    async fn verify(
        &mut self,
        mut block: BlockImportParams<B, ()>,
    ) -> Result<(BlockImportParams<B, ()>, Option<Vec<(CacheKeyId, Vec<u8>)>>), String> {
        let hash = block.header.hash();
        let (checked_header, seal) = self.check_header(block.header)?;
```

```
            let intermediate = PowIntermediate::<Algorithm::Difficulty> { difficulty: None };
            block.header = checked_header;
            block.post_digests.push(seal);
            block
                .intermediates
                .insert(Cow::from(INTERMEDIATE_KEY), Box::new(intermediate) as Box<_>);
            block.post_hash = Some(hash);

            Ok((block, None))
        }
}
```

Safety advice：NONE.

5.5. Incentive layer safety

The incentive layer aims to provide certain incentive measures to encourage nodes to participate in the security verification of blockchain and ensure the balance and benign development of the ecosystem. In the decentralized public chain, we must encourage the participating accounting nodes that abide by the rules and punish the nodes that do not abide by the rules, so that the system can develop in the direction of a virtuous circle. The blockchain incentive layer introduces economic factors into the blockchain technology system, which improves the efficiency of organizational collaboration and the efficiency of value exchange within the ecosystem. It is an important mechanism to ensure the benign cycle development of blockchain.

5.5.1. Token issuance mechanism 【security】

Audit description： Check whether the token distribution mechanism in the public chain system is consistent with the description in the project white paper, and conduct a security audit on the design of the token distribution mechanism.

Risk hazard： It depends on the specific situation.

Audit process： Conduct security audit on the token issuance mechanism of the public chain.

Audit results： After audit, there is no such safety problem.

Safety advice：NONE.

5.5.2. Token allocation mechanism【security】

Audit description： The public chain system realizes self drive through economic incentive distribution. Common distribution mechanisms include bitcoin transfer transaction incentive, Ethereum fuel incentive and other token incentives. Check whether the token distribution mechanism of the current public chain system is consistent with that described in the white paper, and whether the design of the distribution mechanism is reasonable.

Risk hazard： It depends on the specific situation.

Audit process： Audit the mining difficulty adjustment logic of the public chain.

Audit results： After audit, there is no such safety problem.

Safety advice：NONE.

5.6. Application Layer Security

The application layer carries the upper applications of blockchain. At present, it is widely used in financial, entertainment, notarization and other scenarios. It is the part closest to users, and it is also the part most prone to security problems.

Different public chains will show the characteristics of diversified application layer design according to the specific business they implement, which also makes the security problems of application layer attract much attention.

5.6.1. Account system authentication【security】

Audit description： Check whether the blockchain system has authentication logic for user identity and whether there is a problem.

Risk hazard： Forging identity bypasses authority inspection, and serious cases may threaten asset security.

Audit process： Check whether the blockchain system has authentication logic for user identity and whether there is a problem.

Audit results： After audit, there is no such safety problem.

Safety advice： NONE.

5.6.2. Account crud logic【security】

Audit description： Check whether the crud logic of the account system has illegal boundary conditions, and the time when the operation takes effect, and test whether there are loopholes and the impact on nodes and data.

Risk hazard： Conduct malicious operations on the account without

authorization.

Audit process： Check whether the crud logic of the account system has illegal boundary conditions, and the time when the operation takes effect, and test whether there are loopholes and the impact on nodes and data.

Audit results： After audit, there is no such safety problem.

```
/// Create the extrinsics that will initialize the accounts from the sudo account (Alice).
///
/// `start_nonce` is the current nonce of Alice.
fn create_account_extrinsics(
	client: &FullClient,
	accounts: &[sr25519::Pair],
) -> Vec<OpaqueExtrinsic> {
	let start_nonce = fetch_nonce(client, Sr25519Keyring::Alice.pair());

	accounts
		.iter()
		.enumerate()
		.flat_map(|(i, a)| {
			vec![
				// Reset the nonce by removing any funds
				create_extrinsic(
					client,
					Sr25519Keyring::Alice.pair(),
					SudoCall::sudo {
						call: Box::new(
							BalancesCall::set_balance {
								who: AccountId::from(a.public()).into(),
								new_free: 0,
								new_reserved: 0,
							}
							.into(),
```

```
                                ),
                            },
                            Some(start_nonce + (i as u32) * 2),
                        ),
                        // Give back funds
                        create_extrinsic(
                            client,
                            Sr25519Keyring::Alice.pair(),
                            SudoCall::sudo {
                                call: Box::new(
                                    BalancesCall::set_balance {
                                        who: AccountId::from(a.public()).into(),
                                        new_free: 1_000_000 * DOLLARS,
                                        new_reserved: 0,
                                    }
                                    .into(),
                                ),
                            },
                            Some(start_nonce + (i as u32) * 2 + 1),
                        ),
                    ]
                })
                .map(OpaqueExtrinsic::from)
                .collect()
}
fn create_accounts(num: usize) -> Vec<sr25519::Pair> {
    (0..num)
        .map(|i| {
            Pair::from_string(&format!("{}/{}", Sr25519Keyring::Alice.to_seed(), i), None)
                .expect("Creates account pair")
        })
        .collect()
}
```

Safety advice：NONE.

5.6.3. Traditional web security 【security】

Audit description： Check / test traditional web security items such as xxE injection, CSRF, SSRF, path traversal, and clear text transmission of sensitive information.

Risk hazard： Remote command execution vulnerability, CSRF, SSRF, xxE, XSS, arbitrary file reading, sensitive information disclosure, etc.

Audit process： After detection, there is no common Web attack risk such as xxE injection in the public chain code.

Audit results： After audit, there is no such safety problem.

Safety advice：NONE.

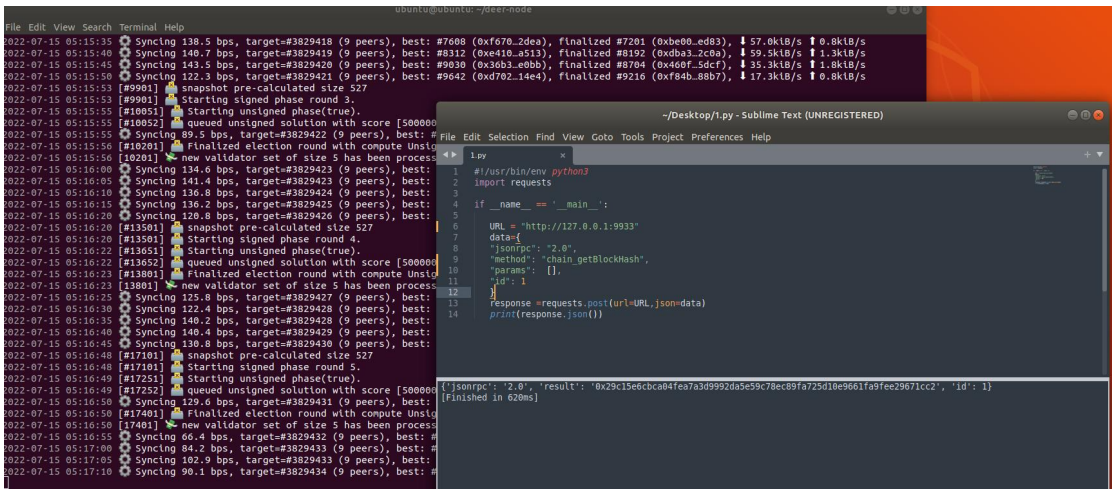
5.6.4. Availability of RPC interface 【security】

Audit description：Check whether the RPC interface can be called normally。

Risk hazard： Sensitive information disclosure, RPC interface call failure.

Audit process：Run the public link node and request data by calling the RPC interface to check whether the RPC interface can be called normally.

Audit results： After audit, the public chain RPC interface can be called normally.



Safety advice：NONE.

5.6.5. RPC interface call permission 【security】

Audit description：Check whether the RPC interface has permission control.

Risk hazard： Malicious call of RPC interface and disclosure of sensitive information.

Audit process： Conduct a security audit of the public chain source code, check whether there is permission control over RPC calls, and whether the RPC interface can call without authorization and obtain sensitive information.

Audit results：According to the audit, the public chain RPC interface will not be external by default. At the same time, the security defense can be carried out through parameter configuration when the node is started.

Safety advice：NONE.

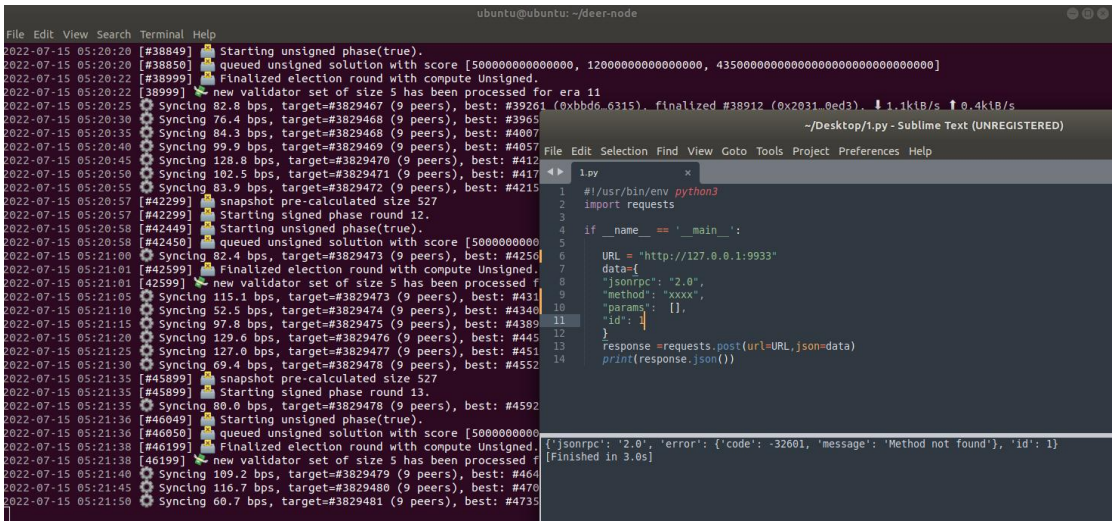
5.6.6. Robustness of RPC interface 【security】

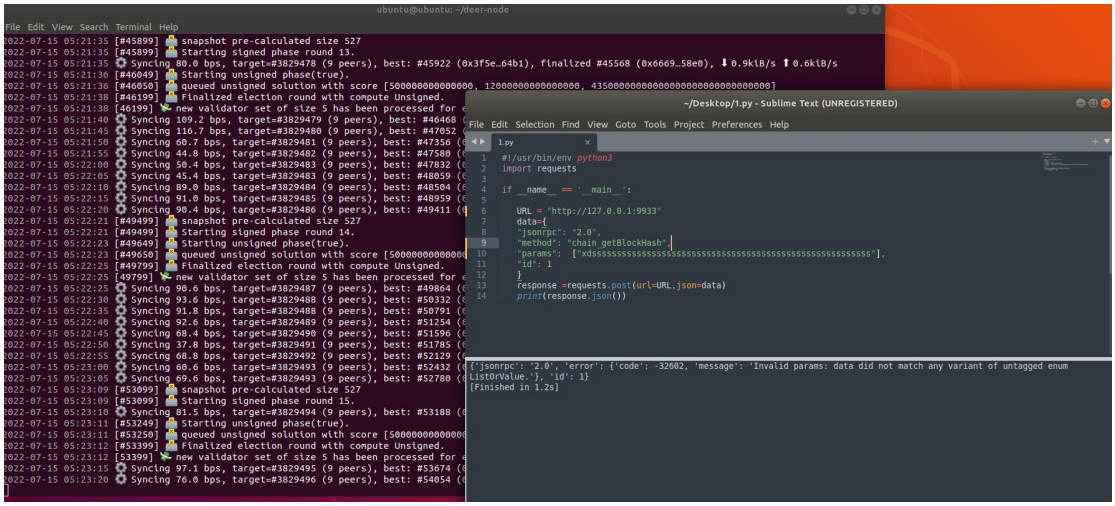
Audit description： Check whether the RPC interface handles the illegal parameters that may be passed in.

Risk hazard： Sensitive information is leaked and nodes are crashed.

Audit process： Conduct a security audit of the public chain source code to check whether the parameters passed in by the user during the RPC call are legitimate

Audit results： After audit, the RPC interface has checked the legitimacy of the parameters passed in by the user, and has good robustness.





Safety advice：NONE.

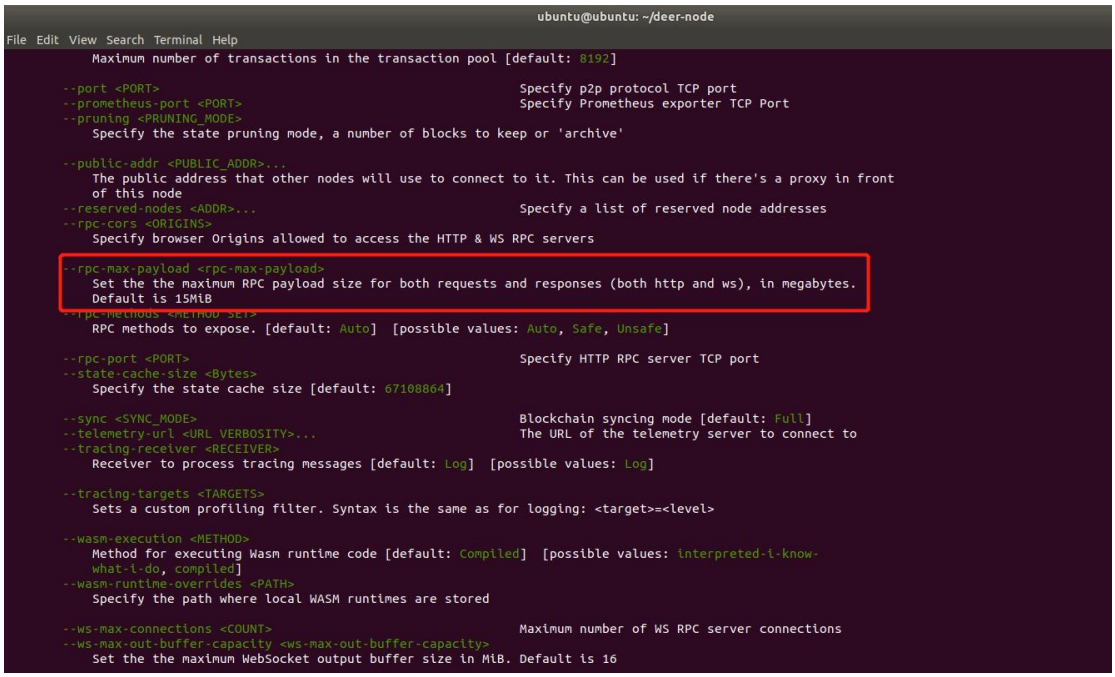
5.6.7. RPC interface denial of service 【security】

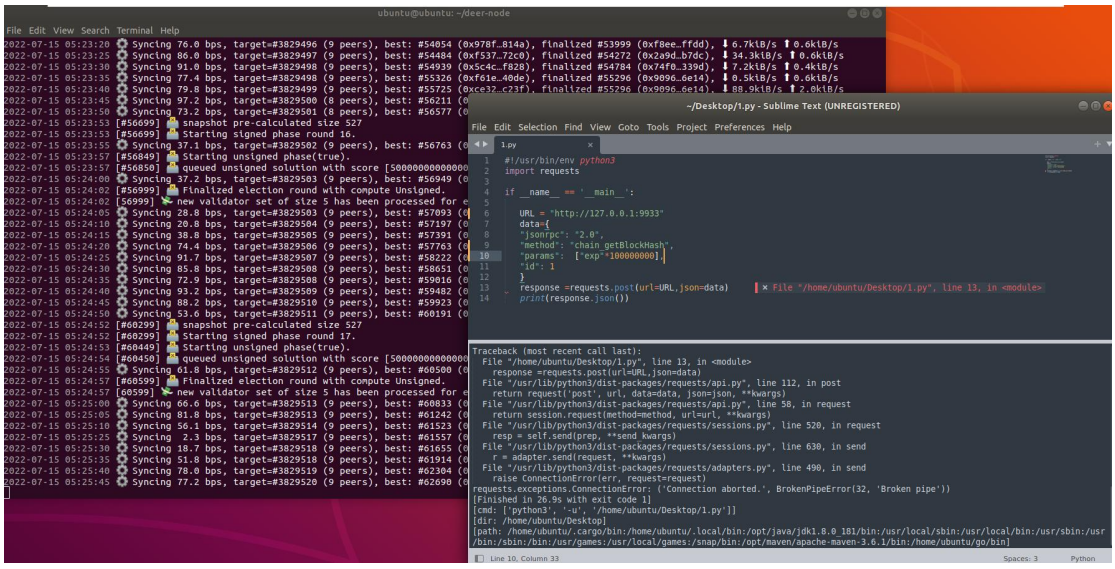
Audit description： Check whether the RPC interface has a limit on the maximum request packets.

Risk hazard： Node collapse.

Audit process： Conduct security audit on the public chain source code and check whether the RPC interface limits the size of the request packet constructed by the user.

Audit results： According to the audit, the RPC interface has the maximum limit processing on the request packets constructed by the user, which can well avoid DoS attacks.





Safety advice：NONE.

5.6.8. Resource is associated with chainid 【security】

Audit description： Check the business logic design related to the

association between resources and chainid.

Risk hazard： Logic design error

Audit process： Audit the business logic design associated with the resource and Chain ID in the public chain source code.

Audit results： After audit, the business logic design of the resource and Chain ID Association in the public chain source code is correct.

```
/// Stores a method name on chain under an associated resource ID.
///
/// # <weight>
/// - O(1) write
/// # </weight>
#[pallet::weight(195_000_000)]
pub fn set_resource(
    origin: OriginFor<T>,
    id: ResourceId,
    method: Vec<u8>,
) -> DispatchResult {
    T::BridgeCommitteeOrigin::ensure_origin(origin)?;
    Self::register_resource(id, method)
}
/// Removes a resource ID from the resource mapping.
///
/// After this call, bridge transfers with the associated resource ID will
/// be rejected.
///
/// # <weight>
/// - O(1) removal
/// # </weight>
#[pallet::weight(195_000_000)]
pub fn remove_resource(origin: OriginFor<T>, id: ResourceId) -> DispatchResult {
    T::BridgeCommitteeOrigin::ensure_origin(origin)?;
    Self::unregister_resource(id)
```

```
}
```

Safety advice：NONE.

5.6.9. Relay related logic design 【security】

Audit description： Check the relay service logic design related to cross chain.

Risk hazard： Logic design error.

Audit process： Audit the relay related business design for cross chain in the public chain source code.

Audit results： After audit, the relay related business logic design in the public chain source code is correct.

```
/// Enables a chain ID as a source or destination for a bridge transfer.
///
/// # <weight>
/// - O(1) lookup and insert
/// # </weight>
#[pallet::weight(195_000_000)]
pub fn whitelist_chain(origin: OriginFor<T>, id: BridgeChainId) -> DispatchResult {
    T::BridgeCommitteeOrigin::ensure_origin(origin)?;
    Self::whitelist(id)
}

/// Adds a new relayer to the relayer set.
///
/// # <weight>
/// - O(1) lookup and insert
/// # </weight>
#[pallet::weight(195_000_000)]
pub fn add_relayer(origin: OriginFor<T>, v: T::AccountId) -> DispatchResult {
```

```
            T::BridgeCommitteeOrigin::ensure_origin(origin)?;
            Self::register_relayer(v)
        }

        /// Removes an existing relayer from the set.
        ///
        /// # <weight>
        /// - O(1) lookup and removal
        /// # </weight>
        #[pallet::weight(195_000_000)]
        pub fn remove_relayer(origin: OriginFor<T>, v: T::AccountId) -> DispatchResult {
            T::BridgeCommitteeOrigin::ensure_origin(origin)?;
            Self::unregister_relayer(v)
        }
```

Safety advice：NONE.

5.6.10. Proposal related logic design 【security】

Audit description： Check whether the business logic design related to the proposal is reasonable.

Risk hazard： Logic design error.

Audit process： Audit the business logic design related to the proposal in the public chain source code.

Audit results： After audit, the business logic design related to the proposal in the public chain source code is correct.

```
        /// Commits a vote in favour of the provided proposal.
        ///
        /// If a proposal with the given nonce and source chain ID does not already exist, it
will
        /// be created with an initial vote in favour from the caller.
```

```
        ///
        /// # <weight>
        /// - weight of proposed call, regardless of whether execution is performed
        /// # </weight>
        #[pallet::weight({
            let dispatch_info = call.get_dispatch_info();
            (dispatch_info.weight + 195_000_000, dispatch_info.class, Pays::Yes)
        })]
        pub fn acknowledge_proposal(
            origin: OriginFor<T>,
            nonce: DepositNonce,
            src_id: BridgeChainId,
            r_id: ResourceId,
            call: Box<<T as Config>::Proposal>,
        ) -> DispatchResult {
            let who = ensure_signed(origin)?;
            ensure!(Self::is_relayer(&who), Error::<T>::MustBeRelayer);
            ensure!(Self::chain_whitelisted(src_id), Error::<T>::ChainNotWhitelisted);
            ensure!(Self::resource_exists(r_id), Error::<T>::ResourceDoesNotExist);

            Self::vote_for(who, nonce, src_id, call)
        }

        /// Commits a vote against a provided proposal.
        ///
        /// # <weight>
        /// - Fixed, since execution of proposal should not be included
        /// # </weight>
        #[pallet::weight(195_000_000)]
        pub fn reject_proposal(
            origin: OriginFor<T>,
            nonce: DepositNonce,
            src_id: BridgeChainId,
            r_id: ResourceId,
            call: Box<<T as Config>::Proposal>,
        ) -> DispatchResult {
```

```
            let who = ensure_signed(origin)?;
            ensure!(Self::is_relayer(&who), Error::<T>::MustBeRelayer);
            ensure!(Self::chain_whitelisted(src_id), Error::<T>::ChainNotWhitelisted);
            ensure!(Self::resource_exists(r_id), Error::<T>::ResourceDoesNotExist);

            Self::vote_against(who, nonce, src_id, call)
        }

        /// Evaluate the state of a proposal given the current vote threshold.
        ///
        /// A proposal with enough votes will be either executed or cancelled, and the status
        /// will be updated accordingly.
        ///
        /// # <weight>
        /// - weight of proposed call, regardless of whether execution is performed
        /// # </weight>
        #[pallet::weight({
            let dispatch_info = prop.get_dispatch_info();
            (dispatch_info.weight + 195_000_000, dispatch_info.class, Pays::Yes)
        })]
        pub fn eval_vote_state(
            origin: OriginFor<T>,
            nonce: DepositNonce,
            src_id: BridgeChainId,
            prop: Box<<T as Config>::Proposal>,
        ) -> DispatchResult {
            ensure_signed(origin)?;

            Self::try_resolve_proposal(nonce, src_id, prop)
        }
    }
        // *** Proposal voting and execution methods ***

        /// Commits a vote for a proposal. If the proposal doesn't exist it will be created.
        fn commit_vote(
```

```
            who: T::AccountId,
            nonce: DepositNonce,
            src_id: BridgeChainId,
            prop: Box<T::Proposal>,
            in_favour: bool,
        ) -> DispatchResult {
            let now = <frame_system::Pallet<T>>::block_number();
            let mut votes = match Votes::<T>::get(src_id, (nonce, prop.clone())) {
                Some(v) => v,
                None =>
                    ProposalVotes { expiry: now + T::ProposalLifetime::get(), ..Default::default() },
            };

            // Ensure the proposal isn't complete and relayer hasn't already voted
            ensure!(!votes.is_complete(), Error::<T>::ProposalAlreadyComplete);
            ensure!(!votes.is_expired(now), Error::<T>::ProposalExpired);
            ensure!(!votes.has_voted(&who), Error::<T>::RelayerAlreadyVoted);

            if in_favour {
                votes.votes_for.push(who.clone());
                Self::deposit_event(Event::VoteFor { chain_id: src_id, nonce, voter: who });
            } else {
                votes.votes_against.push(who.clone());
                Self::deposit_event(Event::VoteAgainst { chain_id: src_id, nonce, voter: who });
            }

            Votes::<T>::insert(src_id, (nonce, prop), votes);

            Ok(())
        }

        /// Attempts to finalize or cancel the proposal if the vote count allows.
        fn try_resolve_proposal(
            nonce: DepositNonce,
```

```
            src_id: BridgeChainId,
            prop: Box<T::Proposal>,
        ) -> DispatchResult {
            if let Some(mut votes) = Votes::<T>::get(src_id, (nonce, prop.clone())) {
                let now = <frame_system::Pallet<T>>::block_number();
                ensure!(!votes.is_complete(), Error::<T>::ProposalAlreadyComplete);
                ensure!(!votes.is_expired(now), Error::<T>::ProposalExpired);

                let status =
                    votes.try_to_complete(RelayerThreshold::<T>::get(),
RelayerCount::<T>::get());
                Votes::<T>::insert(src_id, (nonce, prop.clone()), votes);

                match status {
                    ProposalStatus::Approved  =>  Self::finalize_execution(src_id,  nonce,
prop),
                    ProposalStatus::Rejected => Self::cancel_execution(src_id, nonce),
                    _ => Ok(()),
                }
            } else {
                Err(Error::<T>::ProposalDoesNotExist.into())
            }
        }

        /// Commits a vote in favour of the proposal and executes it if the vote threshold is
met.
        fn vote_for(
            who: T::AccountId,
            nonce: DepositNonce,
            src_id: BridgeChainId,
            prop: Box<T::Proposal>,
        ) -> DispatchResult {
            Self::commit_vote(who, nonce, src_id, prop.clone(), true)?;
            Self::try_resolve_proposal(nonce, src_id, prop)
        }
```

```
        /// Commits a vote against the proposal and cancels it if more than (relayers.len() -
        /// threshold) votes against exist.
        fn vote_against(
            who: T::AccountId,
            nonce: DepositNonce,
            src_id: BridgeChainId,
            prop: Box<T::Proposal>,
        ) -> DispatchResult {
            Self::commit_vote(who, nonce, src_id, prop.clone(), false)?;
            Self::try_resolve_proposal(nonce, src_id, prop)
        }

        /// Execute the proposal and signals the result as an event
        #[allow(clippy::boxed_local)]
        fn finalize_execution(
            src_id: BridgeChainId,
            nonce: DepositNonce,
            call: Box<T::Proposal>,
        ) -> DispatchResult {
            Self::deposit_event(Event::ProposalApproved { chain_id: src_id, nonce });
            call.dispatch(frame_system::RawOrigin::Signed(Self::account_id()).into())
                .map(|_| ())
                .map_err(|e| e.error)?;
            Self::deposit_event(Event::ProposalSucceeded { chain_id: src_id, nonce });
            Ok(())
        }

        /// Cancels a proposal.
        fn cancel_execution(src_id: BridgeChainId, nonce: DepositNonce) -> DispatchResult
{
            Self::deposit_event(Event::ProposalRejected { chain_id: src_id, nonce });
            Ok(())
        }

        /// Initiates a transfer of a fungible asset out of the chain. This should be called by
        /// another pallet.
```

```
pub fn transfer_fungible(
    dest_id: BridgeChainId,
    resource_id: ResourceId,
    to: Vec<u8>,
    amount: U256,
) -> DispatchResult {
    ensure!(Self::chain_whitelisted(dest_id), Error::<T>::ChainNotWhitelisted);
    let nonce = Self::bump_nonce(dest_id);
    BridgeEvents::<T>::append(BridgeEvent::FungibleTransfer(
        dest_id,
        nonce,
        resource_id,
        amount,
        to.clone(),
    ));
    Self::deposit_event(Event::FungibleTransfer {
        dest_id,
        nonce,
        resource_id,
        amount,
        to,
    });
    Ok(())

}
```

Safety advice：NONE.

5.6.11. Cross chain transaction logic design 【security】

Audit description： Check the public chain and audit the business logic design related to cross chain transactions.

Risk hazard： Logic design error.

Audit process： Audit the business logic design related to cross chain transactions of public chain source code.

Audit results： After audit, the business logic design related to the proposal in the public chain source code is correct.

```
        /// Change extra bridge transfer fee that user should pay
        #[pallet::weight(195_000_000)]
        pub fn change_fee(
            origin: OriginFor<T>,
            min_fee: BalanceOf<T>,
            fee_scale: u32,
            dest_id: bridge::BridgeChainId,
        ) -> DispatchResult {
            T::BridgeCommitteeOrigin::ensure_origin(origin)?;
            ensure!(fee_scale <= 1000u32, Error::<T>::InvalidFeeOption);
            BridgeFee::<T>::insert(dest_id, (min_fee, fee_scale));
            Self::deposit_event(Event::FeeUpdated { chain_id: dest_id, min_fee, fee_scale });
            Ok(())
        }

        /// Transfers some amount of the native token to some recipient on a (whitelisted)
        /// destination chain.
        #[pallet::weight(195_000_000)]
        #[transactional]
        pub fn transfer_native(
            origin: OriginFor<T>,
            amount: BalanceOf<T>,
            recipient: Vec<u8>,
            dest_id: bridge::BridgeChainId,
        ) -> DispatchResult {
            let source = ensure_signed(origin)?;
            ensure!(<bridge::Pallet<T>>::chain_whitelisted(dest_id),
Error::<T>::InvalidTransfer);
            let bridge_id = <bridge::Pallet<T>>::account_id();
            ensure!(BridgeFee::<T>::contains_key(&dest_id),
```

```
Error::<T>::FeeOptionsMissing);
                let fee = Self::calculate_fee(dest_id, amount);
                let free_balance = T::Currency::free_balance(&source);
                ensure!(free_balance >= (amount + fee), Error::<T>::InsufficientBalance);

                let imbalance = T::Currency::withdraw(
                        &source,
                        fee,
                        WithdrawReasons::FEE,
                        ExistenceRequirement::AllowDeath,
                )?;
                T::OnFeePay::on_unbalanced(imbalance);
                <T as Config>::Currency::transfer(
                        &source,
                        &bridge_id,
                        amount,
                        ExistenceRequirement::AllowDeath,
                )?;

                <bridge::Pallet<T>>::transfer_fungible(
                        dest_id,
                        T::NativeTokenResourceId::get(),
                        recipient,
                        U256::from(amount.saturated_into::<u128>()),
                )
        }

        /// Returns some amount of the native token as proposal rejected
        #[pallet::weight(195_000_000)]
        pub fn return_transfer_native(
                origin: OriginFor<T>,
                amount: BalanceOf<T>,
                to: <T::Lookup as StaticLookup>::Source,
        ) -> DispatchResult {
                T::BridgeCommitteeOrigin::ensure_origin(origin)?;
                let to = T::Lookup::lookup(to)?;
```

```
        let bridge_id = <bridge::Pallet<T>>::account_id();
        <T as Config>::Currency::transfer(
            &bridge_id,
            &to,
            amount,
            ExistenceRequirement::AllowDeath,
        )?;
        Ok(())
    }

    /// Executes a simple currency transfer using the bridge account as the source
    #[pallet::weight(195_000_000)]
    pub fn transfer(
        origin: OriginFor<T>,
        to: T::AccountId,
        amount: BalanceOf<T>,
        rid: ResourceId,
    ) -> DispatchResult {
        let source = T::BridgeOrigin::ensure_origin(origin)?;
        // transfer to bridge account from external accounts is not allowed.
        if source == to {
            fail!(Error::<T>::InvalidCommand);
        }

        if rid == T::NativeTokenResourceId::get() {
            // ERC20 DEER transfer
            <T as Config>::Currency::transfer(
                &source,
                &to,
                amount,
                ExistenceRequirement::AllowDeath,
            )?;
        } else {
            fail!(Error::<T>::InvalidResourceId);
        }
```

```
            Ok(())
        }
    }
```

Safety advice：NONE.

5.6.12. NFT transaction logic design 【security】

Audit description： Check the business logic design related to NFT creation, NFT issuance, NFT destruction, NFT transaction, order query, order construction transaction in the public chain for audit.

Risk hazard： Logic design error.

Audit process： Audit the business logic design related to NFT transactions in the public chain source code.

Audit results： After audit, the NFT related business logic design in the public chain source code is correct.

```
    fn create_class_deposit(&self, bytes_len: u32) -> Result<BalanceInfo<Balance>> {
        let api = self.client.runtime_api();
        let at = BlockId::hash(self.client.info().best_hash);
        api.create_class_deposit(&at, bytes_len).map_err(|e| RpcError {
            code: ErrorCode::ServerError(Error::RuntimeError.into()),
            message: "Unable to query dispatch info.".into(),
            data: Some(format!("{:?}", e).into()),
        })
    }

    fn mint_token_deposit(&self, bytes_len: u32) -> Result<BalanceInfo<Balance>> {
        let api = self.client.runtime_api();
        let at = BlockId::hash(self.client.info().best_hash);
        api.mint_token_deposit(&at, bytes_len).map_err(|e| RpcError {
```

```
                code: ErrorCode::ServerError(Error::RuntimeError.into()),
                message: "Unable to query dispatch info.".into(),
                data: Some(format!("{:?}", e).into()),
            })
        }
fn mint_token<T: Config<I>, I: 'static>(
        class_id: T::ClassId,
        quantity: T::Quantity,
) -> (T::TokenId, T::Quantity, T::AccountId) {
        let caller = Classes::<T, I>::get(T::ClassId::default()).unwrap().owner;
        if caller != whitelisted_caller() {
            whitelist_account!(caller);
        }
        let to: <T::Lookup as StaticLookup>::Source = T::Lookup::unlookup(caller.clone());
        assert_ok!(NFT::<T, I>::mint(
            SystemOrigin::Signed(caller.clone()).into(),
            to,
            class_id,
            quantity,
            vec![0, 0, 0],
            Some(rate(10)),
            None,
        ));
        let token_id = NextTokenId::<T, I>::get(&class_id).saturating_sub(One::one());
        (token_id, quantity, caller)
}
            /// Create NFT(non fungible token) class
            #[pallet::weight(T::WeightInfo::create_class())]
            #[transactional]
            pub fn create_class(
                origin: OriginFor<T>,
                metadata: Vec<u8>,
                #[pallet::compact] royalty_rate: Perbill,
                permission: ClassPermission,
            ) -> DispatchResult {
                let owner = ensure_signed(origin)?;
```

```
            ensure!(T::RoyaltyRateLimit::get() >= royalty_rate, Error::<T, I>::RoyaltyRateTooHigh);

            let class_id =
                NextClassId::<T, I>::try_mutate(|id| -> Result<T::ClassId, DispatchError> {
                    let current_id = *id;
                    *id = id.checked_add(&One::one()).ok_or(Error::<T, I>::NoAvailableClassId)?;
                    Ok(current_id)
                })?;

            let deposit = Self::create_class_deposit(metadata.len().saturated_into());

            T::Currency::reserve(&owner, deposit)?;

            let class_details = ClassDetails {
                owner: owner.clone(),
                deposit,
                permission,
                metadata,
                total_tokens: Zero::zero(),
                total_issuance: Zero::zero(),
                royalty_rate,
            };

            Classes::<T, I>::insert(class_id, class_details);
            Self::deposit_event(Event::CreatedClass { class_id, owner });
            Ok(().into())
        }

        /// Mint NFT token by class owner
        #[pallet::weight(T::WeightInfo::mint())]
        #[transactional]
        pub fn mint(
            origin: OriginFor<T>,
            to: <T::Lookup as StaticLookup>::Source,
```

```
        #[pallet::compact] class_id: T::ClassId,
        #[pallet::compact] quantity: T::Quantity,
        metadata: Vec<u8>,
        royalty_rate: Option<Perbill>,
        royalty_beneficiary: Option<T::AccountId>,
    ) -> DispatchResult {
        let who = ensure_signed(origin)?;
        ensure!(quantity >= One::one(), Error::<T, I>::InvalidQuantity);
        let to = T::Lookup::lookup(to)?;
        Classes::<T, I>::try_mutate(&class_id, |maybe_class_details| -> DispatchResult {
            let class_details =
                maybe_class_details.as_mut().ok_or(Error::<T, I>::ClassNotFound)?;
            ensure!(&who == &class_details.owner, Error::<T, I>::NoPermission);
            Self::mint_token(
                class_details,
                &who,
                &to,
                class_id,
                quantity,
                metadata,
                royalty_rate,
                royalty_beneficiary,
            )
        })
    }

    /// Mint NFT token anyone else other than class owner
    #[pallet::weight(T::WeightInfo::delegate_mint())]
    #[transactional]
    pub fn delegate_mint(
        origin: OriginFor<T>,
        #[pallet::compact] class_id: T::ClassId,
        #[pallet::compact] quantity: T::Quantity,
        metadata: Vec<u8>,
        royalty_rate: Option<Perbill>,
        royalty_beneficiary: Option<T::AccountId>,
```

```
    ) -> DispatchResult {
        let who = ensure_signed(origin)?;
        ensure!(quantity >= One::one(), Error::<T, I>::InvalidQuantity);
        Classes::<T, I>::try_mutate(&class_id, |maybe_class_details| -> DispatchResult {
            let class_details =
                maybe_class_details.as_mut().ok_or(Error::<T, I>::ClassNotFound)?;
            ensure!(
                class_details.permission.0.contains(Permission::DelegateMintable),
                Error::<T, I>::NoPermission
            );
            let deposit = Self::mint_token_deposit(metadata.len().saturated_into());

            T::Currency::transfer(
                &who,
                &class_details.owner,
                deposit,
                ExistenceRequirement::KeepAlive,
            )?;
            Self::mint_token(
                class_details,
                &who,
                &who,
                class_id,
                quantity,
                metadata,
                royalty_rate,
                royalty_beneficiary,
            )
        })
    }

    /// Burn NFT token
    #[pallet::weight(T::WeightInfo::burn())]
    #[transactional]
    pub fn burn(
        origin: OriginFor<T>,
```

```
            #[pallet::compact] class_id: T::ClassId,
            #[pallet::compact] token_id: T::TokenId,
            #[pallet::compact] quantity: T::Quantity,
        ) -> DispatchResult {
            let owner = ensure_signed(origin)?;
            ensure!(quantity >= One::one(), Error::<T, I>::InvalidQuantity);

            Classes::<T, I>::try_mutate(&class_id, |maybe_class_details| -> DispatchResult {
                let class_details =
                    maybe_class_details.as_mut().ok_or(Error::<T, I>::ClassNotFound)?;

                ensure!(
                    class_details.permission.0.contains(Permission::Burnable),
                    Error::<T, I>::NoPermission
                );

                let token_details = Tokens::<T, I>::try_mutate_exists(
                    &class_id,
                    &token_id,
                    |maybe_token_details| -> Result<TokenDetailsOf<T, I>,
DispatchError> {
                        let token_details =
                            maybe_token_details.as_mut().ok_or(Error::<T,
I>::TokenNotFound)?;
                        token_details.quantity = token_details
                            .quantity
                            .checked_sub(&quantity)
                            .ok_or(Error::<T, I>::NumOverflow)?;
                        let copyed_token_details = token_details.clone();
                        if token_details.quantity.is_zero() {
                            *maybe_token_details = None;
                        }
                        Ok(copyed_token_details)
                    },
                )?;
```

```
                ensure!(token_details.consumers == 0, Error::<T, I>::ConsumerRemaining);

                if token_details.quantity.is_zero() {
                    T::Currency::unreserve(&class_details.owner, token_details.deposit);
                    class_details.total_tokens = class_details
                        .total_tokens
                        .checked_sub(&One::one())
                        .ok_or(Error::<T, I>::NumOverflow)?;
                }

                class_details.total_issuance = class_details
                    .total_issuance
                    .checked_sub(&quantity)
                    .ok_or(Error::<T, I>::NumOverflow)?;

                TokensByOwner::<T, I>::try_mutate_exists(
                    owner.clone(),
                    (class_id, token_id),
                    |maybe_token_amount| -> DispatchResult {
                        let mut token_amount = maybe_token_amount.unwrap_or_default();
                        token_amount.free = token_amount
                            .free
                            .checked_sub(&quantity)
                            .ok_or(Error::<T, I>::NumOverflow)?;
                        if token_amount.free.is_zero() && token_amount.reserved.is_zero() {
                            *maybe_token_amount = None;
                            OwnersByToken::<T, I>::remove((class_id, token_id), owner.clone());
                        } else {
                            *maybe_token_amount = Some(token_amount);
                        }
                        Ok(())
                    },
```

```
                )?;

                Self::deposit_event(Event::BurnedToken { class_id, token_id, quantity,
owner });
                Ok(().into())
            })
        }

        /// Update token royalty.
        #[pallet::weight(T::WeightInfo::update_token_royalty())]
        pub fn update_token_royalty(
            origin: OriginFor<T>,
            #[pallet::compact] class_id: T::ClassId,
            #[pallet::compact] token_id: T::TokenId,
            royalty_rate: Perbill,
        ) -> DispatchResult {
            let who = ensure_signed(origin)?;
            ensure!(T::RoyaltyRateLimit::get() >= royalty_rate, Error::<T,
I>::RoyaltyRateTooHigh);
            Tokens::<T, I>::try_mutate(
                class_id,
                token_id,
                |maybe_token_details| -> DispatchResult {
                    let token_details =
                        maybe_token_details.as_mut().ok_or(Error::<T,
I>::TokenNotFound)?;
                    ensure!(who == token_details.royalty_beneficiary, Error::<T,
I>::NoPermission);

                    let account_token =
                        Self::tokens_by_owner(&who, (class_id,
token_id)).unwrap_or_default();
                    ensure!(
                        account_token.reserved.is_zero() &&
                            account_token.free == token_details.quantity,
                        Error::<T, I>::NoPermission
```

```
                        );
                        token_details.royalty_rate = royalty_rate;

                        Self::deposit_event(Event::UpdatedToken { class_id, token_id });
                        Ok(().into())
                    },
                )
            }

            /// Update token royalty beneficiary.
            #[pallet::weight(T::WeightInfo::update_token_royalty_beneficiary())]
            pub fn update_token_royalty_beneficiary(
                origin: OriginFor<T>,
                #[pallet::compact] class_id: T::ClassId,
                #[pallet::compact] token_id: T::TokenId,
                to: <T::Lookup as StaticLookup>::Source,
            ) -> DispatchResult {
                let who = ensure_signed(origin)?;
                Tokens::<T, I>::try_mutate(
                    class_id,
                    token_id,
                    |maybe_token_details| -> DispatchResult {
                        let token_details =
                            maybe_token_details.as_mut().ok_or(Error::<T,
I>::TokenNotFound)?;
                        ensure!(who == token_details.royalty_beneficiary, Error::<T,
I>::NoPermission);
                        let to = T::Lookup::lookup(to)?;
                        token_details.royalty_beneficiary = to;

                        Self::deposit_event(Event::UpdatedToken { class_id, token_id });
                        Ok(().into())
                    },
                )
            }
```

```
        /// Transfer NFT tokens to another account
        ///
        /// - `to`: the token owner's account
        /// - `class_id`: class id
        /// - `token_id`: token id
        /// - `quantity`: quantity
        #[pallet::weight(T::WeightInfo::transfer())]
        #[transactional]
        pub fn transfer(
            origin: OriginFor<T>,
            #[pallet::compact] class_id: T::ClassId,
            #[pallet::compact] token_id: T::TokenId,
            #[pallet::compact] quantity: T::Quantity,
            to: <T::Lookup as StaticLookup>::Source,
        ) -> DispatchResult {
            let who = ensure_signed(origin)?;
            let to = T::Lookup::lookup(to)?;
            ensure!(quantity >= One::one(), Error::<T, I>::InvalidQuantity);

            Self::transfer_token(
                class_id,
                token_id,
                quantity,
                &who,
                &to,
                TransferReason::Direct,
                Zero::zero(),
            )?;
            Ok(())
        }
    }
    pub fn transfer_token(
        class_id: T::ClassId,
        token_id: T::TokenId,
        quantity: T::Quantity,
        from: &T::AccountId,
```

```
            to: &T::AccountId,
            reason: TransferReason,
            price: BalanceOf<T, I>,
        ) -> Result<bool, DispatchError> {
            if from == to || quantity.is_zero() {
                return Ok(false)
            }
            let token = (class_id, token_id);

            TokensByOwner::<T, I>::try_mutate_exists(
                from,
                token,
                |maybe_from_amount| -> Result<bool, DispatchError> {
                    let mut from_amount = maybe_from_amount.ok_or(Error::<T, I>::TokenNotFound)?;
                    from_amount.free =
                        from_amount.free.checked_sub(&quantity).ok_or(Error::<T, I>::NumOverflow)?;

                    TokensByOwner::<T, I>::try_mutate_exists(
                        to,
                        token,
                        |maybe_to_amount| -> DispatchResult {
                            match maybe_to_amount {
                                Some(to_amount) => {
                                    to_amount.free = to_amount
                                        .free
                                        .checked_add(&quantity)
                                        .ok_or(Error::<T, I>::NumOverflow)?;
                                },
                                None => {
                                    *maybe_to_amount =
                                        Some(TokenAmount { free: quantity, reserved: Zero::zero() });

                                    OwnersByToken::<T, I>::insert(token, to, ());
                                },
```

```
                        }
                        Ok(())
                    },
                )?;

                if from_amount.free.is_zero() && from_amount.reserved.is_zero() {
                    *maybe_from_amount = None;
                    OwnersByToken::<T, I>::remove(token, from);
                } else {
                    *maybe_from_amount = Some(from_amount);
                }

                Self::deposit_event(Event::TransferredToken {
                    class_id,
                    token_id,
                    quantity,
                    from: from.clone(),
                    to: to.clone(),
                    reason,
                    price,
                });

                Ok(true)
            },
        )
    }

    pub fn ensure_transferable(
        class_id: T::ClassId,
        token_id: T::TokenId,
        quantity: T::Quantity,
        owner: &T::AccountId,
    ) -> DispatchResult {
        let token_amount = Self::tokens_by_owner(owner, (class_id, token_id))
            .ok_or(Error::<T, I>::TokenNotFound)?;
        ensure!(token_amount.free >= quantity, Error::<T, I>::InvalidQuantity);
```

```
        let class_details = Classes::<T, I>::get(class_id).ok_or(Error::<T, I>::ClassNotFound)?;
        ensure!(
            class_details.permission.0.contains(Permission::Transferable),
            Error::<T, I>::NoPermission
        );
        Ok(())
    }

    pub fn inc_consumers(class_id: T::ClassId, token_id: T::TokenId) -> DispatchResult {
        Tokens::<T, I>::try_mutate(class_id, token_id, |maybe_token| {
            let token = maybe_token.as_mut().ok_or(Error::<T, I>::TokenNotFound)?;
            token.consumers = token.consumers.saturating_add(1);
            Ok(())
        })
    }

    pub fn dec_consumers(class_id: T::ClassId, token_id: T::TokenId) -> DispatchResult {
        Tokens::<T, I>::try_mutate(class_id, token_id, |maybe_token| {
            let token = maybe_token.as_mut().ok_or(Error::<T, I>::TokenNotFound)?;
            token.consumers = token.consumers.saturating_sub(1);
            Ok(())
        })
    }

    pub fn reserve(
        class_id: T::ClassId,
        token_id: T::TokenId,
        quantity: T::Quantity,
        owner: &T::AccountId,
    ) -> DispatchResult {
        TokensByOwner::<T, I>::try_mutate_exists(
            owner,
            (class_id, token_id),
            |maybe_amount| -> DispatchResult {
                let mut amount = maybe_amount.unwrap_or_default();
                amount.free =
```

```
                        amount.free.checked_sub(&quantity).ok_or(Error::<T,
I>::NumOverflow)?;
                    amount.reserved =
                        amount.reserved.checked_add(&quantity).ok_or(Error::<T,
I>::NumOverflow)?;
                    *maybe_amount = Some(amount);
                    Ok(())
                },
            )
        }

        pub fn unreserve(
            class_id: T::ClassId,
            token_id: T::TokenId,
            quantity: T::Quantity,
            owner: &T::AccountId,
        ) -> DispatchResult {
            TokensByOwner::<T, I>::try_mutate_exists(
                owner,
                (class_id, token_id),
                |maybe_amount| -> DispatchResult {
                    let mut amount = maybe_amount.unwrap_or_default();
                    amount.reserved =
                        amount.reserved.checked_sub(&quantity).ok_or(Error::<T,
I>::NumOverflow)?;
                    amount.free =
                        amount.free.checked_add(&quantity).ok_or(Error::<T,
I>::NumOverflow)?;
                    *maybe_amount = Some(amount);
                    Ok(())
                },
            )
        }

        pub fn swap(
            class_id: T::ClassId,
```

```
            token_id: T::TokenId,
            quantity: T::Quantity,
            from: &T::AccountId,
            to: &T::AccountId,
            price: BalanceOf<T, I>,
            tax_ratio: Perbill,
            reason: TransferReason,
        ) -> DispatchResult {
            let token = Tokens::<T, I>::get(class_id, token_id).ok_or(Error::<T, I>::TokenNotFound)?;
            Self::transfer_token(class_id, token_id, quantity, from, to, reason, price)?;
            let mut royalty_fee = token.royalty_rate * price;
            if royalty_fee < T::Currency::minimum_balance() &&
                T::Currency::free_balance(&token.royalty_beneficiary).is_zero()
            {
                royalty_fee = Zero::zero();
            }
            if !royalty_fee.is_zero() {
                T::Currency::transfer(
                    to,
                    &token.royalty_beneficiary,
                    royalty_fee,
                    ExistenceRequirement::KeepAlive,
                )?;
            }
            let tax_fee = tax_ratio * price;
            if !tax_fee.is_zero() {
                T::Currency::withdraw(
                    to,
                    tax_fee,
                    WithdrawReasons::TRANSFER,
                    ExistenceRequirement::KeepAlive,
                )?;
            }
            let order_fee = price.saturating_sub(royalty_fee).saturating_sub(tax_fee);
            T::Currency::transfer(to, from, order_fee, ExistenceRequirement::KeepAlive)?;
```

```
            Ok(())
        }

        pub fn create_class_deposit(bytes_len: u32) -> BalanceOf<T, I> {
            T::ClassDeposit::get().saturating_add(Self::caculate_byes_deposit(bytes_len))
        }

        pub fn mint_token_deposit(bytes_len: u32) -> BalanceOf<T, I> {
            T::TokenDeposit::get().saturating_add(Self::caculate_byes_deposit(bytes_len))
        }

        fn caculate_byes_deposit(bytes_len: u32) -> BalanceOf<T, I> {
            T::MetaDataByteDeposit::get().saturating_mul((bytes_len).into())
        }

        fn mint_token(
            class_details: &mut ClassDetailsOf<T, I>,
            who: &T::AccountId,
            to: &T::AccountId,
            class_id: T::ClassId,
            quantity: T::Quantity,
            metadata: Vec<u8>,
            royalty_rate: Option<Perbill>,
            royalty_beneficiary: Option<T::AccountId>,
        ) -> DispatchResult {
            NextTokenId::<T, I>::try_mutate(class_id, |id| -> DispatchResult {
                let royalty_rate = royalty_rate.unwrap_or(class_details.royalty_rate);
                ensure!(T::RoyaltyRateLimit::get() >= royalty_rate, Error::<T,
I>::RoyaltyRateTooHigh);

                let total_tokens = class_details
                    .total_tokens
                    .checked_add(&One::one())
                    .ok_or(Error::<T, I>::NumOverflow)?;

                let total_issuance = class_details
```

```
            .total_issuance
            .checked_add(&quantity)
            .ok_or(Error::<T, I>::NumOverflow)?;

        let token_id = *id;
        *id = id.checked_add(&One::one()).ok_or(Error::<T, I>::NoAvailableTokenId)?;

        class_details.total_tokens = total_tokens;
        class_details.total_issuance = total_issuance;

        let deposit = Self::mint_token_deposit(metadata.len().saturated_into());
        T::Currency::reserve(&class_details.owner, deposit)?;

        let token_details = TokenDetails {
            creator: who.clone(),
            metadata,
            deposit,
            quantity,
            consumers: 0,
            royalty_rate,
            royalty_beneficiary: royalty_beneficiary.unwrap_or(to.clone()),
        };
        Tokens::<T, I>::insert(&class_id, &token_id, token_details);
        TokensByOwner::<T, I>::insert(
            &to,
            (class_id, token_id),
            TokenAmount { free: quantity, reserved: Zero::zero() },
        );
        OwnersByToken::<T, I>::insert((class_id, token_id), &to, ());

        Self::deposit_event(Event::MintedToken {
            class_id,
            token_id,
            quantity,
            owner: to.clone(),
            caller: who.clone(),
```

```
                });

                Ok(())
            })
        }
}
    pub fn migrate<T: Config<I>, I: 'static>() -> Weight {
        log::info!(
            target: "runtime::nft",
            "Migrating nft to Releases::V2",
        );
        let pallet_name = <Pallet<T, I>>::name().as_bytes();

        let mut class_count: u32 = 0;
        let mut token_count: u32 = 0;
        let mut attribute_count: u32 = 0;

        let permission = ClassPermission(
            Permission::Burnable | Permission::Transferable |
Permission::DelegateMintable,
        );
        let mut max_class_id: T::ClassId = Zero::zero();
        for (class_id, p) in Class::<T, I>::drain() {
            let (metadata, count) = attributes_to_metadata::<T, I>(class_id, None);
            attribute_count += count;
            let new_class_details = ClassDetails {
                owner: p.owner,
                deposit: p.deposit,
                permission,
                metadata,
                total_tokens: p.instances.saturated_into(),
                total_issuance: p.instances.saturated_into(),
                royalty_rate: p.royalty_rate,
            };
            Classes::<T, I>::insert(class_id, new_class_details);
            if class_id > max_class_id {
```

```
                max_class_id = class_id;
            }
            class_count += 1;
        }

        let mut max_token_id_map: BTreeMap<T::ClassId, T::TokenId> = BTreeMap::new();
        let zero = Zero::zero();

        for (class_id, token_id, p) in Asset::<T, I>::drain() {
            let (metadata, count) = attributes_to_metadata::<T, I>(class_id, Some(token_id));
            attribute_count += count;
            let new_token_details = TokenDetails {
                creator: p.owner.clone(),
                metadata,
                deposit: p.deposit,
                quantity: One::one(),
                consumers: 0,
                royalty_rate: p.royalty_rate,
                royalty_beneficiary: p.royalty_beneficiary,
            };
            let mut token_amount: TokenAmount<T::Quantity> = Default::default();
            if p.reserved {
                token_amount.reserved = One::one();
            } else {
                token_amount.free = One::one();
            }
            let max_token_id = max_token_id_map.get(&class_id).unwrap_or(&zero);
            if token_id > *max_token_id {
                max_token_id_map.insert(class_id, token_id);
            }

            Tokens::<T, I>::insert(class_id, token_id, new_token_details);
            TokensByOwner::<T, I>::insert(p.owner.clone(), (class_id, token_id), token_amount);
            OwnersByToken::<T, I>::insert((class_id, token_id), p.owner, ());
```

```
                token_count += 1;
            }

            for (class_id, max_token_id) in max_token_id_map.iter() {
                NextTokenId::<T, I>::insert(*class_id,
max_token_id.saturating_add(One::one()));
            }

            migration::remove_storage_prefix(pallet_name, b"AssetTransfer", b"");
            migration::remove_storage_prefix(pallet_name, b"Account", b"");
            migration::remove_storage_prefix(pallet_name, b"Attribute", b"");
            migration::remove_storage_prefix(pallet_name, b"MaxClassId", b"");
            NextClassId::<T, I>::put(max_class_id.saturating_add(One::one()));

            StorageVersion::<T, I>::put(Releases::V2);

            log::info!(
                target: "runtime::nft",
                "Migrate {:?} classes, {:?} tokens",
                class_count,
                token_count,
            );

            T::DbWeight::get().reads_writes(
                (class_count + token_count + attribute_count) as Weight,
                (class_count * 2 + token_count * 3 + 5) as Weight,
            )
        }

        #[cfg(feature = "try-runtime")]
        pub fn post_migrate<T: Config<I>, I: 'static>() -> Result<(), &'static str> {
            assert!(StorageVersion::<T, I>::get() == Releases::V2);
            log::debug!(
                target: "runtime::nft",
                "migration: nft storage version v2 POST migration checks succesful!",
            );
```

```
            for (owner, (class_id, token_id), _) in TokensByOwner::<T, I>::iter() {
                assert!(
                    OwnersByToken::<T, I>::get((class_id, token_id), owner.clone()).is_some()
&&
                        Tokens::<T, I>::get(class_id, token_id).is_some(),
                    "invalid token ({:?} {:?})",
                    class_id,
                    token_id
                );
            }
            assert_eq!(Class::<T, I>::iter().count(), 0);
            assert_eq!(Asset::<T, I>::iter().count(), 0);
            Ok(())
        }

        fn attributes_to_metadata<T: Config<I>, I: 'static>(
            class_id: T::ClassId,
            token_id: Option<T::TokenId>,
        ) -> (Vec<u8>, u32) {
            let mut count = 0;
            let mut pairs: Vec<Vec<u8>> = vec![];
            for key in Attribute::<T, I>::iter_key_prefix((class_id, token_id)) {
                if let Some((value, _)) = Attribute::<T, I>::get((class_id, token_id, &key)) {
                    pairs.push([b"\"", &key[..], b"\":\"", &value[..], b"\""].concat());
                }
                count += 1;
            }
            let content = pairs.join(&b',');
            ([b"{", &content[..], b"}"].concat(), count)
        }
    }
    fn create_nft<T: Config<I>, I: 'static>(
        owner: &T::AccountId,
    ) -> (T::ClassId, T::TokenId, T::Quantity) {
        let quantity = One::one();
        let permission = ClassPermission(
            Permission::Burnable | Permission::Transferable | Permission::DelegateMintable,
```

```
        );
        let class_id = NextClassId::<T, I>::get();
        assert_ok!(NFT::<T, I>::create_class(
            SystemOrigin::Signed(owner.clone()).into(),
            vec![0, 0, 0],
            rate(10),
            permission,
        ));
        let to: <T::Lookup as StaticLookup>::Source = T::Lookup::unlookup(owner.clone());
        let token_id = NextTokenId::<T, I>::get(&class_id);
        assert_ok!(NFT::<T, I>::mint(
            SystemOrigin::Signed(owner.clone()).into(),
            to,
            class_id,
            quantity,
            vec![0, 0, 0],
            None,
            None
        ));
        (class_id, token_id, quantity)
}
/// Order detail
#[derive(Clone, Encode, Decode, Eq, PartialEq, RuntimeDebug, TypeInfo)]
pub struct OrderDetails<ClassId, TokenId, Quantity, Balance, BlockNumber> {
        /// Nft class id
        #[codec(compact)]
        pub class_id: ClassId,
        /// Nft token id
        #[codec(compact)]
        pub token_id: TokenId,
        /// Amount of tokens in sale
        #[codec(compact)]
        pub quantity: Quantity,
        /// Total amount of tokens
        #[codec(compact)]
        pub total_quantity: Quantity,
```

```
        /// Price of this order.
        pub price: Balance,
        /// The balances to create an order
        pub deposit: Balance,
        /// This order will be invalidated after `deadline` block number.
        pub deadline: Option<BlockNumber>,
}

/// Offer detail
#[derive(Clone, Encode, Decode, Eq, PartialEq, RuntimeDebug, TypeInfo)]
pub struct OfferDetails<ClassId, TokenId, Quantity, Balance, BlockNumber> {
        /// Nft class id
        #[codec(compact)]
        pub class_id: ClassId,
        /// Nft token id
        #[codec(compact)]
        pub token_id: TokenId,
        /// Amount of tokens
        #[codec(compact)]
        pub quantity: Quantity,
        /// Price of this order.
        pub price: Balance,
        /// This order will be invalidated after `deadline` block number.
        pub deadline: Option<BlockNumber>,
}
            /// Create a order to sell a non-fungible asset
            #[pallet::weight(<T as Config<I>>::WeightInfo::sell())]
            #[transactional]
            pub fn sell(
                origin: OriginFor<T>,
                #[pallet::compact] class_id: T::ClassId,
                #[pallet::compact] token_id: T::TokenId,
                #[pallet::compact] quantity: T::Quantity,
                #[pallet::compact] price: BalanceOf<T, I>,
                deadline: Option<T::BlockNumber>,
            ) -> DispatchResult {
```

```
                let who = ensure_signed(origin)?;
                pallet_nft::Pallet::<T, I>::ensure_transferable(class_id, token_id, quantity, &who)?;
                if let Some(ref deadline) = deadline {
                    ensure!(
                        <frame_system::Pallet<T>>::block_number() < *deadline,
                        Error::<T, I>::InvalidDeadline
                    );
                }
                NextOrderId::<T, I>::try_mutate(|id| -> DispatchResult {
                    let order_id = *id;
                    *id = id.checked_add(&One::one()).ok_or(Error::<T, I>::NoAvailableOrderId)?;

                    T::Currency::reserve(&who, T::OrderDeposit::get())?;
                    pallet_nft::Pallet::<T, I>::reserve(class_id, token_id, quantity, &who)?;
                    let order = OrderDetails {
                        class_id,
                        token_id,
                        quantity,
                        total_quantity: quantity,
                        deposit: T::OrderDeposit::get(),
                        price,
                        deadline,
                    };
                    Orders::<T, I>::insert(who.clone(), order_id, order);

                    Self::deposit_event(Event::CreatedOrder {
                        order_id,
                        class_id,
                        token_id,
                        quantity,
                        seller: who,
                    });
                    Ok(())
                })
```

```
        }

        /// Deal an order
        #[pallet::weight(<T as Config<I>>::WeightInfo::deal_order())]
        #[transactional]
        pub fn deal_order(
            origin: OriginFor<T>,
            order_owner: <T::Lookup as StaticLookup>::Source,
            #[pallet::compact] order_id: T::OrderId,
            #[pallet::compact] quantity: T::Quantity,
        ) -> DispatchResult {
            let who = ensure_signed(origin)?;
            let seller = T::Lookup::lookup(order_owner)?;

            Orders::<T, I>::try_mutate_exists(
                seller.clone(),
                order_id,
                |maybe_order| -> DispatchResult {
                    let order = maybe_order.as_mut().ok_or(Error::<T, I>::OrderNotFound)?;
                    let order_quantity = order.quantity;

                    ensure!(
                        quantity <= order_quantity && quantity >= One::one(),
                        Error::<T, I>::InvalidQuantity
                    );

                    if let Some(ref deadline) = order.deadline {
                        ensure!(
                            <frame_system::Pallet<T>>::block_number() <= *deadline,
                            Error::<T, I>::OrderExpired
                        );
                    }

                    let fee = Perbill::from_rational(quantity, order.total_quantity) * order.price;
```

```
                ensure!(
                    T::Currency::free_balance(&who) >= fee,
                    Error::<T, I>::InsufficientFunds
                );

                let class_id = order.class_id;
                let token_id = order.token_id;
                pallet_nft::Pallet::<T, I>::unreserve(class_id, token_id, quantity,
&seller)?;
                pallet_nft::Pallet::<T, I>::swap(
                    class_id,
                    token_id,
                    quantity,
                    &seller,
                    &who,
                    fee,
                    T::TradeFeeTaxRatio::get(),
                    pallet_nft::TransferReason::Order,
                )?;

                if quantity == order_quantity {
                    T::Currency::unreserve(&seller, order.deposit);
                    *maybe_order = None;
                } else {
                    order.quantity = order.quantity.saturating_sub(quantity);
                }

                Self::deposit_event(Event::DealedOrder {
                    order_id,
                    seller,
                    buyer: who,
                    quantity,
                    fee,
                });
                Ok(())
```

```
                },
            )
        }

        /// Remove an order
        #[pallet::weight(<T as Config<I>>::WeightInfo::remove_order())]
        #[transactional]
        pub fn remove_order(
            origin: OriginFor<T>,
            #[pallet::compact] order_id: T::OrderId,
        ) -> DispatchResult {
            let who = ensure_signed(origin)?;
            Orders::<T, I>::try_mutate_exists(
                who.clone(),
                order_id,
                |maybe_order| -> DispatchResult {
                    let order = maybe_order.as_mut().ok_or(Error::<T, I>::OrderNotFound)?;

                    let class_id = order.class_id;
                    let token_id = order.token_id;
                    let quantity = order.quantity;

                    pallet_nft::Pallet::<T, I>::unreserve(class_id, token_id, quantity, &who)?;
                    T::Currency::unreserve(&who, order.deposit);

                    *maybe_order = None;

                    Self::deposit_event(Event::RemovedOrder { order_id, seller: who });
                    Ok(())
                },
            )
        }

        /// Create a offer to buy a non-fungible asset
```

```
        #[pallet::weight(<T as Config<I>>::WeightInfo::buy())]
        #[transactional]
        pub fn buy(
            origin: OriginFor<T>,
            #[pallet::compact] class_id: T::ClassId,
            #[pallet::compact] token_id: T::TokenId,
            #[pallet::compact] quantity: T::Quantity,
            #[pallet::compact] price: BalanceOf<T, I>,
            deadline: Option<T::BlockNumber>,
        ) -> DispatchResult {
            let who = ensure_signed(origin)?;
            if let Some(ref deadline) = deadline {
                ensure!(
                    <frame_system::Pallet<T>>::block_number() < *deadline,
                    Error::<T, I>::InvalidDeadline
                );
            }

            pallet_nft::Pallet::<T, I>::inc_consumers(class_id, token_id)?;

            NextOfferId::<T, I>::try_mutate(|id| -> DispatchResult {
                let offer_id = *id;
                *id = id.checked_add(&One::one()).ok_or(Error::<T,
I>::NoAvailableOfferId)?;

                T::Currency::reserve(&who, price)?;
                let offer = OfferDetails { class_id, token_id, quantity, price, deadline };
                Offers::<T, I>::insert(who.clone(), offer_id, offer);

                Self::deposit_event(Event::CreatedOffer {
                    offer_id,
                    class_id,
                    token_id,
                    quantity,
                    buyer: who,
                });
```

```
                Ok(())
            })
        }

        /// Deal an offer
        #[pallet::weight(<T as Config<I>>::WeightInfo::deal_offer())]
        #[transactional]
        pub fn deal_offer(
            origin: OriginFor<T>,
            offer_owner: <T::Lookup as StaticLookup>::Source,
            #[pallet::compact] offer_id: T::OrderId,
        ) -> DispatchResult {
            let owner = ensure_signed(origin)?;
            let buyer = T::Lookup::lookup(offer_owner)?;

            Offers::<T, I>::try_mutate_exists(
                buyer.clone(),
                offer_id,
                |maybe_offer| -> DispatchResult {
                    let offer = maybe_offer.as_mut().ok_or(Error::<T, I>::OfferNotFound)?;

                    if let Some(ref deadline) = offer.deadline {
                        ensure!(
                            <frame_system::Pallet<T>>::block_number() <= *deadline,
                            Error::<T, I>::OfferExpired
                        );
                    }

                    T::Currency::unreserve(&buyer, offer.price);

                    let class_id = offer.class_id;
                    let token_id = offer.token_id;
                    let quantity = offer.quantity;

                    pallet_nft::Pallet::<T, I>::dec_consumers(class_id, token_id)?;
```

```
                pallet_nft::Pallet::<T, I>::swap(
                    class_id,
                    token_id,
                    quantity,
                    &owner,
                    &buyer,
                    offer.price,
                    T::TradeFeeTaxRatio::get(),
                    pallet_nft::TransferReason::Offer,
                )?;

                Self::deposit_event(Event::DealedOffer {
                    offer_id,
                    buyer,
                    seller: owner,
                    quantity,
                    fee: offer.price,
                });

                *maybe_offer = None;
                Ok(())
            },
        )
    }

    /// Remove an offer
    #[pallet::weight(<T as Config<I>>::WeightInfo::remove_offer())]
    #[transactional]
    pub fn remove_offer(
        origin: OriginFor<T>,
        #[pallet::compact] offer_id: T::OrderId,
    ) -> DispatchResult {
        let who = ensure_signed(origin)?;
        Offers::<T, I>::try_mutate_exists(
            who.clone(),
```

```
                    offer_id,
                    |maybe_offer| -> DispatchResult {
                        let offer = maybe_offer.as_mut().ok_or(Error::<T, I>::OfferNotFound)?;

                        let class_id = offer.class_id;
                        let token_id = offer.token_id;

                        pallet_nft::Pallet::<T, I>::dec_consumers(class_id, token_id)?;

                        T::Currency::unreserve(&who, offer.price);

                        *maybe_offer = None;

                        Self::deposit_event(Event::RemovedOffer { offer_id, buyer: who });
                        Ok(())
                    },
                )
            }
        }
```

Safety advice: **NONE.**

5.6.13. Data storage logic design【security】

Audit description: Audit the business logic design related to data storage in the public chain source code.

Risk hazard: Logic design error.

Audit process: Audit the business logic design related to data storage in the public chain source code.

Audit results: After audit, the business logic design related to data storage in the public chain source code is correct.

```
fn create_file<T: Config>(
	cid: &FileId,
	no_reserved: bool,
	replicas: &[T::AccountId],
	liquidate_at: BlockNumberFor<T>,
) {
	let reserved = if no_reserved {
		0u32.saturated_into()
	} else {
		FileStorage::<T>::store_file_bytes_fee(1_000_000)
	};
	Files::<T>::insert(
		cid.clone(),
		FileInfo {
			reserved,
			base_fee: T::FileBaseFee::get(),
			file_size: 1_000_000u64,
			add_at: 99u32.saturated_into(),
			fee: FileStorage::<T>::store_file_bytes_fee(1_000_000),
			liquidate_at,
			replicas: replicas.to_vec(),
		},
	);
}

fn create_replica_nodes<T: Config>(
	num_replicas: u32,
	seed: u32,
	node: Option<T::AccountId>,
) -> Vec<T::AccountId> {
	let mut nodes = match node {
		Some(node) => vec![node],
		None => vec![],
	};
	for i in 0..num_replicas {
		let node: T::AccountId = account("replica", i, seed);
```

```
            Nodes::<T>::insert(
                node.clone(),
                NodeInfo {
                    stash: node.clone(),
                    deposit: T::Currency::minimum_balance().saturating_mul(1000u32.into()),
                    machine_id: Some(vec![0u8; 16]),
                    rid: 0,
                    used: 10000000,
                    slash_used: 0,
                    reward: 0u32.into(),
                    power: 10000000000,
                    reported_at: Zero::zero(),
                    prev_reported_at: Zero::zero(),
                },
            );
            nodes.push(node);
        }
        nodes
}

benchmarks! {
        set_enclave {
            let enclave_id = get_enclave();
            let expire_at = 100u32.into();
        }: _(SystemOrigin::Root, enclave_id.clone(), expire_at)
        verify {
            assert_last_event::<T>(Event::<T>::SetEnclave { enclave_id, expire_at }.into());
        }

        stash {
            let stasher = create_funded_user::<T>("stasher", 20000);
            let controller: T::AccountId = account("controller", 0, SEED);
            whitelist_account!(controller);
            let controller_lookup: <T::Lookup as StaticLookup>::Source =
T::Lookup::unlookup(controller.clone());
        }: _(SystemOrigin::Signed(stasher.clone()), controller_lookup)
```

```
	verify {
		assert!(Nodes::<T>::contains_key(&controller));
	}

	withdraw {
		fund_storage_pot::<T>(20000u32.into());
		let stasher = create_funded_user::<T>("stasher", 20000);
		let controller: T::AccountId = account("controller", 0, SEED);
		whitelist_account!(controller);
		let controller_lookup: <T::Lookup as StaticLookup>::Source =
T::Lookup::unlookup(controller.clone());
		assert_ok!(FileStorage::<T>::stash(SystemOrigin::Signed(stasher.clone()).into(),
controller_lookup));
		let amount =
T::Currency::minimum_balance().saturating_mul(10000u32.saturated_into());
		Nodes::<T>::mutate(&controller, |maybe_node_info| {
			if let Some(node_info) = maybe_node_info {
				node_info.deposit = node_info.deposit.saturating_add(amount);
			}
		});
	}: _(SystemOrigin::Signed(controller.clone()))
	verify {
		assert_last_event::<T>(Event::<T>::Withdrawn { controller, stash: stasher,
amount }.into());
	}

	register {
		let enclave = get_enclave();
		assert_ok!(FileStorage::<T>::set_enclave(SystemOrigin::Root.into(), enclave,
1000000u32.into()));

		let stasher = create_funded_user::<T>("stasher", 20000);
		let controller: T::AccountId = account("controller", 0, SEED);
		whitelist_account!(controller);
		let controller_lookup: <T::Lookup as StaticLookup>::Source =
T::Lookup::unlookup(controller.clone());
```

```
        assert_ok!(FileStorage::<T>::stash(SystemOrigin::Signed(stasher.clone()).into(),
controller_lookup));
        let machine_id: Vec<u8> = hex!("2663554671a5f2c3050e1cec37f31e55").into();
        let ias_body =
str2bytes("{\"id\":\"327849746623058382595462695863525135492\",\"timestamp\":\"2021-
07-21T07:23:39.696594\",\"version\":4,\"epidPseudonym\":\"ybSBDhwKvtRlx76tLCjLNVH+zl
6JLGEEuu/c0mcQwk0OGYFRSsJfLApOkp+B/GFAzhTIIEXmYmAOSGDdbc2mFu/wx1HiK1+mFl
+isaCe6ZN7IeLOrfbnVfeR6E7OhvFtc9e1xwyviVa6a9+bCVhQV1THJq7IW7HbaOxW9ZQu6g0
=\",\"advisoryURL\":\"https://security-center.intel.com\",\"advisoryIDs\":[\"INTEL-SA-00161\
",\"INTEL-SA-00477\",\"INTEL-SA-00381\",\"INTEL-SA-00389\",\"INTEL-SA-00320\",\"INTEL
-SA-00329\",\"INTEL-SA-00220\",\"INTEL-SA-00270\",\"INTEL-SA-00293\",\"INTEL-SA-002
33\"],\"isvEnclaveQuoteStatus\":\"GROUP_OUT_OF_DATE\",\"platformInfoBlob\":\"15020065
040009000011110204018007000000000000000000C00000C000000020000000000000B2F
D11FE6C355B3AB0F453E92C88F565CB58ACDCA00D3E13716CE6BDB92A372DA547849872
93BE9EF77C00D94F090A9193BD6147A3C994E3086D14C57C089F35D39\",\"isvEnclaveQuot
eBody\":\"AgABAC8LAAAMAAsAAAAAAAbkva5mzdO2S8iey0QRTKEAAAAAAAAAAAAAAAA
AAAAABRICBf+AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAABwAAAAAAAAAHAAAAAAAAAPmJXfzjBbElHCQkIXgTZKSee1RznLfSzwv1eOTzk7+j
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAACD1xnnferKFHD2uvYqTXdDA
8iZ22kCD5xw7h38CMfOngAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAACH9m21/gilxl3atpQAIEkv
0v5hVBPxPY2RMcR4xoxsgN+kc2W/n++sKQA8+PFpoHZis8WQdRHpnkOc3mnzlv+C\"}");
        let ias_sig =
str2bytes("OcghuZnUiFmEs85hC0Ri2uJfyWR6IhhuCKY/U3UJTRee8GiENQCNj9dAQEYuUbUG
4qEhdJeW4sM3RhV1MuOgYjut6UYXnhGXLDVg48ba+L+IDRQng+E26JYnQ0MOv0mMMJC
NX1l3mHTUHM8e0C/kIWQJ+esuhR6G4WuHp7xyReZfJGbuKAkc6tC+q7e9XU9HvbSRaowjIfF
MrXgJUZh5VG3Cj+6rDi807rL9oAxFTweivHiz6Tcvp3aZ7pH2QpDBL9OD68gwYfDxGvBi6+S1c
hqI7P6pFfWHcT+CISbOo2M6p9HpSVLf/07/9xxCrDU2/M5hDxSlVbXqKQKW2Mxt8A==");
        let ias_cert =
str2bytes("MIIEoTCCAwmgAwIBAgIJANEHdl0yo7CWMA0GCSqGSIb3DQEBCwUAMH4xCzAJ
BgNVBAYTAlVTMQswCQYDVQQIDAJDQTEUMBIGA1UEBwwLU2FudGEgQ2xhcmExGjAYBgN
VBAoMEUludGVsIENvcnBvcmF0aW9uMTAwLgYDVQQDDCdJbnRlbCBTR1ggQXR0ZXN0YXR
pb24gUmVwb3J0IFNpZ25pbmcgQ0EwHhcNMTYxMTIyMDkzNjU4WhcNMjYxMTIwMDkzNjU
4WjB7MQswCQYDVQQGEwJVUzELMAkGA1UECAwCQ0ExFDASBgNVBAcMC1NhbnRhIENsY
```

```
XJhMRowGAYDVQQKDBFJbnRlbCBDb3Jwb3JhdGlvbjEtMCsGA1UEAwwkSW50ZWwgU0dYIE
F0dGVzdGF0aW9uIFJlcG9ydCBTaWduaW5nMIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBC
gKCAQEAqXot4OZuphR8nudFrAFiaGxxkgma/Es/BA+tbeCTUR106AL1ENcWA4FX3K+E9BBL0
/7X5rj5nIgX/R/1ubhkKWw9gfqPG3KeAtIdcv/uTO1yXv50vqaPvE1CRChvzdS/ZEBqQ5oVvLTP
Z3VEicQjlytKgN9cLnxbwtuvLUK7eyRPfJW/ksddOzP8VBBniolYnRCD2jrMRZ8nBM2ZWYwnXn
wYeOAHV+W9tOhAImwRwKF/95yAsVwd21ryHMJBcGH70qLagZ7Ttyt++qO/6+KAXJuKwZqj
RlEtSEz8gZQeFfVYgcwSfo96oSMAzVr7V0L6HSDLRnpb6xxmbPdqNol4tQIDAQABo4GkMIGh
MB8GA1UdIwQYMBaAFHhDe3amfrzQr35CN+s1fDuHAVE8MA4GA1UdDwEB/wQEAwIGwDA
MBgNVHRMBAf8EAjAAMGAGA1UdHwRZMFcwVaBToFGGT2h0dHA6Ly90cnVzdGVkc2Vydm
ljZXMuaW50ZWwuY29tL2NvbnRlbnQvQ1JML1NHWC9BdHRlc3RhdGlvblJlcG9ydFNpZ25pb
mdDQS5jcmwwDQYJKoZIhvcNAQELBQADggGBAGcIthtcK9IVRz4rRq+ZKE+7k50/OxUsmW8
aavOzKb0iCx07YQ9rzi5nU73tME2yGRLzhSViFs/LpFa9lpQL6JL1aQwmDR74TxYGBAIi5f4I5TJo
CCEqRHz91kpG6Uvyn2tLmnIdJbPE4vYvWLrtXXfFBSSPD4Afn7+3/XUggAlc7oCTizOfbbtOFIY
A4g5KcYgS1J2ZAeMQqbUdZseZCcaZZZn65tdqee8UXZlDvx0+NdO0LR+5pFy+juM0wWbu5
9MvzcmTXbjsi7HY6zd53Yq5K244fwFHRQ8eOB0IWB+4PfM7FeAApZvlfqlKOlLcZL2uyVmzRky
R5yW72uo9mehX44CiPJ2fse9Y6eQtcfEhMPkmHXI01sN+KwPbpA39+xOsStjhP9N1Y1a2tQA
Vo+yVgLgV2Hws73Fc0o3wC78qPEA+v2aRs/Be3ZFDgDyghc/1fgU+7C+P6kbqd4poyb6IW8
KCJbxfMJvkordNOgOUUxndPHEi/tb/U7uLjLOgPA==");
        let sig =
hex!("90639853f8e815ede625c0b786c8453230790193aa5b29f5dca76e48845344503c8373a
5cd9536d02504e0d74dfaef791af7f65e081a7be827f6d5e492424ca4").into();
    }: _(SystemOrigin::Signed(controller.clone()), machine_id.clone(), ias_cert, ias_sig,
ias_body, sig)
    verify {
        assert_last_event::<T>(Event::<T>::NodeRegistered { controller, machine_id }.into());
    }

    report {
        let x in 0..T::MaxFileReplicas::get();
        let y in 0..T::MaxFileReplicas::get();

        System::<T>::set_block_number(50000u32.into());

        let enclave = get_enclave();
        assert_ok!(FileStorage::<T>::set_enclave(SystemOrigin::Root.into(), enclave,
1000000u32.into()));
```

```
        let stasher = create_funded_user::<T>("stasher", 20000);
        let controller: T::AccountId = account("controller", 0, SEED);
        whitelist_account!(controller);
        let controller_lookup: <T::Lookup as StaticLookup>::Source =
T::Lookup::unlookup(controller.clone());
        assert_ok!(FileStorage::<T>::stash(SystemOrigin::Signed(stasher.clone()).into(),
controller_lookup));
        let machine_id: Vec<u8> = hex!("2663554671a5f2c3050e1cec37f31e55").into();
        let ias_body =
str2bytes("{\"id\":\"327849746623058382595462695863525135492\",\"timestamp\":\"2021-
07-21T07:23:39.696594\",\"version\":4,\"epidPseudonym\":\"ybSBDhwKvtRlx76tLCjLNVH+zl
6JLGEEuu/c0mcQwk0OGYFRSsJfLApOkp+B/GFAzhTIIEXmYmAOSGDdbc2mFu/wx1HiK1+mFl
+isaCe6ZN7leLOrfbnVfeR6E7OhvFtc9e1xwyviVa6a9+bCVhQV1THJq7lW7HbaOxW9ZQu6g0
=\",\"advisoryURL\":\"https://security-center.intel.com\",\"advisoryIDs\":[\"INTEL-SA-00161\
",\"INTEL-SA-00477\",\"INTEL-SA-00381\",\"INTEL-SA-00389\",\"INTEL-SA-00320\",\"INTEL
-SA-00329\",\"INTEL-SA-00220\",\"INTEL-SA-00270\",\"INTEL-SA-00293\",\"INTEL-SA-002
33\"],\"isvEnclaveQuoteStatus\":\"GROUP_OUT_OF_DATE\",\"platformInfoBlob\":\"15020065
040009000011110204018007000000000000000000C00000C000000020000000000000B2F
D11FE6C355B3AB0F453E92C88F565CB58ACDCA00D3E13716CE6BDB92A372DA547849872
93BE9EF77C00D94F090A9193BD6147A3C994E3086D14C57C089F35D39\",\"isvEnclaveQuot
eBody\":\"AgABAC8LAAAMAAsAAAAAAAbkva5mzdO2S8iey0QRTKEAAAAAAAAAAAAAAAA
AAAAABRICBf+AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAABwAAAAAAAAHAAAAAAAAAPmJXfzjBbElHCQklXgTZKSee1RznLfSzwv1eOTzk7+j
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAACD1xnnferKFHD2uvYqTXdDA
8iZ22kCD5xw7h38CMfOngAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAACH9m21/gilxl3atpQAIEkv
0v5hVBPxPY2RMcR4xoxsgN+kc2W/n++sKQA8+PFpoHZis8WQdRHpnkOc3mnzlv+C\"}");
        let ias_sig =
str2bytes("OcghuZnUiFmEs85hC0Ri2uJfyWR6lhhuCKY/U3UJTRee8GiENQCNj9dAQEYuUbUG
4qEhdJeW4sM3RhV1MuOgYjut6UYXnhGXLDVg48ba+L+IDRQng+E26JYnQ0MOv0mMMJC
NX1l3mHTUHM8e0C/klWQJ+esuhR6G4WuHp7xyReZfJGbuKAkc6tC+q7e9XU9HvbSRaowjIfF
MrXgJUZh5VG3Cj+6rDi807rL9oAxFTweivHiz6Tcvp3aZ7pH2QpDBL9OD68gwYfDxGvBi6+S1c
hqI7P6pFfWHcT+ClSbOo2M6p9HpSVLf/07/9xxCrDU2/M5hDxSlVbXqKQKW2Mxt8A==");
```

```
        let ias_cert =
str2bytes("MIIEoTCCAwmgAwIBAgIJANEHdl0yo7CWMA0GCSqGSIb3DQEBCwUAMH4xCzAJ
BgNVBAYTAlVTMQswCQYDVQQIDAJDQTEUMBIGA1UEBwwLU2FudGEgQ2xhcmExGjAYBgN
VBAoMEUludGVsIENvcnBvcmF0aW9uMTAwLgYDVQQDDCdJbnRlbCBTR1ggQXR0ZXN0YXR
pb24gUmVwb3J0IFNpZ25pbmcgQ0EwHhcNMTYxMTIyMDkzNjU4WhcNMjYxMTIwMDkzNjU
4WjB7MQswCQYDVQQGEwJVUzELMAkGA1UECAwCQ0ExFDASBgNVBAcMC1NhbnRhIENsY
XJhMRowGAYDVQQKDBFJbnRlbCBDb3Jwb3JhdGlvbjEtMCsGA1UEAwwkSW50ZWwgU0dYIE
F0dGVzdGF0aW9uIFJlcG9ydCBTaWduaW5nMIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBC
gKCAQEAqXot4OZuphR8nudFrAFiaGxxkgma/Es/BA+tbeCTUR106AL1ENcWA4FX3K+E9BBL0
/7X5rj5nIgX/R/1ubhkKWw9gfqPG3KeAtIdcv/uTO1yXv50vqaPvE1CRChvzdS/ZEBqQ5oVvLTP
Z3VEicQjlytKgN9cLnxbwtuvLUK7eyRPfJW/ksddOzP8VBBniolYnRCD2jrMRZ8nBM2ZWYwnXn
wYeOAHV+W9tOhAImwRwKF/95yAsVwd21ryHMJBcGH70qLagZ7Ttyt++qO/6+KAXJuKwZqj
RlEtSEz8gZQeFfVYgcwSfo96oSMAzVr7V0L6HSDLRnpb6xxmbPdqNol4tQIDAQABo4GkMIGh
MB8GA1UdIwQYMBaAFHhDe3amfrzQr35CN+s1fDuHAVE8MA4GA1UdDwEB/wQEAwIGwDA
MBgNVHRMBAf8EAjAAMGAGA1UdHwRZMFcwVaBToFGGT2h0dHA6Ly90cnVzdGVkc2Vydm
ljZXMuaW50ZWwuY29tL2NvbnRlbnQvQ1JML1NHWC9BdHRlc3RhdGlvblJlcG9ydFNpZ25pb
mdDQS5jcmwwDQYJKoZIhvcNAQELBQADggGBAGcIthtcK9IVRz4rRq+ZKE+7k50/OxUsmW8
aavOzKb0iCx07YQ9rzi5nU73tME2yGRLzhSViFs/LpFa9lpQL6JL1aQwmDR74TxYGBAIi5f4I5TJo
CCEqRHz91kpG6Uvyn2tLmnIdJbPE4vYvWLrtXXfFBSSPD4Afn7+3/XUggAlc7oCTizOfbbtOFIY
A4g5KcYgS1J2ZAeMQqbUdZseZCcaZZZn65tdqee8UXZIDvx0+NdO0LR+5pFy+juM0wWbu5
9MvzcmTXbjsi7HY6zd53Yq5K244fwFHRQ8eOB0IWB+4PfM7FeAApZvlfqlKOlLcZL2uyVmzRky
R5yW72uo9mehX44CiPJ2fse9Y6eQtcfEhMPkmHXI01sN+KwPbpA39+xOsStjhP9N1Y1a2tQA
Vo+yVgLgV2Hws73Fc0o3wC78qPEA+v2aRs/Be3ZFDgDyghc/1fgU+7C+P6kbqd4poyb6IW8
KCJbxfMJvkordNOgOUUxndPHEi/tb/U7uLjLOgPA==");
        let sig =
hex!("90639853f8e815ede625c0b786c8453230790193aa5b29f5dca76e48845344503c8373a
5cd9536d02504e0d74dfaef791af7f65e081a7be827f6d5e492424ca4").into();

        assert_ok!(FileStorage::<T>::register(SystemOrigin::Signed(controller.clone()).into(),
machine_id.clone(), ias_cert, ias_sig, ias_body, sig));

        let mut add_files = vec![];
        for i in 0 .. x {
            let a = ((i / 26) / 26 % 26 + 97) as u8;
            let b = ((i / 26) % 26 + 97) as u8;
            let c = ((i % 26) + 97) as u8;
```

```
                let suffix: Vec<u8> = vec![a, b, c];
                let cid: FileId = [&FILE_ID_PREFIX[..], &suffix[..]].concat();
                create_file::<T>(&cid, false, &[], 0u32.into());
                add_files.push((cid, 1_000_000));
            }
            let mut del_files = vec![];
            for i in 0 .. y {
                let a = ((i / 26) / 26 % 26 + 65) as u8;
                let b = ((i / 26) % 26 + 97) as u8;
                let c = ((i % 26) + 97) as u8;
                let suffix: Vec<u8> = vec![a, b, c];
                let cid: FileId = [&FILE_ID_PREFIX[..], &suffix[..]].concat();
                let replicas = create_replica_nodes::<T>(T::EffectiveFileReplicas::get(), 2000u32
+ i as u32, Some(controller.clone()));
                create_file::<T>(&cid, false, &replicas, 1000u32.into());
                del_files.push(cid);
            }
            let liquidate_files = vec![];
            let rid = 1000;
            let power = 1000_000_000;
            let                priv_k:                Vec<u8>                =
hex!("e394cf1de366242a772f44904ba475f5317ce8baedac5485ccd812db2ccf28ab").into();
            let                pub_k:                Vec<u8>                =
hex!("87f66db5fe0888c65ddab6940020492fd2fe615413f13d8d9131c478c68c6c80dfa47365b
f9fefac29003cf8f169a07662b3c5907511e99e439cde69f396ff82").into();

            let sig: Vec<u8> = sign::p256_sign(
                &machine_id,
                &priv_k,
                &pub_k,
                0,
                rid,
                &add_files,
                &del_files,
                power,
            );
```

```
    }: _(SystemOrigin::Signed(controller.clone()), rid, power, sig, add_files, del_files,
liquidate_files)
    verify {
        assert!(Nodes::<T>::contains_key(&controller));
    }


    store {
        let cid = str2bytes("QmS9ErDVxHXRNMJRJ5i3bp1zxCZzKP8QXXNH1yeeeeeeeA");
        let caller = create_funded_user::<T>("caller", 10000);
        let fee = T::Currency::minimum_balance().saturating_mul(2000u32.saturated_into());
    }: _(SystemOrigin::Signed(caller.clone()), cid.clone(), 100u64, fee)
    verify {
        assert_last_event::<T>(Event::<T>::FileAdded { cid, caller, fee, first: true }.into());
    }

    force_delete {
        let cid = str2bytes("QmS9ErDVxHXRNMJRJ5i3bp1zxCZzKP8QXXNH1yeeeeeeeA");
        let caller = create_funded_user::<T>("caller", 10000);
        let fee = T::Currency::minimum_balance().saturating_mul(2000u32.saturated_into());
        assert_ok!(FileStorage::<T>::store(SystemOrigin::Signed(caller.clone()).into(),
cid.clone(), 100u64, fee));
        System::<T>::set_block_number(50000u32.into());
    }: _(SystemOrigin::Root, cid.clone())
    verify {
        assert_last_event::<T>(Event::<T>::FileForceDeleted { cid }.into());
    }

    session_end {
        Summarys::<T>::insert(0, SummaryInfo { power: 100 * MB2, used: 10 *
MB2, ..Default::default() });
        FileStorage::<T>::session_end();
        Summarys::<T>::insert(1, SummaryInfo { power: 100 * MB2, used: 10 *
MB2, ..Default::default() });
        FileStorage::<T>::session_end();
        Summarys::<T>::insert(2, SummaryInfo { power: 100 * MB2, used: 10 *
```

```
MB2, ..Default::default() });
	}: {
		FileStorage::<T>::session_end();
	}
	verify {
		assert_eq!(Session::<T>::get().current, 3);
	}
}
		/// Add file to storage
		#[pallet::weight(T::WeightInfo::store())]
		pub fn store(
			origin: OriginFor<T>,
			cid: FileId,
			file_size: u64,
			fee: BalanceOf<T>,
		) -> DispatchResult {
			let who = ensure_signed(origin)?;
			ensure!(
				file_size > 0 && file_size <= T::MaxFileSize::get(),
				Error::<T>::InvalidFileSize
			);
			ensure!(is_cid(&cid), Error::<T>::InvalidCid);

			if let Some(mut file) = Files::<T>::get(&cid) {
				let new_reserved = fee.saturating_add(file.reserved);
				let min_fee = Self::store_file_bytes_fee(file.file_size);
				ensure!(new_reserved >= min_fee, Error::<T>::NotEnoughFee);
				T::Currency::transfer(
					&who,
					&Self::account_id(),
					fee,
					ExistenceRequirement::KeepAlive,
				)?;
				file.reserved = new_reserved;
				Files::<T>::insert(cid.clone(), file);
				Self::deposit_event(Event::<T>::FileAdded { cid, caller: who, fee, first:
```

```
false });
                } else {
                    let min_fee = Self::store_file_fee(file_size);
                    ensure!(fee >= min_fee, Error::<T>::NotEnoughFee);
                    T::Currency::transfer(
                        &who,
                        &Self::account_id(),
                        fee,
                        ExistenceRequirement::KeepAlive,
                    )?;
                    let base_fee = T::FileBaseFee::get();
                    Files::<T>::insert(
                        cid.clone(),
                        FileInfo {
                            reserved: fee.saturating_sub(base_fee),
                            base_fee,
                            file_size,
                            add_at: Self::now_at(),
                            fee: Zero::zero(),
                            liquidate_at: Zero::zero(),
                            replicas: vec![],
                        },
                    );
                    Self::deposit_event(Event::<T>::FileAdded { cid, caller: who, fee, first:
true });
                }
                Ok(())
            }

            /// Force delete unsoloved file
            #[pallet::weight(T::WeightInfo::force_delete())]
            pub fn force_delete(origin: OriginFor<T>, cid: FileId) -> DispatchResult {
                T::ForceOrigin::ensure_origin(origin)?;

                if let Some(file) = Files::<T>::get(&cid) {
                    let now = Self::now_at();
```

```
                    ensure!(
                        !file.base_fee.is_zero() &&
                            now > file.add_at.saturating_add(T::LiquidateDuration::get()),
                        Error::<T>::UnableToDeleteFile
                    );
                    StoragePotReserved::<T>::mutate(|v| {
                        *v = v.saturating_add(file.base_fee).saturating_add(file.reserved)
                    });
                    Files::<T>::remove(&cid);
                    Self::deposit_event(Event::<T>::FileForceDeleted { cid });
                }
                Ok(())
            }
        }
        fn report_add_file(ctx: &mut ReportContextOf<T>, cid: &FileId, file_size: u64) {
            if let Some(mut file) = Files::<T>::get(cid) {
                if !file.liquidate_at.is_zero() {
                    let mut new_nodes = vec![];
                    let mut is_included = false;
                    for (index, replica_account) in file.replicas.iter().enumerate() {
                        let replica_node = ctx
                            .node_infos
                            .entry(replica_account.clone())
                            .or_insert_with(||
Nodes::<T>::get(replica_account).unwrap_or_default());
                        if Self::is_prev_reported(replica_node, &ctx.session) {
                            new_nodes.push(replica_account.clone());
                        } else {
                            let node_change =

        ctx.node_changes.entry(replica_account.clone()).or_default();
                            if (index as u32) < T::MaxFileReplicas::get() {
                                node_change.slash_used_dec =
                                    node_change.slash_used_dec.saturating_add(file_size);
                            } else {
                                node_change.used_dec                                    =
```

```
node_change.used_dec.saturating_add(file_size);
                                }
                            }
                            if replica_account == &ctx.reporter {
                                is_included = true;
                            }
                        }
                        if !is_included && (new_nodes.len() as u32) < T::MaxFileReplicas::get() {
                            new_nodes.push(ctx.reporter.clone());
                            let                                node_change                                =
ctx.node_changes.entry(ctx.reporter.clone()).or_default();
                            node_change.used_inc                                =
node_change.used_inc.saturating_add(file_size);
                        }
                        file.replicas = new_nodes;
                        Files::<T>::insert(cid, file);
                    } else {
                        let is_file_exist = Self::liquidate_file(
                            ctx,
                            cid,
                            &mut file,
                            vec![ctx.reporter.clone()],
                            Some(file_size),
                        );
                        if is_file_exist {
                            let                                node_change                                =
ctx.node_changes.entry(ctx.reporter.clone()).or_default();
                            node_change.used_inc                                =
node_change.used_inc.saturating_add(file_size);
                        }
                    }
                }
            }

            fn report_delete_file(ctx: &mut ReportContextOf<T>, cid: &FileId) {
                if let Some(mut file) = Files::<T>::get(cid) {
```

```
                if let Ok(index) = file.replicas.binary_search(&ctx.reporter) {
                    file.replicas.remove(index);
                    let node_change =
ctx.node_changes.entry(ctx.reporter.clone()).or_default();
                    if (index as u32) < T::MaxFileReplicas::get() {
                        node_change.slash_used_dec =
                            node_change.slash_used_dec.saturating_add(file.file_size);
                    } else {
                        node_change.used_dec =
node_change.used_dec.saturating_add(file.file_size);
                    }
                    Files::<T>::insert(cid, file);
                }
            }
        }

        fn report_liquidate_file(ctx: &mut ReportContextOf<T>, cid: &FileId) {
            if let Some(mut file) = Files::<T>::get(cid) {
                if file.liquidate_at.is_zero() || file.liquidate_at > ctx.now_at {
                    return
                }

                let file_fee = file.fee;
                let mut total_order_reward: BalanceOf<T> = Zero::zero();
                let each_order_reward = Self::share_ratio() * file_fee;
                let mut replicas = vec![];
                for (index, replica_account) in file.replicas.iter().enumerate() {
                    let replica_node = ctx
                        .node_infos
                        .entry(replica_account.clone())
                        .or_insert_with(||
Nodes::<T>::get(replica_account).unwrap_or_default());
                    if Self::is_prev_reported(replica_node, &ctx.session) {
                        let node_change =
ctx.node_changes.entry(replica_account.clone()).or_default();
                        node_change.reward =
```

```
node_change.reward.saturating_add(each_order_reward);
                        total_order_reward                                          =
total_order_reward.saturating_add(each_order_reward);
                        if replica_account == &ctx.reporter {
                            node_change.reward                                      =
node_change.reward.saturating_add(each_order_reward);
                            total_order_reward                                      =
total_order_reward.saturating_add(each_order_reward);
                        }
                        replicas.push(replica_account.clone());
                    } else {
                        let                         node_change                     =
ctx.node_changes.entry(replica_account.clone()).or_default();
                        if (index as u32) < T::MaxFileReplicas::get() {
                            node_change.slash_used_dec =
                                node_change.slash_used_dec.saturating_add(file.file_size);
                        } else {
                            node_change.used_dec                                    =
node_change.used_dec.saturating_add(file.file_size);
                        }
                    }
                }
                let is_file_exist = Self::liquidate_file(ctx, cid, &mut file, replicas.clone(), None);
                if !is_file_exist {
                    for node in replicas.iter() {
                        let node_change = ctx.node_changes.entry(node.clone()).or_default();
                        node_change.used_dec                                        =
node_change.used_dec.saturating_add(file.file_size);
                    }
                }
                let unpaid_reward = file_fee.saturating_sub(total_order_reward);
                if !unpaid_reward.is_zero() {
                    ctx.session_store_reward                                        =
ctx.session_store_reward.saturating_add(unpaid_reward);
                }
            }
```

```
    }

    fn liquidate_file(
        ctx: &mut ReportContextOf<T>,
        cid: &FileId,
        file: &mut FileInfoOf<T>,
        nodes: Vec<T::AccountId>,
        maybe_file_size: Option<u64>,
    ) -> bool {
        let first = file.liquidate_at.is_zero();
        let expect_order_fee =
            Self::store_file_bytes_fee(maybe_file_size.unwrap_or(file.file_size));
        if let Some(file_size) = maybe_file_size {
            if first {
                ctx.storage_pot_add = ctx.storage_pot_add.saturating_add(file.base_fee);
                // user underreported the file size
                if file.file_size < file_size && file.reserved < expect_order_fee {
                    let to_reporter_reward = Self::share_ratio() * file.reserved;
                    let node_change = ctx.node_changes.entry(ctx.reporter.clone()).or_default();
                    node_change.reward = node_change.reward.saturating_add(to_reporter_reward);
                    ctx.session_store_reward = ctx
                        .session_store_reward

    .saturating_add(file.reserved.saturating_sub(to_reporter_reward));
                    Self::delete_file(cid);
                    return false
                }
                file.base_fee = Zero::zero();
                file.file_size = file_size;
            }
        }
        let (order_fee, new_reserved) = if file.reserved > expect_order_fee {
            (expect_order_fee, file.reserved.saturating_sub(expect_order_fee))
        } else {
```

```
                (file.reserved, Zero::zero())
            };
            if order_fee.is_zero() {
                Self::delete_file(cid);
                return false
            }
            let now_at = Self::now_at();
            let mut duration = T::LiquidateDuration::get();
            if order_fee < expect_order_fee {
                duration = Perbill::from_rational(order_fee, expect_order_fee) * duration;
            }
            if first {
                Self::deposit_event(Event::<T>::FileStored { cid: cid.clone() });
            }

            file.fee = order_fee;
            file.file_size = file.file_size;
            file.liquidate_at = now_at.saturating_add(duration);
            file.replicas = nodes;
            file.reserved = new_reserved;
            Files::<T>::insert(cid, file);
            return true
        }

        fn is_reported(node_info: &NodeInfoOf<T>, session: &SessionStateOf<T>) -> bool {
            !node_info.reported_at.is_zero() && node_info.reported_at >= session.begin_at
        }

        fn is_prev_reported(node_info: &NodeInfoOf<T>, session: &SessionStateOf<T>) ->
bool {
            if session.prev_begin_at.is_zero() {
                return true
            }
            if Self::is_reported(node_info, session) {
                return node_info.prev_reported_at >= session.prev_begin_at
            } else {
```

```
                return node_info.reported_at >= session.prev_begin_at
            }
        }

        fn delete_file(cid: &FileId) {
            Files::<T>::remove(cid);
            Self::deposit_event(Event::<T>::FileDeleted { cid: cid.clone() });
        }

        /// Reserved deposit balance for node's used storage space
        fn deposit_for_used(space: u64) -> BalanceOf<T> {
            Self::share_ratio() * Self::store_file_bytes_fee(space)
        }

        fn store_file_fee(file_size: u64) -> BalanceOf<T> {
            T::FileBaseFee::get().saturating_add(Self::store_file_bytes_fee(file_size))
        }

        fn share_ratio() -> Perbill {
            Perbill::from_rational(1, T::EffectiveFileReplicas::get().saturating_add(1)) *
                T::StoreRewardRatio::get()
        }

        fn store_file_bytes_fee(file_size: u64) -> BalanceOf<T> {
            let mut file_size_in_mega = file_size / 1_048_576;
            if file_size % 1_048_576 != 0 {
                file_size_in_mega += 1;
            }
            T::FileSizePrice::get().saturating_mul(file_size_in_mega.saturated_into())
        }
```

Safety advice： NONE.

5.7. Other safety issues

5.7.1. File permission security 【security】

Audit description： Check whether the permissions of relevant log files and keystore files are set correctly.

Risk hazard： File permission is one of the lowest security settings of the system, which ensures that files can be operated by available users. Improper file permission settings may lead to file tampering, information disclosure and other risks.

Audit process： Check whether the permissions of relevant log files and keystore files are set correctly.

Audit results： After detection, there is no security problem with the file permission setting in the public chain code.

```
ubuntu@ubuntu:~/deer-node/data/chains/deert1/db/full$ ls -al
total 775928
drwxrwxr-x 2 ubuntu ubuntu    20480 Jul 15 05:29 .
drwxrwxr-x 3 ubuntu ubuntu     4096 Jul 15 05:14 ..
-rw-r--r-- 1 ubuntu ubuntu  2499984 Jul 15 05:15 000040.sst
-rw-r--r-- 1 ubuntu ubuntu   400157 Jul 15 05:15 000045.sst
-rw-r--r-- 1 ubuntu ubuntu   402047 Jul 15 05:15 000057.sst
-rw-r--r-- 1 ubuntu ubuntu   337900 Jul 15 05:15 000066.sst
-rw-r--r-- 1 ubuntu ubuntu   405333 Jul 15 05:15 000069.sst
-rw-r--r-- 1 ubuntu ubuntu   408130 Jul 15 05:15 000082.sst
-rw-r--r-- 1 ubuntu ubuntu   407944 Jul 15 05:15 000091.sst
-rw-r--r-- 1 ubuntu ubuntu   345632 Jul 15 05:15 000098.sst
-rw-r--r-- 1 ubuntu ubuntu   404975 Jul 15 05:16 000109.sst
-rw-r--r-- 1 ubuntu ubuntu   404836 Jul 15 05:16 000124.sst
```

Safety advice： NONE.

5.7.2. Concurrent security risks 【security】

Audit description： Check whether there are concurrent security risks in map, slice and other operations in the public chain code.

Risk hazard： When multiple threads share data, it may produce inconsistent

results, which may cause data loss, value coverage and other problems.

Audit process: Check whether there are concurrent security risks in map, slice and other operations in the public chain code.

Audit results: After audit, there is no such safety problem.

Safety advice: NONE.

5.7.3. Centralization degree detection 【security】

Audit description: Identify whether there is over centralized design in the system design.

Risk hazard: Centralized design is easy to concentrate risks. Once a single point of failure occurs, it can lead to serious problems such as data tampering and chain forking.

Audit process: Identify whether there is over centralized design in the system design.

Audit results: After audit, there is no such safety problem.

Safety advice: NONE.

5.7.4. Third party dependence on Security 【security】

Audit description: The development of the public chain project cannot avoid the use of third-party libraries. While introducing third-party libraries to enrich functions and strengthen the business processing capacity of the system, it also introduces the security problems of corresponding libraries, such as our common

third-party dependent libraries such as fastjson and Jackson.

Risk hazard： Remote command execution, sensitive information disclosure.

Audit process： Conduct security screening on the third-party dependencies used in the public chain to detect whether there is a risk pool.

Audit results： After audit, deer network does not use any risky third-party libraries.

Safety advice： NONE.

6. Appendix: analysis tools

6.1. Gosec

Gosec checks whether the source code has security problems by scanning go ast. It can scan for hard coded credentials, not check audit errors, use un escaped data in HTML templates, create temporary files using predictable paths, extract file traversal during zip archiving, and other security problems. It can be configured to only run rule subsets, exclude some file paths, and generate reports in different formats.

6.2. Fortify

Fortify is a powerful static code scanning and analysis tool. Its ability to find code vulnerabilities is very strong, mainly by compiling the code and relying on its powerful built-in rule base to find vulnerabilities. Secondly, when developing this business tool, the force SCA team also provides an interface for custom rules. As long as it is authorized by the genuine version, it can customize rules on this basis to enhance the vulnerability identification ability of fortify SCA. At the same time, through custom rules, it can also reduce false positives and improve the accuracy and efficiency of static analysis.

6.3. Safesql

Golang static analysis tool can detect database security problems such as SQL

injection.

6.4. Flawfinder

Flawfinder is an open source static scanning and analysis tool for c/c++. It performs static search according to the internal dictionary database and matches simple defects and vulnerabilities. Flawfinder tool does not need to compile c/c++ code, and can directly scan and analyze.

6.5. Sublime Text-IDE

Sublime text supports syntax highlighting in multiple programming languages, has excellent code auto completion function, and also has the function of snippet, which can save commonly used code snippets and call them whenever necessary. VIM mode is supported, and most commands in VIM mode can be used. Support macros, which simply means recording operations or writing commands by yourself, and then playing the operations or commands just recorded.

7. DISCLAIMERS

Chainlion only issues this report on the facts that have occurred or existed before the issuance of this report, and assumes corresponding responsibilities. For the facts occurring or existing after the issuance, chainlion cannot judge the security

status of its smart contract, and is not responsible for it. The security audit analysis and other contents in this report are only based on the documents and materials provided by the information provider to chainlion as of the issuance of this report. Chainlion assumes that the information provided is not missing, tampered with, deleted or concealed. If the information provided is missing, tampered with, deleted, concealed or reflected inconsistent with the actual situation, chainlion shall not be liable for the losses and adverse effects caused thereby. Chainlion only conducted the agreed safety audit on the safety of the project and issued this report. Chainlion is not responsible for the background and other conditions of the project.